

Principes d'Implantation de Scheme et Lisp

Christian Queinnec

Principes d'Implantation de Lisp et Scheme

Christian Queinnec

Table des matières

Avis au lecteur **xiii**
Post-préface . . . xx

1 Fondements de l'interprétation **1**
1.1 Évaluation . . . 2
1.2 Évaluateur fondamental . . . 3
1.3 Évaluation des atomes . . . 4
1.4 Évaluation des formes . . . 6
1.4.1 Citation . . . 7
1.4.2 Alternative . . . 8
1.4.3 Séquence . . . 9
1.4.4 Affectation . . . 10
1.4.5 Abstraction . . . 11
1.4.6 Application fonctionnelle . . . 11
1.5 Représentation de l'environnement . . . 12
1.6 Représentation des fonctions . . . 14
1.6.1 Liaisons dynamique et lexicale . . . 18
1.6.2 Implantation superficielle ou profonde . . . 22
1.7 Environnement global . . . 23
1.8 Lancement de l'interprète . . . 26
1.9 Conclusion . . . 26
1.10 Exercices . . . 27
1.11 Lectures recommandées . . . 28

2 Lisp 1, 2 ... ω **29**
2.1 Lisp_1 . . . 30
2.2 Lisp_2 . . . 30
2.2.1 Évaluation du terme fonctionnel . . . 33
2.2.2 Dualité des mondes . . . 34
2.2.3 Jouer en Lisp_2 . . . 35
2.2.4 Enrichir l'environnement fonctionnel . . . 36
2.3 Autres extensions . . . 37
2.4 Comparaison entre Lisp_1 et Lisp_2 . . . 38
2.5 Espaces de noms . . . 40

2.5.1 Variables dynamiques 41
2.5.2 Variables dynamiques en COMMON LISP 45
2.5.3 Variables dynamiques sans forme spéciale 47
2.5.4 Conclusions 49
2.6 Récursion 50
2.6.1 Récursion simple 51
2.6.2 Récursion mutuelle 52
2.6.3 Récursion locale en $Lisp_2$ 53
2.6.4 Récursion locale en $Lisp_1$ 54
2.6.5 Créer des liaisons non initialisées 56
2.6.6 Récursion sans affectation 58
2.7 Conclusion 63
2.8 Exercices 64
2.9 Lectures recommandées 66

3 Échappement, reprise : continuations 67
3.1 Formes de manipulation des continuations 70
3.1.1 Le couple `catch` et `throw` 70
3.1.2 Le couple `block` et `return-from` 71
3.1.3 Échappements à durée de vie dynamique 73
3.1.4 Comparaison entre `catch` et `block` 75
3.1.5 Échappement à durée de vie illimitée 76
3.1.6 Protection 79
3.2 Les acteurs d'un calcul 82
3.2.1 Rappel sur les objets 82
3.2.2 L'interprète à continuations 84
3.2.3 Citation 85
3.2.4 Alternative 85
3.2.5 Séquence 86
3.2.6 Environnement des variables 86
3.2.7 Fonctions 87
3.3 Initialisation de l'interprète 89
3.4 Implantation des formes de contrôle 90
3.4.1 Implantation de `call/cc` 90
3.4.2 Implantation de `catch` 91
3.4.3 Implantation de `block` 92
3.4.4 Implantation de `unwind-protect` 93
3.5 Comparaison entre `call/cc` et `catch` 95
3.6 Programmation par continuations 97
3.6.1 Valeurs multiples 97
3.6.2 Récursion terminale 98
3.7 Continuations partielles 100
3.8 Conclusion 101
3.9 Exercices 102
3.10 Lectures recommandées 104

4 Affectation et effets de bord **105**
4.1 Affectation . . . 105
4.1.1 Les boîtes . . . 108
4.1.2 Affectation sur variable libre . . . 110
4.1.3 Affectation sur variable prédéfinie . . . 114
4.2 Effets de bord . . . 115
4.2.1 Égalité . . . 116
4.2.2 Égalité de fonctions . . . 118
4.3 Implantation . . . 120
4.3.1 Alternative . . . 122
4.3.2 Séquence . . . 122
4.3.3 Environnement . . . 123
4.3.4 Référence à une variable . . . 123
4.3.5 Affectation . . . 123
4.3.6 Application fonctionnelle . . . 124
4.3.7 Abstraction . . . 124
4.3.8 Mémoire . . . 125
4.3.9 Représentation des valeurs . . . 126
4.3.10 Comparaison avec la programmation par objets . . . 128
4.3.11 L'environnement initial . . . 128
4.3.12 Paires pointées . . . 130
4.3.13 Comparaison . . . 130
4.3.14 Lancer l'interprète . . . 131
4.4 Entrées/sorties et mémoire . . . 132
4.5 Sémantique des citations . . . 133
4.6 Conclusions . . . 137
4.7 Exercices . . . 137

5 Sémantique dénotationnelle **139**
5.1 Rappels sur le λ-calcul . . . 141
5.2 Sémantique de Scheme . . . 143
5.2.1 Référence à une variable . . . 145
5.2.2 Séquence . . . 146
5.2.3 Alternative . . . 147
5.2.4 Affectation . . . 149
5.2.5 Abstraction . . . 149
5.2.6 Application fonctionnelle . . . 150
5.2.7 `call/cc` . . . 150
5.2.8 Conclusions partielles . . . 151
5.3 Sémantique du λ-calcul . . . 151
5.4 Fonctions d'arité variable . . . 153
5.5 Ordre d'évaluation des applications . . . 156
5.6 Liaison dynamique . . . 159
5.7 Environnement global . . . 162
5.7.1 L'environnement global en Scheme . . . 162
5.7.2 Environnement automatiquement extensible . . . 165

5.7.3 Environnement hyperstatique . . . 165
5.8 Les dessous de ce chapitre . . . 166
5.9 λ-calcul et Scheme . . . 168
5.9.1 Passage de continuations . . . 169
5.9.2 Environnement dynamique . . . 172
5.10 Conclusions . . . 173
5.11 Exercices . . . 173
5.12 Lectures recommandées . . . 174

6 Interprétation rapide **175**
6.1 Interprète rapide . . . 175
6.1.1 Migration des dénotations . . . 176
6.1.2 Bloc d'activation . . . 176
6.1.3 L'interprète : le début . . . 179
6.1.4 Classification des variables . . . 183
6.1.5 Lancement de l'interprète . . . 187
6.1.6 Fonction d'arité variable . . . 187
6.1.7 Formes réductibles . . . 188
6.1.8 Intégration des primitives . . . 190
6.1.9 Variantes sur environnements . . . 193
6.1.10 Conclusions sur l'interprète à calculs migrés . . . 196
6.2 Rejet de l'environnement . . . 197
6.2.1 Conclusion sur l'interprète à environnement mis en registre 201
6.3 Dilution des continuations . . . 201
6.3.1 Fermetures . . . 202
6.3.2 Le prétraiteur . . . 202
6.3.3 Citation . . . 203
6.3.4 Référence . . . 203
6.3.5 Alternative . . . 204
6.3.6 Affectation . . . 204
6.3.7 Séquence . . . 204
6.3.8 Abstraction . . . 205
6.3.9 Application . . . 206
6.3.10 Formes réductibles . . . 207
6.3.11 Appel aux primitives . . . 208
6.3.12 Lancement de l'interprète . . . 208
6.3.13 La fonction `call/cc` . . . 209
6.3.14 La fonction `apply` . . . 209
6.3.15 Conclusions sur l'interprète sans continuation . . . 210
6.4 Conclusions . . . 211
6.5 Exercices . . . 211
6.6 Lectures recommandées . . . 212

7 Compilation **213**
7.1 Compilation vers des octets . . . 215
7.1.1 Introduction du registre `*val*` . . . 215
7.1.2 Invention de la pile . . . 216
7.1.3 Individualisation des instructions . . . 218
7.1.4 Protocole d'appel fonctionnel . . . 220
7.2 Langage et machine cible . . . 221
7.2.1 Désassemblage . . . 224
7.3 Codage des instructions . . . 225
7.4 Instructions . . . 227
7.4.1 Variables locales . . . 228
7.4.2 Variables globales . . . 229
7.4.3 Sauts . . . 230
7.4.4 Invocations . . . 231
7.4.5 Divers . . . 232
7.4.6 Lancement du compilateur-interprète . . . 233
7.4.7 Point d'orgue . . . 235
7.5 Continuations . . . 235
7.6 Échappements . . . 237
7.7 Variables dynamiques . . . 240
7.8 Exceptions . . . 243
7.9 Compilation séparée . . . 248
7.9.1 Compiler un fichier . . . 248
7.9.2 Fabriquer une application . . . 250
7.9.3 Exécuter une application . . . 253
7.10 Conclusions . . . 254
7.11 Exercices . . . 255
7.12 Lectures recommandées . . . 256

8 Évaluation et réflexivité **257**
8.1 Programmes et valeurs . . . 257
8.2 `eval` comme forme spéciale . . . 263
8.2.1 Création de variables globales . . . 265
8.3 `eval` comme fonction . . . 265
8.4 Le coût d'`eval` . . . 267
8.5 `eval` interprété . . . 268
8.5.1 Interchangeabilité des représentations . . . 268
8.5.2 Environnement global . . . 269
8.6 Réification d'environnements . . . 272
8.6.1 La forme spéciale `export` . . . 272
8.6.2 La fonction `eval/b` . . . 274
8.6.3 Enrichissement d'environnements . . . 275
8.6.4 Réification d'environnement clos . . . 278
8.6.5 La forme spéciale `import` . . . 281
8.6.6 Accès simplifié aux environnements . . . 286
8.7 Interprète réflexif . . . 287

8.7.1 La forme `define` . . . 292
8.8 Conclusions . . . 293
8.9 Exercices . . . 293
8.10 Lectures recommandées . . . 294

9 Macrologie **295**
9.1 La préparation . . . 296
9.1.1 Mondes multiples . . . 297
9.1.2 Monde central . . . 297
9.2 La macroexpansion . . . 298
9.2.1 Mode exogène . . . 298
9.2.2 Mode endogène . . . 300
9.3 Appels aux macros . . . 301
9.4 Expanseurs . . . 302
9.5 Acceptabilité du macroexpansé . . . 304
9.6 Définition des macros . . . 306
9.6.1 Mondes multiples . . . 306
9.6.2 Monde unique . . . 309
9.6.3 Évaluation simultanée . . . 315
9.6.4 Redéfinition de macros . . . 315
9.6.5 Comparaisons . . . 316
9.7 Portée des macros . . . 317
9.8 Evaluation et expansion . . . 320
9.9 Emploi des macros . . . 322
9.9.1 Autres caractéristiques . . . 323
9.9.2 Arpenteur de code . . . 324
9.10 Captures inopinées . . . 325
9.11 Un système de macros . . . 327
9.11.1 Objectification . . . 328
9.11.2 Les formes spéciales . . . 333
9.11.3 Les niveaux d'évaluation . . . 334
9.11.4 Les macros . . . 335
9.11.5 Limites . . . 338
9.12 Conclusions . . . 339
9.13 Exercices . . . 339
9.14 Lectures recommandées . . . 340

10 Compilation vers C **341**
10.1 Objectification . . . 343
10.2 Arpentage . . . 343
10.3 Introduction de boîtes . . . 344
10.4 Élimination des fonctions imbriquées . . . 345
10.5 Collecte des citations et des fonctions . . . 350
10.6 Collecte de variables temporaires . . . 352
10.6.1 Point d'orgue . . . 353
10.7 Génération de C . . . 354

10.7.1 Environnement global 355
10.7.2 Citations 357
10.7.3 Déclaration des données 360
10.7.4 Compilation des expressions 361
10.7.5 Compilation des applications fonctionnelles 364
10.7.6 Environnement prédéfini 366
10.7.7 Compilation des fonctions 366
10.7.8 Initialisation du programme 368
10.8 Représentation des données 371
10.8.1 Déclaration des valeurs 374
10.8.2 Variables globales 375
10.8.3 Définition des fonctions 376
10.9 Bibliothèque d'exécution 377
10.9.1 Allocation 377
10.9.2 Fonctions sur les paires 378
10.9.3 Invocation 379
10.10 Avoir `call/cc` ou pas ? 382
10.10.1 La fonction `call/ep` 383
10.10.2 La fonction `call/cc` 384
10.11 Interface avec C 392
10.12 Conclusions 393
10.13 Exercices 393
10.14 Lectures recommandées 393

11 Essence d'un système à objets **395**
11.1 Fondations 397
11.2 Représentation des objets 398
11.3 Définition de classes 400
11.4 Représentation des classes 404
11.5 Les fonctions d'accompagnement 406
11.5.1 Prédicat 407
11.5.2 Allocateur sans initialisation 408
11.5.3 Allocateur avec initialisation 411
11.5.4 Accès aux champs 413
11.5.5 Lecteur de champ 413
11.5.6 Écrivain de champ 415
11.5.7 Longueur de champ 416
11.6 Création des classes 416
11.7 Fonctions d'accompagnement prédéfinies 418
11.8 Fonctions génériques 418
11.9 Méthodes 423
11.10 Conclusions 425
11.11 Exercices 425
11.12 Lectures recommandées 426

Solutions des exercices **427**

Bibliographie **454**

Index **473**

Avis au lecteur

Quoique la littérature sur Lisp soit abondante et déjà largement accessible au public français, un ouvrage pourtant manquait que vous tenez entre les mains ! Le substrat mathématico-logique sur lequel est fondé Lisp/Scheme exige que ses utilisateurs modernes ne répugnent point à lire des programmes usant, voire abusant, de haute technologie, c'est-à-dire de fonctionnelles d'ordre très largement supérieur et/ou de continuations. Les concepts de demain sont élaborés sur ces bases ; les méconnaître rendrait l'avenir plus inquiétant [1]. Expliquer ces entités, leur genèse, leurs variantes est un point que ce livre tente de couvrir en moult détails. Le folklore nous enseigne que si le lispien moyen connaît la valeur de chacune des constructions qu'il emploie, il en ignore généralement le coût. Cet ouvrage comble aussi cette lacune par une étude approfondie de la sémantique et de l'implantation canonique des différents traits de Lisp tels que sédimentés par plus de trente ans d'histoire.

Lisp est un langage plaisant en lequel de nombreux problèmes fondamentaux, non triviaux, peuvent être simplement étudiés. Lisp est, avec ML qui privilégie le typage et l'absence d'effets de bord, le plus représentatif des langages applicatifs. Les concepts qu'illustre cette classe de langages doivent absolument être maîtrisés par les étudiants et les informaticiens d'aujourd'hui ou de demain [2]. Fondés sur la notion de fonction que des siècles de mathématique ont lentement mûrie, les langages applicatifs sont omniprésents en informatique sous des formes variées (*shell* et flots d'UN⋆X, langage d'extension d'édition de texte en Emacs ou de dessin assisté sous AutoCAD ...). Ne pas reconnaître ces modèles, c'est méconnaître la composabilité de leurs éléments primitifs et donc se limiter à un mot à mot pénible pour l'écriture de programmes sans architecture.

Public

Cet ouvrage s'adresse :

---

1. La première édition de ce livre date de 1994 et les adjonctions à cette préface apparaissent dans ces notes de bas de page. Depuis 1994, Perl, Python, C# se sont mis à proposer des fermetures, les classes internes de Java sont également des sortes de fermetures. Le terme de continuation est apparu dans le contexte des systèmes d'exploitation [DBRD91]. Le moteur Rhino implantant JavaScript procure également des continuations.

2. Les modèles d'exécution des langages dynamiques tels que JavaScript, Perl, Python, Ruby, Scheme mais aussi Java sont macroscopiquement assez proches.

— aux étudiants de master qui ont à étudier l'implantation de langages (applicatifs ou pas) par interprétation et/ou compilation ;
— à tous les programmeurs (et computistes) en Lisp ou Scheme qui pourront comprendre finement le coût ou la subtilité des constructions qu'ils emploient afin de s'améliorer dans leur art et produire des programmes plus efficaces et plus portables ;
— aux nombreux amoureux des langages de programmation qui y trouveront de multiples pistes de réflexion sur leur langage favori.

Philosophie

Le présent ouvrage s'inspire de cours professés en DEA ITCP, en DESS GLA de l'université Pierre et Marie Curie (Paris 6) ainsi qu'en enseignement d'approfondissement à l'École Polytechnique. Il prend typiquement la suite d'un cours d'initiation à un langage applicatif (Lisp, Scheme, ML ...), qui culmine généralement par une description du langage en lui-même. L'ouvrage a pour ambition de très largement couvrir la sémantique et l'implantation d'interprètes ou de compilateurs pour des langages applicatifs. Il présente en effet pas moins de douze interprètes et deux compilateurs (vers du code-octet et vers le langage C) sans oublier un système d'objets (dérivé du populaire MEROON). Contrairement à de nombreux livres, cet ouvrage ne néglige aucunement des aspects aussi importants que la réflexivité, l'introspection, l'évaluation dynamique sans oublier les macros, tous phénomènes essentiels de la famille des dialectes de Lisp.

La facture de ce livre s'inspire principalement de deux ouvrages : *Anatomy of Lisp* [All78] qui couvrait l'implantation de Lisp des années soixante-dix et *Operating System Design : The XINU Approach* [Com84] dont les programmes ne celaient aucun détail et, par là même, s'acquéraient la totale confiance des lecteurs. Nous avons donc souhaité réaliser un ouvrage précis dont le thème central est la sémantique des langages applicatifs et de Scheme en particulier. Au travers de multiples implantations explorant différents aspects, la construction d'un tel système est entièrement explicitée. La plupart des grandes variations schismatiques qu'ont pu connaître les langages applicatifs est analysée, démontée, implantée et comparée. Tous les détails, même les plus infimes, sont divulgués. Cela permet au lecteur de ne point se bloquer par manque d'informations, le conforte dans l'idée que l'on ne lui cache rien, le met, enfin, à pied d'œuvre pour expérimenter par lui-même les concepts manipulés puisque tous les programmes de ce livre peuvent être récupérés électroniquement.

Structure

L'ouvrage est formé de deux parties. La première est une progression partant de l'implantation d'un interprète naïf pour Lisp et débouchant sur la sémantique de Scheme. Le fil conducteur est le besoin de préciser, d'affiner et de définir successivement les espaces de noms (Lisp_1, Lisp_2 ...), les continuations (et les multiples formes de contrôle associées), les affectations et les écritures dans les structures de données.

Chapitre	Signature
1	`(eval exp env)`
2	`(eval exp env fenv)`
	`(eval exp env fenv denv)`
	`(eval exp env denv)`
3	`(eval exp env cont)`
4	`(eval e r s k)`
5	`((meaning e) r s k)`
6	`((meaning e sr) r k)`
	`((meaning e sr tail?) k)`
	`((meaning e sr tail?))`
7	`(run (meaning e sr tail?))`
10	`(->C (meaning e sr))`

FIGURE 1 – Signatures (approximatives) des interprètes et compilateurs

Cette lente augmentation du langage défini s'accompagne d'une régression du langage de définition qui s'appauvrit jusqu'à ne plus être qu'une sorte de λ-calcul. La description obtenue à ce stade est alors convertie en son équivalent dénotationnel par une simple curryfication. La nécessité de ce parcours quasi initiatique de recherche de précision dans le langage décrit nous semble, après plus de six ans d'enseignement, une très bonne approche de la sémantique dénotationnelle qui surgit ainsi justifiée et non parachutée.

La seconde partie est un voyage inverse. Partant de la sémantique dénotationnelle et en quête d'efficacité, nous abordons l'interprétation rapide (par prétraitement des parties statiques), puis ce conditionnement (cette précompilation) est mis en œuvre pour un compilateur de code-octet. Sont alors traitées, dans ce cadre séparant bien la préparation des programmes de leur exécution : l'évaluation dynamique (`eval`), les aspects réflexifs (environnements de première classe, interprète auto-interprétable et tour d'interprètes) et les sémantiquement redoutables macros. Pour sacrifier à la mode, un second compilateur vers le langage C est présenté. Enfin le livre se clôt par l'implantation d'un système d'objets, objets qui sont justement utilisés pour décrire finement l'implantation de certains interprètes et de certains compilateurs.

La répétition est mère de la pédagogie. La multiplicité des interprètes présentés, écrits dans des styles volontairement variés (naïf, par objets, par fermetures, dénotationnel …) couvre l'essentiel des techniques utilisées pour l'implantation des langages applicatifs et permet d'exciter la réflexion des lecteurs quant aux différences qu'ils présentent entre eux. C'est la (re)connaissance de ces différences, esquissées en figure 1, qui fonde la compréhension intime de ce que sont un langage et son implantation : Lisp n'est pas une de ces implantations en particulier, c'est une famille de dialectes qui tous composent leur propre cocktail à partir des traits présentés.

De manière générale, les chapitres forment des entités plutôt autonomes d'environ quarante pages et sont accompagnés d'exercices, dont les corrigés sont donnés en fin d'ouvrage. La bibliographie contient non seulement des références historiques (permettant d'apprécier l'évolution de Lisp depuis 1960) mais aussi des références aux

travaux actuels de recherches.

Coda

Si ce livre se veut divertissant et instructif tout à la fois, il n'est pas nécesssairement simple à lire. Certains sujets ne dévoilent leur beauté que pressés de près et seulement si l'ardeur mise à leur siège est à la mesure de leur complexité. L'étude des langages de programmation est une ascèse qui ne se peut apprendre sans au moins quelques outils théoriques comme le λ-calcul ou la sémantique dénotationnelle. Le dessein de cet ouvrage est d'amener le lecteur à ces connaissances suivant un ordre harmonieux qui ne supprime pas pour autant tout effort de sa part.

Il va de soi que cet ouvrage nécessite quelques connaissances préalables de Lisp ou de Scheme : le lecteur doit connaître les quelque trente fonctions de base qui suffisent pour débuter et être à même de déchiffrer sans trop d'efforts des programmes récursifs. Le langage adopté pour cet ouvrage est Scheme dont un abrégé apparaît plus loin, augmenté d'une couche d'objets, nommée MEROON, qui n'est utilisée que lorsqu'il s'agira d'exhiber les problèmes de représentation et d'implantation. Tous ces programmes ont été testés et tournent sous Scheme. Ils ne devraient pas poser de problèmes de portage aux lecteurs ayant assimilé ce livre !

Il ne me reste plus qu'à remercier les organismes qui m'ont procuré les machines (Apple Mac SE30 puis Sony News 3260 [3]) et les moyens qui m'ont permis de composer cet ouvrage : l'École Polytechnique, l'Institut National de Recherche en Informatique et Automatique (INRIA–Rocquencourt) et le Greco-PRC de Programmation du Centre National de la Recherche Scientifique (CNRS) [4]. Je remercie aussi tous ceux qui ont concouru à l'élaboration de ce livre par tous les moyens dont ils disposaient et qu'ils ont su me permettre d'apprécier. Que soient donc remerciés Sophie Anglade, Josy Baron, Jérôme Chailloux, Marc Feeley, Jean-Marie Geffroy, Christian Jullien, Jean-Jacques Lacrampe, Michel Lemaître, Luc Moreau, Jean-François Perrot, Daniel Ribbens, Bernard Serpette, Manuel Serrano, Basile Starynkevitch, Pierre Weis ainsi que ma muse préférée, Claire N***. Bien entendu les erreurs, qui certainement (et bien malheureusement) subsistent, sont miennes.

Notations

Les extraits de programmes apparaissent dans `cette police de caractères qui fait irrésistiblement penser aux machines à écrire de jadis`. Il est possible que certaines parties soient en italiques pour attirer l'œil sur des variantes concentrées en cet endroit.

La notation « $\rightarrow$ » exprime la relation « a pour valeur » tandis que le signe « $\equiv$ » exprime l'équivalence, c'est-à-dire la relation « a même valeur que ». Quand l'évaluation d'une forme est détaillée, une barre verticale désigne l'environnement dans lequel

3. et, depuis 1994, divers PC et un PowerBook.

4. sans oublier l'UPMC (université Pierre et Marie Curie), son service de formation permanente et le LIP6 (laboratoire d'informatique de Paris 6).

l'expression immédiatement précédente doit être considérée. Toutes ces notations apparaissent dans l'exemple qui suit :

```
(let ((a (+ b 1)))
  (let ((f (lambda () a)))
    (foo (f) a) ) )|                    ; foo a pour valeur la fonction
                   | b→ 3               ; de création de paires c'est-à-dire la valeur
                   | foo≡ cons          ; de la variable globale cons.
≡ (let ((f (lambda () a)))
    (foo (f) a) )|
                 | a→ 4
                 | b→ 3
                 | foo≡ cons
≡ (foo (f) a)|
             | a→ 4
             | b→ 3
             | foo≡ cons
             | f≡ (lambda () a)|
                               | a→ 4
→ (4 . 4)
```

Tous les programmes sont écrits en *broken english*, la langue des mots-clés de Lisp, y compris les messages d'erreur qui ressemblent à ceux que l'on peut aujourd'hui observer.

Nous utiliserons çà et là quelques fonctions non standard en Scheme, comme `gensym` qui permet de créer des symboles garantis nouveaux, c'est-à-dire différents de tout symbole précédemment connu. Nous utilisons aussi, au chapitre 10, `format` et `pp` pour imprimer. Ces fonctions existent dans la plupart des implantations de Lisp ou Scheme.

Certains exemples ne se peuvent comprendre que par rapport à un dialecte particulier comme COMMON LISP, Dylan, EULISP, IS-Lisp, Le-Lisp[5], Scheme, etc. Dans ce cas, le nom du dialecte est apposé en regard de l'exemple comme dans ce qui suit :

IS-Lisp
```
(defdynamic fooncall
  (lambda (one :rest others)
    (funcall one others) ) )
```

Afin de faciliter la traversée de ce livre et lorsque sont évoquées des discussions passées ou à venir, un signe indicateur **[☞ p.]** mentionne la page concernée. C'est aussi le cas de variantes proposées en exercices dont le numéro est evoqué comme suit **[☞ Ex.]**. Enfin un index permet de retrouver toutes les définitions de fonctions citées. **[☞ p. 473]**

Enfin nous avons gardé aux fonctions le genre féminin. Ainsi la fameuse `car` retourne-t-elle le premier élément d'une liste non vide.

Petit précis de Scheme

Ce bref résumé n'a pas l'ambition d'un livre totalement dédié à l'apprentissage du langage Scheme comme le sont excellement et en anglais [AG85, Dyb87, SF89] ou en français [Cha96, Huf96, Blo00, BDP+04]. Il n'expose que les grands traits de

5. Le-Lisp est une marque déposée de l'Institut National de Recherche en Informatique et Automatique (INRIA).

ce dialecte dont nous userons dans toute la suite, suite qui vous conduira à un plus haut niveau de compréhension de ce même dialecte par un auto-amorçage du plus bel effet. Le document de référence est le *Revised revised revised revised revised Report on Scheme* ou R^5RS [KCR98].

Scheme permet de manipuler des symboles, des caractères, des chaînes de caractères, des listes, des nombres, des booléens, des vecteurs, des flux et des fonctions (ou procédures). Tous ces types de données ont leur prédicat associé : `symbol?`, `char?`, `string?`, `pair?`, `number?`, `boolean?`, `vector?` et `procedure?`. Ils ont aussi, quand approprié, des sélecteurs et modificateurs correspondants comme `string-ref`, `string-set!`, `vector-ref`, `vector-set!` et pour les listes : `car`, `cdr`, `set-car!` et `set-cdr!`. Les sélecteurs `car` et `cdr` peuvent être composés, ainsi `cadr` désigne le second terme d'une liste.

Ces valeurs peuvent être implicitement nommées et créées par simple mention : c'est le cas des symboles (ou identificateurs), des caractères (préfixé de `#\` comme dans `#\Z` ou `#\space`), des chaînes (lorsqu'encloses entre guillemets), des listes dûment parenthésées, des nombres, des booléens (`#t` et `#f`) et des vecteurs (avec la syntaxe `#(do re mi)`). Ces valeurs peuvent être dynamiquement construites ou converties entre elles, lorsque cela est possible, grâce notamment à `string->symbol`, `int->char`, `string`, `make-string`, `cons`, `list`, `vector` et `make-vector`.

Les entrées/sorties sont l'objet des fonctions `read` qui lit une expression, `display` qui l'imprime et `newline` qui permet de passer à la ligne.

La séquence (ou forme `begin`) permet de regrouper une suite d'expressions à évaluer séquentiellement, par exemple :

```
(begin (display 1) (display 2) (newline))
```

Plusieurs formes conditionnelles existent dont la plus simple est la forme *if—then—else—* classique notée, en Scheme, (`if` *`condition alors sinon`*). Pour traiter des aiguillages à plus de deux voies, on trouve `cond` et `case`. La forme `cond` contient une suite de clauses débutant par une expression booléenne ; celles-ci sont tour à tour évaluées jusqu'à ce qu'une retourne Vrai (c'est-à-dire non Faux, non `#f`), les formes qui suivent cette expression booléenne ayant réussi sont alors évaluées et leur résultat devient celui de la forme `cond` tout entière. Voici un exemple de forme `cond` dans laquelle figure un comportement par défaut introduit par le mot-clé `else` :

```
(cond ((eq? x 'flip) 'flop)
      ((eq? x 'flop) 'flip)
      (else (list x "neither flip nor flop")) )
```

La forme `case` a comme premier paramètre une forme qui fournit une clé que l'on recherche ensuite dans toutes les clauses qui suivent et dont chacune précise la ou les clés qui la déclenche. Lorsqu'une clé adaptée est trouvée, les formes associées sont évaluées et leur résultat devient celui de la forme `case` tout entière. Voici la forme `cond` de l'exemple précédent mise à la sauce de `case` :

```
(case x
  ((flip) 'flop)
  ((flop) 'flip)
  (else (list x "neither flip nor flop")) )
```

Les fonctions sont définies par la forme `lambda`. Juste après le mot-clé `lambda`, se situent les variables de la fonction suivies d'expressions indiquant comment se calcule celle-ci. Les variables peuvent être modifiées par affectation, notée `set!`. Les

littéraux sont introduits par `quote`. Les formes `let`, `let*` et `letrec` introduisent toutes des variables locales dont la valeur d'initialisation est diversement calculée.

La forme `define` permet de définir des valeurs nommées de tout genre. On usera de la facilité d'écriture qu'offrent les `define` internes ainsi que de la syntaxe non essentielle où le nom de la fonction à définir est stipulé par sa forme d'appel. En voici un exemple :

```
(define (rev l)
  (define nil '())
  (define (reverse l r)
    (if (pair? l) (reverse (cdr l) (cons (car l) r)) r) )
  (reverse l nil) )
```

Cet exemple pourrait aussi être écrit, sans syntaxe inessentielle :

```
(define rev
  (lambda (l)
    (letrec ((nil '())
             (reverse
               (lambda (l r)
                 (if (pair? l)
                     (reverse (cdr l) (cons (car l) r))
                     r ) ) ))
      (reverse l nil) ) ) )
```

Ici s'achèvent ces quelques rappels de Scheme.

Programmes

Les programmes figurant dans ce livre, les divers interprètes et compilateurs, le système d'objets et les tests associés sont disponibles via Internet en :

`https://christian.queinnec.org/Books/LiSP-2ndEdition.tgz`

On peut aussi les trouver en

`https://github.com/appleby/Lisp-In-Small-Pieces`

L'adresse électronique de l'auteur est `Christian@Queinnec.org`.

Lectures recommandées

Comme nous supposons Scheme connu, on se référera aux classiques [AG85, SF89, ML95, HKK99, FFFK03] et, en français, [Cha96, Huf96, Blo00, BDP+04] sans oublier les exercices de [MQRS99, BDP+06]. On pourra s'armer aussi, pour mieux profiter de cet ouvrage, des manuels de référence de COMMON LISP [Ste90], Dylan [App92b], EULISP [PE92], IS-Lisp [ISO94], Le-Lisp, OakLisp [LP88], Scheme [KCR98] et T [RAM84]. Enfin, on consultera [BG94] pour un large tour d'horizon sur les langages de programmation en général.

Post-préface

Cet ouvage fût initialement publié, en 1994, sous le titre « Les Langages Lisp », chez InterÉditions. Une nouvelle édition corrigée est sortie des éditions Paracamplus en 2006 : cette édition est désormais totalement épuisée. À part cette nouvelle post-préface, la publication de l'ouvrage chez Amazon Publishing Company a le même contenu mais permet d'être imprimée à la demande. Que soient remerciés tous les lecteurs ayant fait pression, tout au long de ces dernières années, pour obtenir sa ré-édition !

1

Fondements de l'interprétation

Ce chapitre présente l'interprète de base sur lequel nous fonderons la plus grande part de cet ouvrage. Résolument simple, il s'apparente à Scheme plus qu'à Lisp mais expliquera celui-ci grâce à celui-là. Seront successivement discutés : les articulations de cet interprète, le fameux couple formé d'`eval` et d'`apply`, puis les qualités attendues des environnements et des fonctions. Enfin, diverses variantes seront esquissées procurant autant de pistes que poursuivront les prochains chapitres. Ces variantes sont autant de doutes qui devraient révéler à l'intrépide lecteur les abysses qu'il côtoie. Cet interprète et ses variations sont écrits en Scheme naturel sans restriction linguistique.

La littérature sur Lisp n'a que rarement résisté au plaisir narcissique de donner la définition de Lisp en Lisp. Ce pli, qu'a marqué pour la première fois le manuel de référence de Lisp 1.5 [MAE+62], a été très largement imité ; citons seulement [Rib69], [Gre77], [Que82], [Cay83], [Cha80], [SJ93] et en langue étrangère [Rey72], [Gor75], [SS75], [All78], [McC78b], [Lak80], [Hen80], [BM84], [Cli84], [FW84], [dRS84], [AG85], [R3R86], [Mas86], [Dyb87], [WH88], [Kes88], [LF88], [Dil88], [Kam90], [BDP+04] parmi tant d'autres !

Ces évaluateurs sont très variés à la fois par le langage qu'ils définissent et par celui qu'ils utilisent pour ce faire mais surtout, par le but poursuivi : l'évaluateur proposé dans [Lak80] montre comment sont immergés des objets et concepts graphiques en Lisp ; l'évaluateur de [BM84] tient à jour un paramètre représentant la taille de l'évaluation.

Le *langage de définition* n'est pas indifférent. Que l'affectation et les opérateurs chirurgicaux (`set-car!`, `set-cdr!` etc.) soient autorisés augmente la richesse du langage de définition et permet ainsi de diminuer la taille (en nombre de lignes) des descriptions, voire de simuler précisément le *langage décrit* en termes rappelant les instructions de bas niveau de la machine (éventuellement virtuelle). Inversement, la description utilise plus de concepts. Restreindre le langage de définition complique la tâche mais diminue les risques de divergences sémantiques. Même si la taille de la description augmente, le langage décrit est plus précis et, de ce fait, mieux partagé.

La figure 1.1 classe quelques interprètes représentatifs en fonction de la complexité du langage de définition (en abscisse) et du langage défini (en ordonnée). La progression des connaissances y apparaît clairement : des problèmes de plus en plus

complexes sont attaqués avec des moyens cherchant à se restreindre. Le présent ouvrage correspond au vecteur orienté qui part d'un Lisp très riche implantant Scheme pour arriver au λ-calcul implantant un Lisp très riche.

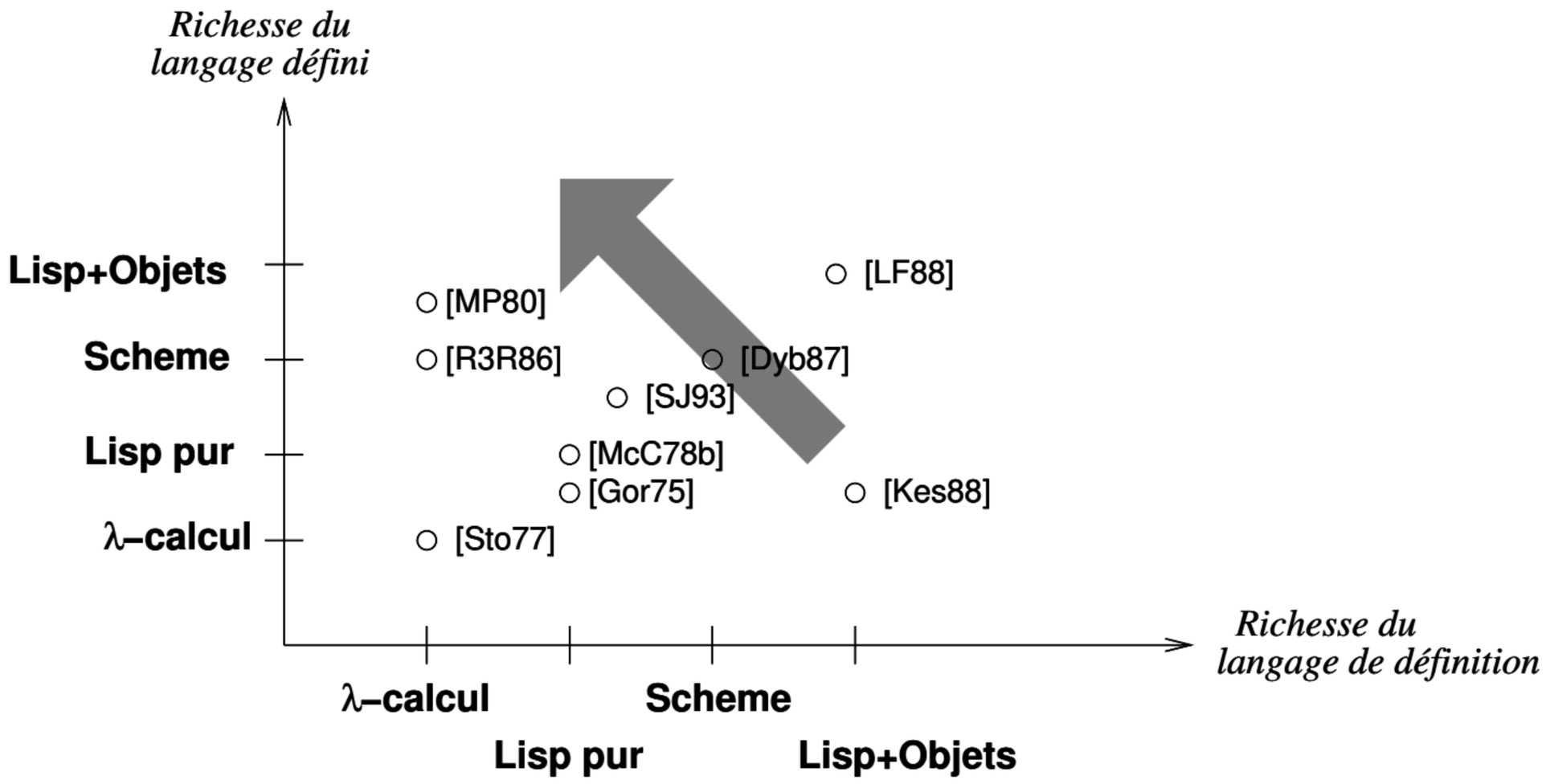


FIGURE 1.1 – Classification de quelques évaluateurs

1.1 Évaluation

L'essentiel d'un interprète pour Lisp se concentre en une unique fonction autour de laquelle tournoient quelques autres plus utilitaires. Cette fonction, nommée `eval`, prend un programme en argument et délivre en sortie sa valeur.

La présence d'un évaluateur explicite est un trait caractéristique de Lisp. Son existence n'est pas due au hasard mais est plutôt le fruit d'un dessein précis. Un langage est universel s'il a la puissance d'une machine de Turing, ce qui, compte tenu de la rudimentarité de celle-ci (une mémoire de cellules binaires, une tête de lecture/écriture mobile), n'est pas difficile à obtenir ; il est probablement plus délicat de concevoir un langage utile qui ne soit pas universel.

La fameuse thèse de Church dit que toute fonction calculable peut s'écrire dans tout langage universel. Un système Lisp peut être comparé à une fonction prenant des programmes en entrée et retournant leur valeur en sortie. L'existence de tels systèmes prouve leur calculabilité : un système Lisp peut donc être écrit en un langage universel. *Ergo* la fonction `eval` peut être écrite en Lisp et, plus généralement, le comportement de Fortran peut être décrit en Fortran, etc.

Ce qui rend Lisp unique est qu'une explicitation non triviale d'`eval` est d'une taille raisonnable, de une à vingt pages environ [1] suivant le niveau de détail souhaité. Cette propriété est le résultat d'un effort de conception important pour régulariser le

1. La définition de ce chapitre mesure environ 150 lignes tandis que celle (pédagogique) de [BDP+04] mesure (commentaires compris) 688 lignes.

langage, supprimer les cas particuliers et surtout établir une syntaxe presque abstraite tant est auguste sa simplicité.

De nombreuses et intéressantes propriétés découlent de la présence d'`eval` et de sa définissabilité en Lisp même.

— Apprendre Lisp peut s'effectuer par la lecture d'un manuel de référence (qui expose les fonctions par thèmes) ou bien par l'étude de la fonction `eval`. Le défaut de cette dernière approche est que pour lire `eval`, il faut connaître Lisp, ce qui était le résultat souhaité de l'opération et non son prérequis. En fait, il suffit de connaître le sous-ensemble de Lisp utilisé par `eval`. D'autre part, le langage que définit `eval` est plus dépouillé en ce sens qu'il ne procure que l'essence du langage réduit aux seules formes spéciales et fonctions primordiales.
Disposer de deux approches interagissantes pour l'apprentissage de Lisp est un indéniable atout que procure Lisp.

— Que la définition d'`eval` soit disponible en Lisp même permet d'immerger simplement l'environnement de programmation au sein du langage et fournit à moindre coût : traceur, metteur au point et autre évaluateur réversible [Lie87]. En effet, l'écriture de ces outils de contrôle d'évaluation n'est qu'une décoration du code d'`eval` pour, par exemple, imprimer les appels fonctionnels, demander à l'utilisateur son indice de satisfaction afin d'éventuellement poursuivre l'évaluation, mémoriser des résultats intermédiaires, etc.
Ces qualités ont pendant longtemps assuré la suprématie de Lisp en matière d'environnement de programmation. Encore aujourd'hui, qu'`eval` soit définissable permet que soient expérimentés facilement de nouveaux modèles de metteur au point.

— Enfin, `eval` est un outil de programmation. Cet outil est controversé puisqu'il implique au sein d'une application écrite en Lisp et usant d'`eval`, de conserver un interprète ou un compilateur entier, mais surtout de renoncer à de nombreuses optimisations. L'emploi d'`eval` n'est donc pas sans conséquence. Il se justifie dans certains cas, notamment lorsque Lisp sert à la définition et la mise en œuvre d'un langage de programmation incrémentielle.
Outre ce coût important, la sémantique d'`eval` n'est pas claire, ce qui a justifié sa mise à l'écart de la définition officielle de Scheme dans le R3RS [R3R86] ; `eval` apparaît néanmoins dans le R5RS [KCR98]. **[☞ p. 257]**

1.2 Évaluateur fondamental

Il est possible de distinguer dans un programme les *variables libres* des *variables liées*. Une variable est *libre* lorsqu'aucune forme liante (`lambda`, `let` ...) ne la qualifie ; dans le cas contraire les variables sont dites *liées*. Comme son nom l'indique, une variable libre est affranchie de toute contrainte ; sa valeur peut être n'importe quoi. Il importe donc, pour connaître la valeur d'un fragment de programme comportant des variables libres, que soient connues les valeurs de ces mêmes variables libres. On nomme *environnement*, la structure de données associant variables et valeurs. La

fonction `evaluate` [2] est donc binaire, prend un programme accompagné d'un environnement et retourne une valeur.

```
(define (evaluate exp env) ...)
```

1.3 Évaluation des atomes

Une importante caractéristique de Lisp est que les programmes sont *représentés* par les expressions mêmes du langage. Mais comme toute représentation suppose un certain *codage*, il est nécessaire d'expliciter comment sont représentés les programmes. Les principales conventions sont qu'une variable est représentée par le symbole éponyme et qu'une application fonctionnelle est représentée par une liste dont le premier terme représente la fontion à appliquer et dont les termes suivants représentent les arguments qui seront soumis à cette fonction.

Comme tout compilateur, `evaluate` débute par une analyse syntaxique de l'expression à évaluer pour en déduire ce qu'elle représente. Le titre de cette section est donc impropre puisqu'il ne s'agit pas littéralement d'évaluer des atomes mais des programmes dont la représentation est atomique. Il importe de bien distinguer le programme de sa représentation, le message du medium. La fonction `evaluate` travaille sur la représentation, en déduit l'intention escomptée, exécute enfin ce qui est demandé.

```
(define (evaluate exp env)
  (if (atom? exp)  ; (atom? exp) ≡ (not (pair? exp))
      ...
      (case (car exp)
        ...
        (else ...) ) ) )
```

Si une expression n'est pas une liste, ce peut être un symbole ou une donnée immédiate, comme un nombre ou une chaîne de caractères, etc. Lorsque l'expression est un symbole, l'expression représente une *variable* et sa valeur est celle que lui attribue l'environnement.

```
(define (evaluate exp env)
  (if (atom? exp)
      (if (symbol? exp) (lookup exp env) exp)
      (case (car exp)
        ...
        (else ...) ) ) )
```

La fonction `lookup`, explicitée plus loin, sait retrouver la valeur d'une variable dans un environnement. Notez que la fonction `lookup` a pour signature :

(`lookup` *variable environnement*) → *valeur*

Il y a donc eu une conversion implicite entre symbole et variable. Une écriture plus pointilleuse aurait été à la place de `(lookup exp env)` :

```
... (lookup (symbol->variable exp) env) ...
```

---

2. Afin d'aider le lecteur à ne pas confondre l'évaluateur que nous décrivons avec la fonction `eval`, le plus souvent unaire, nous nommerons `evaluate` et `invoke` les fonctions `eval` et `apply`. Ces nouveaux noms faciliteront aussi la vie du lecteur qui souhaiterait expérimenter ces programmes.

Cette dernière écriture met bien en relief que le symbole, valeur de `exp`, doit être transmué en une variable et que, loin d'être l'identité, la fonction `symbol->variable` [3] convertit une entité syntaxique (un symbole) en une entité sémantique (une variable). Une variable, en effet, n'est rien d'autre qu'un objet imaginaire auquel le langage et le programmeur attachent un certain sens mais qui, pour des raisons matérielles, n'est manipulée que par le biais d'une représentation. La représentation a été choisie pour sa légèreté : `symbol->variable` est l'identité parce que Lisp possède le concept de symbole parmi ses types de base. D'autres représentations auraient pu être adoptées : une variable pourrait apparaître sous la forme d'une suite de caractères préfixée par un signe dollar. En ce dernier cas, la fonction de conversion `symbol->variable` aurait été moins simpliste.

Si une variable est un concept imaginaire, la fonction `lookup` ne saurait l'accepter en premier argument puisqu'elle ne sait opérer que sur des objets tangibles. Il faut donc encore une fois encoder la variable en une représentation, une clé, permettant à `lookup` de retrouver sa valeur dans l'environnement. Une écriture plus précise serait donc :

```
... (lookup (variable->key (symbol->variable exp)) env) ...
```

Mais la paresse naturelle des lispiens les pousse à adopter comme clé associée à une variable le symbole de même nom. Ainsi donc, `variable->key` n'est qu'un inverse de `symbol->variable`, la composition de ces deux fonctions n'étant que l'identité.

On a coutume, lorsque l'expression est atomique (lorsqu'il ne s'agit pas d'une paire pointée) et qu'elle n'est pas un symbole, de la considérer comme la représentation d'une constante qui est sa propre valeur. Cette idempotence est qualifiée d'*autoquote*. Un objet « autoquoté » n'a pas besoin d'être « quoté » et a pour valeur lui-même, voir [Cha94] pour un exemple. Là encore, ce choix n'est pas évident pour plusieurs raisons. Tous les objets atomiques ne se dénotent pas naturellement en eux-mêmes. La valeur de la chaîne de caractères `"a?b:c"` pourrait être d'appeler le compilateur C sur cette chaîne, puis d'exécuter le programme résultant et de réinjecter les résultats en Lisp. D'autres types d'objets semblent absolument rétifs au concept d'évaluation comme, par exemple, les fonctions. La variable `car` a pour valeur la fonction qui extrait d'une paire son fils gauche (nommons la *car*), fonction dont on sait, par ailleurs, l'intérêt. Mais le programme représenté par la fonction *car* elle-même, quelle valeur a-t-il ? Évaluer une fonction est en général l'indice d'une erreur qu'il vaut mieux détecter au plus tôt. Un autre exemple est la liste vide `()` qui pourrait suggérer, de par sa forme écrite, qu'elle est une application vide, c'est-à-dire une application fonctionnelle sans argument et dont on a même oublié de mentionner la fonction. Cette syntaxe est pas définie par la norme de Scheme et n'a, par conséquent, pas de valeur [4]. Il faut donc analyser soigneusement les expressions et ne rendre *autoquote* que les seules données qui le méritent, à savoir nombres, caractères et chaînes de caractères. **[☞ p. 7]** On pourrait ainsi écrire :

---

3. Je n'aime personnellement pas les noms formés comme $x->y$ pour indiquer une conversion car cet ordre complique la compréhension des compositions : $(y->z\,(x->y\,...))$ est moins simple que $(z<-y\,(y<-x\,...))$. En revanche, $x->y$ se lit plus simplement que $y<-x$. On voit là une des difficultés auxquelles se heurtent les concepteurs de langages.

4. Ce qui n'empêche pas certaines implantations de lui donner une valeur !

```
(define (evaluate e env)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e) (string? e) (char? e)
                 (boolean? e) (vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
      ... ) )
```

On voit dans le fragment précédent apparaître le premier cas d'erreur possible. La plupart des systèmes Lisp ont un système d'exception personnel ; aussi est-il délicat d'écrire du code portable en ce domaine. Dans une situation erronée, nous ferons appel à `wrong`[5] avec, pour premier argument, une chaîne de caractères décrivant succinctement (et en *basic english*) la nature de l'erreur et, en arguments suivants, les objets explicitant les raisons de ce dysfonctionnement. Mentionnons toutefois que si les systèmes les plus rudimentaires se contentent d'émettre un message sybillin tel que `Bus error: core dumped` lorsqu'une erreur survient, d'autres abandonnent le calcul en cours et reviennent d'eux-mêmes dans la boucle fondamentale d'interaction. D'autres encore permettent d'associer à un calcul un gestionnaire d'exceptions qui se saisira de l'objet représentant l'erreur ou l'exception et qui décidera de la conduite à tenir. **[☞ p. 243]** Les machines-Lisp proposaient même des gestionnaires d'exceptions qui sont de véritables systèmes experts, analysant l'erreur et le code correspondant pour proposer les choix de correction les plus appropriés à l'utilisateur. Toute une gamme !

1.4 Évaluation des formes

Tout langage possède un certain nombre de formes syntaxiques de base intangibles, irredéfinissables et impervertissables. Ce sont, en Lisp, les *formes spéciales*. Elles sont représentées par des listes dont le premier terme est un symbole particulier appartenant à la liste des *opérateurs spéciaux*. Un dialecte de Lisp se caractérise par le jeu de formes spéciales qu'il arbore et sa bibliothèque de fonctions primitives, celles qui ne peuvent être écrites dans le langage même et qui ont des répercussions sémantiques profondes, comme, par exemple, `call/cc` en Scheme. Lisp n'est, d'une certaine manière, qu'un banal λ-calcul appliqué augmenté d'un jeu de formes spéciales. Mais le génie d'un Lisp particulier s'exprime par ce jeu. Scheme a choisi de minimiser le nombre d'opérateurs spéciaux (`quote`, `if`, `set!` et `lambda`) tandis que COMMON LISP (CLtL2 [Ste90]) en procure plus d'une trentaine circonscrivant les nombreux cas où de performantes générations de code véloce sont possibles.

Du fait du codage des formes spéciales, leur analyse syntaxique est simple et se fonde sur une analyse du premier terme des formes : un unique `case` suffit à la tâche. Lorsqu'une forme ne débute pas par un mot-clé, c'est par défaut une *application*[6] *fonctionnelle*. Pour l'instant nous ne retiendrons qu'un petit sous-ensemble de formes

---

5. Notez que nous n'avons pas écrit « la fonction `wrong` ». Les techniques de récupération d'erreurs seront vues page 243.

6. On peut distinguer l'*application fonctionnelle* où une fonction est appliquée à ses arguments d'une *application* qui est un programme autonome susceptible d'être lancé depuis un système d'exploitation. Les deux sens étant clairement différents, nous commettrons l'abus de parler seulement d'application.

spéciales générales (les chapitres suivants en introduiront de nouvelles plus spécialisées) à savoir : `quote`, `if`, `begin`, `set!` et `lambda`.

```
(define (evaluate e env)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e) (string? e) (char? e)
                 (boolean? e) (vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote)  (cadr e))
        ((if)     (if (evaluate (cadr e) env)
                      (evaluate (caddr e) env)
                      (evaluate (cadddr e) env) ))
        ((begin)  (eprogn (cdr e) env))
        ((set!)   (update! (cadr e) env (evaluate (caddr e) env)))
        ((lambda) (make-function (cadr e) (cddr e) env))
        (else     (invoke (evaluate (car e) env)
                          (evlis (cdr e) env) )) ) ) )
```

Pour alléger la définition précédente, l'analyse syntaxique a été réduite à son minimum et ne s'embarrasse pas de vérifier que les `if` sont vraiment ternaires[7], que les citations sont bien écrites, etc. Nous supposerons que les programmes analysés sont corrects.

1.4.1 Citation

La *citation*, ou forme spéciale `quote`, permet d'introduire une valeur qui, sans citation explicite, aurait pu être confondue avec une expression légale : un fragment de programme. La décision de représentation d'un programme sous la forme d'une expression du langage rend nécessaire, lorsque l'on souhaite citer une expression du langage, de la discriminer des programmes qui usurpent le même espace. D'autres choix syntaxiques auraient pu être faits qui auraient évité ce problème. Les M-expressions originellement prévues pour être la syntaxe normale des programmes écrits en Lisp [McC60] éliminaient ces problèmes mais interdisaient les macros qui se sont révélées un fabuleux outil d'extension de syntaxe : les M-expressions sont mortes peu après [McC78a]. La forme spéciale `quote` est donc le discriminateur entre programme et donnée.

La citation consiste à retourner en valeur le terme qui suit le mot-clé, ce qu'énonce fort brièvement le fragment :

```
... (case (car e)
      ((quote) (cadr e)) ...) ...
```

Y a-t-il une différence entre une citation implicite ou explicite, par exemple entre `33` et `'33` ou encore entre[8] `#(fa do sol)` et `'#(fa do sol)` ? La première comparaison porte sur des objets immédiats alors que la seconde porte sur des objets composites (quoique atomiques dans la terminologie lispienne). Il est possible d'imaginer des

---

7. Les formes `if` ne sont pas nécessairement ternaires. Scheme et COMMON LISP acceptent les `if` binaires ou ternaires, EULISP et IS-Lisp uniquement les ternaires, Le-Lisp les `if` au moins binaires avec un `progn` (ou `begin` en Scheme) implicite sur l'alternant.

8. Rappelons qu'en Scheme la notation `#(...)` représente la citation d'un vecteur.

sens divergents pour ces deux fragments. La citation explicite se contente de retourner comme valeur sa citation alors que `#(fa do sol)` pourrait, à chaque évaluation, retourner un nouvel exemplaire d'un vecteur à trois composantes initialisés par trois symboles particuliers. En d'autres mots, `#(fa do sol)` ne serait qu'une abréviation de `(vector 'fa 'do 'sol)` dont le comportement est fortement différent de `'#(fa do sol)` ainsi, du reste, que de `(vector fa do sol)`. Nous reviendrons plus tard sur le sens à accorder à la citation qui est loin d'être simple. **[☞ p. 133]**

1.4.2 Alternative

L'*alternative*, ou forme spéciale `if`, est considérée ici comme ternaire. C'est une structure de contrôle qui, après avoir évalué son premier argument (la *condition*), choisit, suivant la valeur obtenue, de retourner la valeur de son second (la *conséquence*) ou troisième (l'*alternant*) argument. C'est ce qu'exprime le fragment ci-dessous :

```
... (case (car e) ...
     ((if) (if (evaluate (cadr e) env)
               (evaluate (caddr e) env)
               (evaluate (cadddr e) env) )) ... ) ...
```

Cette programmation ne rend pas entièrement justice à la représentation des booléens. Comme le lecteur l'a certainement remarqué, nous mélangeons ici deux langages. Le premier est Scheme (ou du moins y ressemble singulièrement au point que l'on ne peut encore les distinguer) tandis que le second est aussi Scheme (ou du moins s'en approche). Le premier sert à implanter le second ; il y a donc entre eux le même rapport qu'il y a entre, par exemple, TEX [Knu84] et le langage de la première implantation de TEX : Pascal. La discussion sur le sens que revêt l'alternative peut aussi bien s'entendre comme définissant l'alternative du langage de définition que celle du langage en cours de définition.

La fonction `evaluate` retourne une valeur appartenant au langage en cours de définition. Cette valeur n'entretient a priori aucun rapport avec les booléens du langage de définition. Sachant que l'on reprend la convention que tout objet différent du booléen *faux* peut être considéré comme le booléen *vrai*, une programmation plus soigneuse serait donc :

```
... (case (car e) ...
     ((if) (if (not (eq? (evaluate (cadr e) env) the-false-value))
               (evaluate (caddr e) env)
               (evaluate (cadddr e) env) )) ... ) ...
```

On suppose bien sûr que la variable `the-false-value` a pour valeur la représentation (dans le langage de définition) du booléen *faux* du langage en cours de définition. Un large choix est ouvert pour représenter cette valeur et par exemple :

```
(define the-false-value (cons "false" "boolean"))
```

La comparaison d'une valeur quelconque au booléen *faux* étant effectuée par le comparateur physique `eq?`, une paire pointée fait donc très bien l'affaire et ne peut être confondue avec aucune autre valeur possible du langage en cours de définition.

La discussion n'est pas oiseuse. Le *sexe* du booléen *faux*, de la liste vide `()` et du symbole `NIL` défraie encore la chronique du monde lispien. La position la plus propre, indépendamment de toute volonté de préserver l'existant, est que *faux* est différent de `()`, qui n'est que la liste vide, et que tous deux n'ont rien à voir avec le symbole qui

s'épelle `N`, `I` et `L`. Cette position est celle de Scheme qui l'a adoptée quelques semaines avant d'être normalisé à l'IEEE [IEE91]. Là où cependant les choses se gâtent, c'est que `()` se prononce *nil* ! En Lisp traditionnel *faux*, `()` et `NIL` sont un seul et même symbole. En Le-Lisp, `NIL` est une variable dont la valeur est `()`, la liste vide étant assimilée au booléen *faux* et au symbole vide `||`.

1.4.3 Séquence

La *séquence* permet de condenser en une seule entité syntaxique, une suite de formes à évaluer séquentiellement. Similaire au fameux `begin...end` des langages de la famille d'Algol, Scheme préfixe cette forme spéciale par `begin` là où les autres Lisps emploient plutôt `progn`, généralisation immédiate de `prog1`, `prog2`, ... L'évaluation en séquence d'une suite de formes sera sous-traitée à une fonction particulière : `eprogn`.

```
... (case (car e) ...
     ((begin) (eprogn (cdr e) env)) ... ) ...
```

```
(define (eprogn exps env)
  (if (pair? exps)
      (if (pair? (cdr exps))
          (begin (evaluate (car exps) env)
                 (eprogn (cdr exps) env) )
          (evaluate (car exps) env) )
      '() ) )
```

Le sens de la séquence est maintenant biblique. On remarquera toutefois, au sein de la définition de `eprogn`, l'appel récursif terminal à `evaluate` pour le traitement du terme final de la séquence. Le calcul du terme final d'une séquence s'effectue comme si, à lui seul, il remplaçait la séquence toute entière. La récursion terminale sera mieux caractérisée ultérieurement. **[☞ p. 98]**

Une autre remarque concerne le sens qu'il faut attribuer à `(begin)`. Nous avons ici défini que `(begin)` retournait la liste vide `()`. Mais pourquoi diantre la liste vide, pourquoi pas n'importe quoi d'autre : `libellule` ou même `(papillon)` ? Ce choix provient de l'héritage de Lisp où l'usage de rendre `nil` lorsque rien ne semble s'imposer de meilleur prévalait. Mais dans un monde où `nil`, `()` et le booléen *faux* sont distincts, que rendre ? Nous singulariserons le langage décrit dans ce chapitre pour que `(begin)` retourne la valeur de `empty-begin`, soit le nombre (presque arbitraire) `813`[Leb05] :

```
(define (eprogn exps env)
  (if (pair? exps)
      (if (pair? (cdr exps))
          (begin (evaluate (car exps) env)
                 (eprogn (cdr exps) env) )
          (evaluate (car exps) env) )
      empty-begin ) )
(define empty-begin 813)
```

Le problème naît de ce que l'implantation que nous définissons doit nécessairement retourner une valeur. Le langage peut, comme Scheme, n'attribuer aucun sens particulier à `(begin)`, ce qui peut s'interpréter de deux façons au moins : ou bien cette écriture est permise au sein d'une implantation particulière : c'est une extension qui doit donc retourner une valeur librement élue par l'implantation en question ; ou bien

l'écriture `(begin)` est illicite et donc erronée. Par voie de conséquence, il vaut mieux s'abstenir d'utiliser cette forme pour laquelle aucune garantie de légalité et donc de valeur n'existe. Quelques implantations ont un objet `#<unspecified>` qui peut servir à cet usage ainsi que, plus généralement, dans tous les cas où l'on ne sait ce que l'on peut retourner car rien ne semble approprié. Cet objet est souvent imprimable et ne doit point être confondu avec l'indéfini. **[☞ p. 56]**

La séquence n'a pas d'intérêt si le langage est purement fonctionnel, c'est-à-dire sans aucun effet de bord. À quoi sert d'évaluer un programme dont on n'a cure du résultat ? C'est cependant méconnaître les jeux vidéo où la vitesse de la machine excède les réflexes des joueurs. Pour ralentir la machine, on programme des boucles d'attente obtenant ainsi de magnifiques effets de bord sur le temps. Voilà donc à quoi sert `progn` dans le domaine des jeux vidéo programmés en langage fonctionnel pur. En présence d'opérations de lecture ou d'écriture classiques, qui sont des effets de bord sur les flux d'entrée et de sortie, la séquence trouve tout son intérêt car il est nettement plus clair de poser une question (par `display`) puis d'attendre la réponse (par `read`) que l'inverse. La séquence est alors la forme explicite qui permet de séquentialiser une suite d'évaluations. D'autres formes spéciales peuvent aussi introduire un certain ordre comme l'alternative :

`(if` α β β`)` $\equiv$ `(begin` α β`)`

Mais cette règle peut aussi être simulée par [9] :

`(begin` α β`)` $\equiv$ `((lambda (`*void*`)` β`)` α`)`

Cette dernière règle montre, s'il en était encore besoin, que `begin` n'est pas une forme spéciale vraiment nécessaire à Scheme puisqu'elle est émulable par l'appplication fonctionnelle qui impose que les arguments soient calculés avant le corps de la fonction invoquée : c'est la technique dite d'*appel par valeur*.

1.4.4 Affectation

Comme dans de nombreux autres langages, la valeur d'une variable peut être modifiée ; on dit alors qu'on affecte la variable. Comme il s'agit de modifier dans l'environnement la valeur de la variable, nous laisserons ce soin à la fonction `update!` [10] qui sera explicitée plus loin.

```
... (case (car e)
      ((set!) (update! (cadr e) env (evaluate (caddr e) env))) ...
```

L'*affectation* s'effectue en deux temps : la nouvelle valeur est calculée puis devient la nouvelle valeur de la variable. Notez que la variable n'est pas le résultat d'un calcul. De nombreuses variantes sémantiques existent pour l'affectation que l'on discutera aussi plus tard. **[☞ p. 105]** Sachez toutefois que la valeur d'une affectation n'est pas spécifiée en Scheme.

---

9. La variable *void* ne doit pas être libre dans β, ce qui est trivialement vérifiée si *void* n'apparaît pas du tout dans β. C'est pourquoi l'on utilise souvent des `gensym` pour créer de nouvelles variables garanties inédites, voir l'exercice 1.11.

10. Conformément à l'usage courant en Scheme, les fonctions qui font des effets de bord ont leur nom suffixé par un point d'exclamation.

1.4.5 Abstraction

Les fonctions (pour *procedure* dans le jargon Scheme) sont le résultat de l'évaluation de la forme spéciale `lambda` qui elle-même se nomme, par référence au λ-calcul, une *abstraction*. Nous déléguerons à `make-function` le soin de véritablement créer la fonction en lui fournissant tous les paramètres disponibles dont elle, `make-function`, pourrait avoir besoin et notamment la liste des variables, le corps de la fonction et l'environnement courant :

```
... (case (car e) ...
        ((lambda) (make-function (cadr e) (cddr e) env)) ...) ...
```

1.4.6 Application fonctionnelle

Lorsqu'une liste ne possède pas un opérateur spécial en premier terme, c'est alors une *application fonctionnelle* ou, par référence au λ-calcul, une *combinaison*. La fonction, obtenue par évaluation du premier terme, est appliquée à ses arguments obtenus par évaluation des termes suivants. C'est exactement ce que reflète :

```
... (case (car e) ...
        (else (invoke (evaluate (car e) env)
                      (evlis (cdr e) env) )) ) ...
```

La fonction utilitaire `evlis` retourne d'une liste d'expressions la liste des valeurs de ces mêmes expressions. Elle se définit simplement comme :

```
(define (evlis exps env)
  (if (pair? exps)
      (cons (evaluate (car exps) env)
            (evlis (cdr exps) env) )
      '() ) )
```

La fonction `invoke` a la charge d'appliquer son premier argument (une fonction, sauf anomalie) à son second argument (la liste des arguments de la fonction de premier argument) et de retourner en résultat la valeur de cette application. Afin d'éclaircir les multiples usages du mot « argument » dans la phrase précédente, observez que `invoke` est semblable à la classique `apply`, hors l'irruption explicite de l'environnement. Nous remettons à plus tard la représentation exacte des fonctions et environnements.

Retour sur `evaluate`

La description précédente est assez précise. Quelques fonctions utilitaires n'ont pas encore été explicitées comme `lookup` et `update!`, qui traitent des environnements, et, similairement, `make-function` et `invoke`, qui traitent des fonctions. Pourtant, de nombreuses questions trouvent déjà réponse. Par exemple, il est évident que le dialecte défini a un seul espace de noms, est mono-valeur (ou Lisp_1) **[☞ p. 29]** et que les objets fonctionnels y sont présents. En revanche, quel est l'ordre d'évaluation des arguments ?

L'ordre d'évaluation des arguments du Lisp défini est semblable à l'ordre d'évaluation des arguments du `cons` qui apparaît dans `evlis`. On pourrait cependant imposer l'ordre que l'on souhaite, par exemple gauche-droite, par une séquence explicite comme dans :

```
(define (evlis exps env)
```

```
(if (pair? exps)
    (let ((argument1 (evaluate (car exps) env)))
      (cons argument1 (evlis (cdr exps) env)) )
    '() ) )
```

Sans augmenter l'arsenal de traits utilisés dans le Lisp de définition [11], nous avons augmenté la précision de la description du Lisp défini. La première partie de cet ouvrage tendra à définir de plus en plus précisément un certain dialecte tout en restreignant de plus en plus le dialecte servant à la définition.

1.5 Représentation de l'environnement

L'environnement associe des valeurs à des variables. La structure de donnée traditionnelle de Lisp pour représenter de telles associations est la bien nommée *liste d'associations* ou *A-liste*. Nous allons donc représenter l'environnement comme une A-liste associant variables et valeurs. Pour simplifier, nous représenterons les variables par les symboles de même nom !

Les fonctions `lookup` et `update!` se définissent donc très aisément :

```
(define (lookup id env)
  (if (pair? env)
      (if (eq? (caar env) id)
          (cdar env)
          (lookup id (cdr env)) )
      (wrong "No such binding" id) ) )
```

On voit apparaître un second cas d'erreur qui surgit lorsque l'on désire connaître la valeur d'une variable sans définition. Là encore, on se contentera d'exprimer le trouble dans lequel l'interprète est plongé par un appel à `wrong`.

Dans les vieux temps et principalement lorsque les interprètes disposaient de très peu de mémoire [12], les implanteurs ont souvent favorisé un mode *autoquote* généralisé. Une variable sans valeur en avait quand même une et cette valeur était le symbole de même nom ! Triste retour des choses que d'avoir attaché tant d'importance à séparer symbole et variable et de voir revenir (au galop) l'infernale confusion.

S'il est pratique pour les implanteurs de ne jamais provoquer une erreur et donc d'offrir un monde idyllique d'où l'erreur est bannie, ce noble dessein n'a qu'un inconvénient : le but d'un programme n'est pas de n'être pas erroné mais de remplir son office. L'erreur est une sorte de grossier garde-fou qui, quand elle survient, démontre que le programme est en désaccord avec son intention. Il est de bon aloi que soient signalées le plus rapidement possibles les erreurs et les contextes erronés afin que celles-là soient au plus tôt corrigées. Le mode *autoquote* est donc un mauvais choix de conception car il permet à certaines situations de croupir occultement.

La fonction `update!` qui modifie l'environnement est susceptible de provoquer la même erreur : on ne peut modifier la valeur d'une variable inconnue. Nous reviendrons sur ce point quand nous parlerons d'environnement global.

```
(define (update! id env value)
```

---

11. Comme on le sait, `let` est une macro s'expansant en une application fonctionnelle : `(let ((`x π_1`))` π_2`)` $\equiv$ `((lambda (`x`)` π_2`)` π_1`)`.

12. La mémoire (avec les entrées/sorties), bien que de prix sans cesse décroissant, forme toujours la partie la plus chère d'un calculateur.

```
(if (pair? env)
    (if (eq? (caar env) id)
        (begin (set-cdr! (car env) value)
               value )
        (update! id (cdr env) value) )
    (wrong "No such binding" id) ) )
```

Le contrat de la fonction `update!` prévoyait qu'elle retourne une valeur qui deviendrait celle, finale, de la forme spéciale d'affectation. La valeur de l'affectation n'est pas définie en Scheme, ce qui signifie qu'un programme portable ne peut s'attendre à une valeur précise. Plusieurs possibilités s'offrent comme de retourner *(i)* la valeur qui vient d'être affectée (c'est ce qui est fait plus haut dans `update!`) *(ii)* l'ancien contenu de la variable (ce qui pose un léger problème à l'initialisation lorsque l'on donne une première valeur à la variable) *(iii)* un objet représentant ce qui n'est pas spécifié, sur lequel on ne peut faire grand chose, quelque chose comme `#<UFO>` [13] *(iv)* la valeur d'une forme à valeur non spécifiée comme l'est la forme `set-cdr!` en Scheme.

L'environnement peut être vu comme un type abstrait composite. On peut extraire ou modifier des sous-parties d'un environnement, ce sont les fonctions de sélection ou modification. Il reste à définir comment sont construits et enrichis les environnements.

L'environnement initial est vide, ce qui se représente simplement :

```
(define env.init '())
```

Nous veillerons plus loin à procurer un environnement standard un peu plus étoffé.

Lorsqu'une fonction est appliquée, un nouvel environnement est construit qui lie les variables de la fonction à leur valeur. La fonction `extend` étend un environnement `env` avec une liste de variables `variables` et une liste de valeurs `values`.

```
(define (extend env variables values)
  (cond ((pair? variables)
         (if (pair? values)
             (cons (cons (car variables) (car values))
                   (extend env (cdr variables) (cdr values)) )
             (wrong "Too less values") ) )
        ((null? variables)
             (if (null? values)
                 env
                 (wrong "Too much values") ) )
        ((symbol? variables) (cons (cons variables values) env)) ) )
```

La principale difficulté est qu'il faut analyser syntaxiquement toutes les formes possibles qu'une *<liste-de-variables>* peut posséder au sein d'une abstraction en Scheme [14]. Une *liste de variables* est représentée par une liste de symboles éventuellement pointée, c'est-à-dire terminée non pas par `()` mais par un symbole. Plus formellement, une liste de variables répond à la pseudo-grammaire suivante :

---

13. *Unknown Flying Object* est l'équivalent du célèbre (en français) `**OVNI**` de Le-Lisp.

14. Certains systèmes Lisp, comme COMMON LISP, ont cédé à la tentation d'enrichir la liste de variables de mot-clés divers (`&aux`, `&key`, `&rest` ...) compliquant singulièrement le mécanisme de liaison. D'autres généralisent la liaison des variables au filtrage [SJ93].

$$
\begin{array}{rcl}
\textit{<liste-de-variables>} & := & \texttt{()} \\
 & | & \textit{<variable>} \\
 & | & \texttt{(}\ \textit{<variable>}\ \texttt{.}\ \textit{<liste-de-variables>}\ \texttt{)} \\
\textit{<variable>} & \in & \textbf{Symbol}
\end{array}
$$

Lorsque l'on étend un environnement, il doit y avoir accord entre les noms et les valeurs. Usuellement, il doit y avoir autant de valeurs qu'il y a de variables, sauf si la liste de variables se termine par une variable *pointée* ou n-aire qui peut prendre toutes les valeurs superflues réunies en une liste. Deux erreurs peuvent donc être signalées suivant qu'il y a trop ou pas assez de valeurs.

1.6 Représentation des fonctions

La façon la plus simple de représenter des fonctions est peut-être d'utiliser des fonctions. Ne vous y méprenez pas : les deux emplois du mot « fonction » dans la phrase précédente renvoient à des entités différentes. Plus précisément donc : « La façon la plus simple de représenter des fonctions du langage en cours de définition est d'utiliser des fonctions du langage d'implantation. » La représentation ici choisie va minimiser le protocole d'appel, c'est-à-dire la fonction `invoke` qui n'aura plus qu'à vérifier que son premier argument est bien une fonction, c'est-à-dire un objet applicable :

```
(define (invoke fn args)
  (if (procedure? fn)
      (fn args)
      (wrong "Not a function" fn) ) )
```

On ne peut vraiment faire plus simpliste [15].

Un nouveau cas d'erreur est apparu, lorsque l'on cherche à appliquer un objet qui n'est pas une fonction. La détection se fait ici au moment de l'application, c'est-à-dire après évaluation de tous les arguments. D'autres stratégies sont possibles comme de chercher à prévenir au plus tôt l'utilisateur. On peut, dans ce dernier cas, imposer un ordre dans l'évaluation des applications fonctionnelles :

1. évaluer le terme en position fonctionnelle ;
2. si sa valeur n'est pas applicable, déclencher une erreur ;
3. évaluer les arguments de la gauche vers la droite ;
4. comparer le nombre d'arguments et l'arité de la fonction à appliquer et déclencher une erreur s'il n'y a pas accord.

Ordonner l'évaluation des arguments de la gauche vers la droite a l'intérêt de la simplicité pour ceux qui lisent de gauche à droite et permet de mettre au point plus aisément puisque l'ordre d'évaluation est apparent mais complique toutefois la tâche du compilateur s'il cherche à réordonner l'évaluation des arguments pour, par exemple, améliorer l'allocation des registres car il lui faut prouver que le réordonnancement des expressions ne perturbe pas la sémantique du programme.

---

15. Le lecteur pourrait se demander pourquoi avoir individualisé la fonction `invoke`, de définition aussi simple et seulement appelée une fois depuis `evaluate`. La raison est que nous fixons ici la structure générale des interprètes dont nous allons gaver le lecteur et que nous verrons d'autres codages moins simples mais plus performants requérant de compliquer `invoke`. On pourra aussi, avec fruit, s'atteler aux exercices 1.7 et 1.8.

On peut bien sûr chercher à faire encore mieux, en hâtant la vérification d'arité :

1. évaluer le terme en position fonctionnelle ;
2. si sa valeur n'est pas applicable, déclencher une erreur, sinon déterminer son arité ;
3. évaluer les arguments de la gauche vers la droite tant que leur nombre est toujours en accord avec l'arité précédente, déclencher une erreur sinon ;
4. appliquer la fonction à ses arguments [16].

COMMON LISP impose que les arguments soient évalués de la gauche vers la droite mais permet, pour des raisons d'efficacité, au terme en position fonctionnelle d'être évalué avant ou après. Scheme, quant à lui, n'impose aucun ordre d'évaluation pour les termes d'une application fonctionnelle. La position fonctionnelle n'y est pas distinguée. Qu'il n'y ait pas d'ordre imposé signifie que l'évaluateur est libre de choisir celui qui lui plaît ; le compilateur peut donc réordonner sans souci les termes. L'utilisateur, ne sachant plus quel est l'ordre choisi, doit donc utiliser `begin` lorsqu'il souhaite séquentialiser certains effets. Stylistiquement parlant, il est peu élégant d'user de l'application fonctionnelle pour séquentialiser des effets de bord : évitez donc, par exemple, d'écrire `(f (set! f` π`) (set! f` π'`))` où rien ne transparaît de la fonction qui sera réellement appliquée. Les erreurs de ce genre, dues à la méconnaissance de l'ordre précis d'évaluation sont très dures à détecter.

Environnement d'exécution du corps d'une fonction

L'application d'une fonction revient à évaluer son corps dans un environnement où ses variables sont liées aux valeurs qu'elles ont prises lors de l'application. Rappelons qu'à l'appel de `make-function`, nous lui avions fourni tous les paramètres dont nous disposions depuis `evaluate`. Dans tout le reste de cette section, les différents environnements en jeu seront notés en italiques afin d'attirer agréablement le regard.

Nudisme

Tentons une première définition :

```
(define (make-function variables body env)
  (lambda (values)
     (eprogn body (extend env.init variables values)) ) )
```

Conformément au contrat précédemment exposé, le corps de la fonction est bien évalué dans un environnement où les variables sont liées à leur valeur. Par exemple, le combinateur K ainsi défini `(lambda (a b) a)` peut être appliqué comme dans :

```
(K 1 2)   →  1
```

L'ennui est que les moyens mis à disposition d'une fonction sont dès lors assez réduits. Le corps d'une fonction ne peut rien utiliser d'autre que ses variables puisque l'environnement initial *env.init* est vide. Il n'a même pas accès à l'environnement global dans lequel sont prédéfinis les fonctions usuelles comme `car`, `cons`, etc.

---

16. La fonction pourra alors examiner le type de ses arguments mais cette tâche ne relève pas du protocole d'appel fonctionnel.

Rustine

Réitérons donc notre précédent et infructueux essai :

```
(define (make-function variables body env)
  (lambda (values)
     (eprogn body (extend env.global variables values)) ) )
```

Cette nouvelle version permet donc au corps des fonctions d'utiliser l'environnement global et toutes ses fonctions usuelles. Mais comment définir globalement deux fonctions mutuellement récursives et que vaut l'expression de gauche qui figure macroexpansée à droite ?

```
(let ((a 1))                          ((lambda (a)
  (let ((b (+ 2 a)))      ≡             ((lambda (b)
     (list a b) ) )                        (list a b) )
                                         (+ 2 a) ) )
                                      1 )
```

Détaillons comment est évaluée cette expression :

```
((lambda (a) ((lambda (b) (list a b)) (+ 2 a))) 1)|
                                                  | env.global
= ((lambda (b) (list a b)) (+ 2 a))|
                                   | a→ 1
                                   | env.global

= (list a b)|
            | b→ 3
            | env.global
```

Le corps de la fonction interne `(lambda (b) (list a b))` s'évalue dans un environnement obtenu en étendant l'environnement global par la variable `b`. Cet environnement exclut la variable `a` qui n'est pas visible ! Notre essai échoue donc encore.

Rustine révisée

Puisque l'on désire voir la variable `a` au sein de la fonction interne, il suffit de fournir l'environnement courant à `invoke` qui le transmettra aux fonctions appelées. Il y a donc une notion d'environnement courant, tenu à jour par `evaluate` et `invoke`. Modifions donc ces fonctions mais, toutefois, afin de ne pas confondre ces nouvelles (et temporaires) définitions avec les anciennes, nous les préfixerons par `d.`.

```
(define (d.evaluate e env)
  (if (atom? e) ...
      (case (car e)
        ...
        ((lambda) (d.make-function (cadr e) (cddr e) env))
        (else (d.invoke (d.evaluate (car e) env)
                        (evlis (cdr e) env)
                        env )) ) ) )
(define (d.invoke fn args env)
  (if (procedure? fn)
      (fn args env)
      (wrong "Not a function" fn) ) )

(define (d.make-function variables body def.env)
  (lambda (values current.env)
     (eprogn body (extend current.env variables values)) ) )
```

On pourra observer dans la définition de `d.make-function` que fournir l'environnement de définition `env` n'est plus utile puisque seul est utilisé l'environnement courant *current.env*.

Et maintenant, notre exemple marche comme le montre la même expression redécorée des environnements utilisés :

```
((lambda (a) ((lambda (b) (list a b)) (+ 2 a))) 1)|
                                                  | env.global
= ((lambda (b) (list a b)) (+ 2 a))|
                                   | a→ 1
                                   | env.global
= (list a b)|
            | b→ 3
            | a→ 1
            | env.global
```

On voit bien sur cet exemple la discipline de pile qu'adoptent les liaisons. Toute forme liante empile ses nouvelles liaisons sur l'environnement courant et les enlève après exécution.

Rustine révisée révisée

Mais il y a encore un problème. Considérez cette légère variation :

```
(((lambda (a)
    (lambda (b) (list a b)) )
  1 )
 2 )
```

La fonction `(lambda (b) (list a b))` est créée dans un environnement où `a` est lié à `1` mais au moment où elle est appliquée, l'environnement courant est étendu par la seule liaison concernant `b` d'où, encore une fois, `a` est absente. La fonction `(lambda (b) (list a b))` a donc oublié (ou perdu) sa variable `a`.

Le lecteur avait pu le remarquer dans la précédente définition de `d.make-function`, deux environnements étaient présents : l'environnement de définition de la fonction *def.env* et l'environnement d'appel de la fonction, *current.env*. Deux types d'instants marquent la vie d'une fonction : sa naissance et ses applications. S'il n'y a qu'une seule et unique naissance, il peut en revanche y avoir de nombreuses applications voire même aucune. Le *seul* environnement que l'on puisse donc naturellement associer à une fonction est celui de sa création [17]. Revenons donc aux précédentes définitions des fonctions `evaluate` et `invoke` puis posons que `make-function` est maintenant :

```
(define (make-function variables body env)
   (lambda (values)
      (eprogn body (extend env variables values)) ) )
```

Tous les exemples se comportent alors favorablement et en particulier le précédent qui a pour trace d'évaluation :

```
(((lambda (a) (lambda (b) (list a b))) 1) 2)|
                                            | env.global
= ( (lambda (b) (list a b))|
                           | a→ 1
                           | env.global
```

17. On peut aussi laisser au programme le soin de choisir explicitement l'environnement dans lequel clore la fonction, voir la forme `closure` page 107.

```
      2 )|
         |  env.global
=  (list a b)|
            |  b→ 2
            |  a→ 1
            |  env.global
```

La forme `(lambda (b) (list a b))` est créée dans l'environnement global étendu par la variable `a`. Lorsqu'elle est appliquée (dans l'environnement global), elle étend son environnement de définition par `b` et permet donc à son corps de s'évaluer dans un environnement où `a` et `b` sont connues. Lorsqu'elle retourne son résultat, l'évaluation se poursuit dans l'environnement global. On nomme souvent *fermeture* la valeur d'une abstraction car elle clôt (ou enferme) son environnement de définition.

On remarquera que la présente définition de `make-function` use elle-même (au sein du langage de définition) de fermetures. Ce n'est pas obligatoire comme on le verra au chapitre 3. **[☞ p. 87]** La fonction `make-function` a pour valeur une fermeture, ceci est un trait distinctif des langages fonctionnels d'ordre supérieur.

1.6.1 Liaisons dynamique et lexicale

La précédente discussion a au moins deux mérites. En premier lieu, elle exhibe clairement la complexité liée aux problèmes d'environnement. Une évaluation s'effectue toujours au sein d'un certain environnement et la gestion de celui-ci est un point majeur que doivent résoudre efficacement les évaluateurs. Nous verrons, au chapitre 3, des constructions plus complexes, comme les échappements et la forme `unwind-protect` qui nécessitent de définir très précisément quels sont les environnements en lice.

Les deux dernières variantes présentées plus haut caractérisent respectivement (comme on les nomme usuellement) les liaisons *dynamique* [18] ou *lexicale*. Dans un *Lisp lexical*, une fonction évalue son corps dans son environnement de définition étendu par ses variables tandis que, dans un *Lisp dynamique*, une fonction étend l'environnement courant, c'est-à-dire celui d'application. Si la mode actuellement préfère les Lisps lexicaux, il ne faut pas en conclure que les langages dynamiques n'ont aucun avenir. D'une part, il existe des langages fort utiles, récents et dynamiques comme TEX [Knu84], Gnu Emacs Lisp [LLSt93] ou Perl [WS90] [19] ; d'autre part, le concept de liaison dynamique est un important concept de programmation : il correspond à l'établissement d'une liaison valide seulement le temps d'un calcul et défaite automatiquement (de façon garantie) à l'achèvement de celui-ci [20]. Ce tour programmatique peut être employé avec fruit dans des calculs prospectifs comme par exemple ceux que l'on mène en intelligence artificielle. On pose une hypothèse à partir de laquelle on développe des conséquences. Lorsque l'on découvre une incohérence, on se doit

---

18. Dans le contexte des langages à objets, le terme de *liaison dynamique* est plutôt associé au fait que la méthode associée à un message est déterminée par le type dynamique de l'objet à qui il est envoyé plutôt que par son type statique.

19. Perl depuis procure à la fois des variables dynamiques (avec le mot-clé `local`) et des variables lexicales (avec le mot-clé `my`).

20. C'est cette même fin de calcul que capture la clause `finally` d'une forme `try` dans de nombreux langages tels que Caml, Java, Perl, etc.

d'abandonner cette hypothèse afin d'en explorer une autre : c'est la technique bien connue de retour arrière. Si les conséquences ont été enregistrées sans aucun effet de bord, par exemple dans des structures telles que des listes d'association, l'abandon de l'hypothèse recyclera automatiquement les conséquences. Si, par contre, on avait usé de modifications physiques telles que des affectations de variables globales, modifications de tableaux, etc., alors l'abandon de l'hypothèse doit conduire à la restauration de l'environnement dans lequel cette dernière fut formulée. Un seul oubli serait fatal ! La liaison dynamique permet d'assurer que, pendant tout un calcul et quelle que soit l'issue de celui-ci, la variable dynamique sera toujours correctement positionnée.

Les variables sont des entités programmatiques dotées d'une *portée*. La portée d'une variable est une notion essentiellement géographique correspondant à la zone du texte du programme où elle est visible et donc accessible. En Scheme pur (c'est-à-dire débarrassé des syntaxes superflues, mais utiles, telle `let`), une seule forme liante existe : `lambda`, c'est la seule forme introduisant des variables et leur conférant une portée réduite au corps seul de la fonction. La portée d'une variable dans un Lisp dynamique n'est par contre et a priori aucunement restreinte. Considérez l'exemple suivant :

```
(define (foo x) (list x y))
(define (bar y) (foo 1991))
```

En Lisp lexical, la variable `y` dans `foo` est une référence à la variable globale `y` qui ne peut aucunement être confondue avec la variable locale `y` de `bar`. En Lisp dynamique, la variable `y` de `bar` est visible, et vue, du corps de la fonction `foo`[21] puisque lorsque `foo` est invoquée, l'environnement courant contient cette variable `y`. Si l'on donne donc la valeur `0` à la variable globale `y`, on obtient alors les résultats suivants :

```
(define y 0)
(list (bar 100) (foo 3))    → ((1991 0) (3 0))      en Lisp lexical
(list (bar 100) (foo 3))    → ((1991 100) (3 0))    en Lisp dynamique
```

On remarquera, en Lisp dynamique, que `bar` n'a aucun moyen de savoir que la fonction `foo` qu'elle appelle référence sa propre variable `y`. Inversement, la fonction `foo` ne sait pas où peut se trouver la variable `y` qu'elle référence. C'est pour cela que `bar` doit installer `y` dans l'environnement courant afin que `foo` puisse l'y retrouver. Enfin et juste avant que `bar` ne retourne, `y` doit être retirée de l'environnement courant. On remarquera cependant qu'en l'absence de variables libres, les deux sortes de Lisp sont indiscernables.

La liaison lexicale a été ainsi nommée car l'on peut toujours, de toute variable et à partir du simple texte de la fonction, retrouver la forme l'ayant liée ou savoir si c'est une variable globale. La méthode est simple, on pointe son crayon (ou sa souris) sur la variable et on remonte de la droite vers la gauche, de bas en haut jusqu'à trouver la première forme englobante liant cette même variable.

Le nom de la liaison dynamique joue sur un autre concept qui est la *durée de vie dynamique* que l'on étudiera plus tard. **[☞ p. 73]** Scheme ne connaît de variables que lexicales ; COMMON LISP procure les deux sortes de liaisons sous une même syntaxe ; EULISP et IS-Lisp enfin séparent clairement (et syntaxiquement) les deux en deux espaces de noms séparés. **[☞ p. 40]**

---

21. Pour l'étymologie de `foo`, voir [Ray91].

La portée peut être occultée localement par masquage : une variable peut en cacher une autre si elles portent le même nom. Les formes liantes étant lexicalement emboîtées, les portées lexicales sont donc disjointes ou emboîtées : c'est la fameuse discipline de bloc héritée d'Algol 60.

Légèrement inspiré du λ-calcul à qui il a emprunté le nom de la forme spéciale `lambda` [Per79], Lisp 1.0 fut défini comme un Lisp dynamique. Toutefois John McCarthy a reconnu assez tôt qu'il attendait que l'expression suivante retournât `(2 3)` plutôt que `(1 3)` :

```
(let ((a 1))
  ((let ((a 2)) (lambda (b) (list a b)))
   3 ) )
```

Cette anomalie (bogue ?) fût corrigée par introduction d'une nouvelle forme spéciale nommée `function` dont l'argument était une forme `lambda` et qui créait une fermeture, c'est-à-dire une fonction associée à son environnement de définition. Lorsque cette fermeture était appliquée, au lieu d'étendre l'environnement courant, elle étendait son environnement de définition qu'elle avait clos (c'est-à-dire préservé) en elle. Plus programmatiquement, la forme spéciale `function` [22] se définirait ainsi avec `d.evaluate` et `d.invoke` :

```
(define (d.evaluate e env)
  (if (atom? e) ...
      (case (car e)
        ...
        ((function)   ; Syntax: (function (lambda variables body))
         (let* ((f (cadr e))
                (fun (d.make-function (cadr f) (cddr f) env)) )
           (d.make-closure fun env) ) )
        ((lambda) (d.make-function (cadr e) (cddr e) env))
        (else (d.invoke (d.evaluate (car e) env)
                        (evlis (cdr e) env)
                        env )) ) ) )

(define (d.invoke fn args env)
  (if (procedure? fn)
      (fn args env)
      (wrong "Not a function" fn) ) )

(define (d.make-function variables body env)
  (lambda (values current.env)
     (eprogn body (extend current.env variables values)) ) )

(define (d.make-closure fun env)
  (lambda (values current.env)
     (fun values env) ) )
```

Mais l'histoire ne se finit pas avec `function` qui fut considérée comme une complaisance que l'utilisateur devait accorder à une implantation insuffisante. Les premiers compilateurs se rendirent très vite compte que, pour des raisons de performance, les environnements lexicaux avaient un très grand avantage qui était que l'on savait

22. Cette simulation n'est point tout à fait exacte en ce sens que dans de nombreux dialectes (comme dans [Ste84]), `lambda` n'est pas une forme spéciale mais un mot-clé (une sorte de marqueur syntaxique similaire au `else` qui peut figurer, en Scheme, dans un `cond` ou un `case`). Il est en effet trivial de remarquer que `lambda` ne nécessite pas `d.evaluate` pour être traitée et de restreindre les emplacements où peuvent apparaître les formes `lambda` : en premier terme des applications fonctionnelles, accompagnée de `function` ou dans les définitions fonctionnelles.

dès la compilation, et dans tous les cas, où se trouveraient les variables à l'exécution : un accès quasi direct pouvait être engendré plutôt qu'une recherche dynamique de la valeur. Par défaut donc, toutes les variables des fonctions compilées étaient considérées comme lexicales, sauf celles qui étaient explicitement déclarées comme dynamiques ou, dans le jargon, *spéciales*. La déclaration `(declare (special x))` pour le seul usage du compilateur (Lisp 1.5, MacLisp, COMMON LISP etc.) désigne la variable x comme ayant un comportement spécial.

Mais il n'y avait pas seulement ces raisons d'efficacité, il y a aussi la perte de la *transparence référentielle*. La transparence référentielle est la propriété que possède un langage quand la substitution d'une expression dans un programme par une autre expression équivalente (c'est-à-dire calculant la même chose) ne modifie pas le comportement du programme : ils retournent la même valeur ou ne terminent pas tous les deux. Considérez par exemple :

```
(let ((x (lambda () 1))) (x)) ≡ ((let ((x (lambda () 1))) x)) ≡ 1
```

La transparence référentielle est perdue dès que le langage dispose d'effets de bord ; dans ce cas, de nouvelles définitions plus précises quant à la relation d'équivalence à employer pour parler de transparence référentielle sont nécessaires. Scheme, sans affectation, ni effets de bord, ni manipulation de continuation est référentiellement transparent. **[☞ Ex. 3.10]** C'est aussi un but vers lequel on doit tendre lorsque l'on cherche à écrire des programmes réutilisables, c'est-à-dire dépendant le moins possible du contexte d'emploi.

Les variables d'une fonction telle que `(lambda (u) (+ u u))` sont ce que l'on nomme traditionnellement, *muettes*. Leur nom n'a aucune importance et peut être remplacé par tout autre nom. La fonction `(lambda (n347) (+ n347 n347))` n'est ni plus ni moins [23] que la fonction `(lambda (u) (+ u u))`. On peut donc attendre du langage que soit respecté cet invariant. Il n'en est rien en Lisp dynamique. Considérez seulement [24] :

```
(define (map fn l)          ;  mapcar in Lisp
  (if (pair? l)
      (cons (fn (car l)) (map fn (cdr l)))
      '() ) )
(let ((l '(a b c)))
  (map (lambda (x) (list-ref l x))
       '(2 1 0) ) )
```

En Scheme, le résultat serait `(c b a)` mais deviendrait `(0 0 0)` en Lisp dynamique ! La raison est que la variable `l` qui est libre dans le corps de `(lambda (x) (list-ref l x))` est interceptée, capturée, accrochée par la variable `l` de `map`.

On peut simplement résoudre cet ennui en changeant les noms en conflit, il suffit de renommer l'une des deux variables `l`. On choisira de renommer celle de `map` qui est probablement plus utile. Mais quel nom choisir qui puisse à peu près garantir que le même problème ne resurgira pas ? Nous suggérons de préfixer les noms des variables par le numéro de sécurité sociale du programmeur et de les suffixer par l'heure universelle exprimée en centièmes de seconde sexagésimale de temps moyen compté

---

23. En termes techniques du λ-calcul, ce changement de noms de variables s'appelle une α-conversion.
24. La fonction `list-ref` extirpe d'une liste son élément de rang n.

à partir du 1er janvier 1901 à midi. Les risques de collision seront ainsi nettement amoindris mais seulement amoindris, au péril de la lisibilité !

La situation au début des années quatre-vingt était donc particulièrement délicate. On enseignait Lisp par l'observation de son interprète qui différait du compilateur sur le point (fondamental) de la valeur d'une variable. Scheme ayant montré dès 1975 [SS75] que l'on pouvait réconcilier interprète et compilateur et donc vivre dans un monde complètement lexical, la définition de COMMON LISP entérina ce fait et postula que la *bonne* sémantique était celle du compilateur, que donc tout était lexical par défaut. L'interprète n'avait plus qu'à se conformer à ces nouveaux canons. Le succès grandissant de Scheme, des langages fonctionnels tel ML et de leurs divers épigones, répandit puis imposa cette nouvelle vue.

1.6.2 Implantation superficielle ou profonde

Mais les choses ne sont pas si simples et comme dans la fable de la lance et de la cuirasse, les implanteurs ont su trouver des moyens d'accélérer la vitesse de recherche de la valeur des variables dynamiques. Quand l'environnement est représenté comme une liste d'association, le coût de la recherche de la valeur d'une variable (la fonction `lookup`) est linéaire par rapport à la longueur de la liste[25]. Ce mécanisme est nommé *liaison profonde* (*deep binding*). Une autre technique existe, dite de *liaison superficielle* (*shallow binding*), où à chaque variable est associé un endroit où est toujours stockée sa valeur indépendamment de l'environnement courant. La plus simple réalisation est que cet endroit soit un champ particulier du symbole associé à la variable : la `Cval` ou *value-cell*. Le coût de `lookup` devient constant (une indirection suivie d'un éventuel déplacement). Comme il y a rarement gain sans perte, il faut avouer que l'appel fonctionnel est alors plus coûteux puisqu'il faut sauvegarder les valeurs des variables qui vont être liées et modifier les symboles associés aux variables pour qu'ils contiennent leurs nouvelles valeurs. Au retour de la fonction, il faut en plus restaurer les anciennes valeurs dans les symboles d'où elles provenaient, ce qui peut compromettre la récursion terminale (mais voir [SJ93]).

On peut simuler partiellement la liaison superficielle[26] en changeant la représentation de l'environnement. On supposera dans ce qui suit que les listes de variables ne sont pas pointées (ce qui allège le décodage des listes de variables) et l'on ne vérifiera pas l'arité des fonctions. Ces nouvelles fonctions sont préfixées par `s.` afin d'être reconnues.

```
(define (s.make-function variables body env)
  (lambda (values current.env)
    (let ((old-bindings
           (map (lambda (var val)
                  (let ((old-value (getprop var 'apval)))
                    (putprop var 'apval val)
                    (cons var old-value) ) )
```

25. Fort heureusement, les statistiques montrent que l'on cherche plus souvent les premières variables que celles qui sont plus profondément enfouies. D'autre part on remarquera que les environnements lexicaux sont plus courts que les environnements dynamiques qui comportent toutes les liaisons en cours de calcul.

26. L'affectation d'une variable close [BCSJ86] n'est toutefois pas traitée ici.

```
                variables
                values ) ))
        (let ((result (eprogn body current.env)))
          (for-each (lambda (b) (putprop (car b) 'apval (cdr b)))
                    old-bindings )
          result ) ) ) )
(define (s.lookup id env)
  (getprop id 'apval) )
(define (s.update! id env value)
  (putprop id 'apval value) )
```

Les fonctions `putprop` et `getprop` (non standard, en Scheme, mais semblables aux fonctions `put` et `get` qui apparaissent dans [AG85] **[☞ Ex. 2.6]** car elles mettent en œuvre de fort inefficaces effets de bord globaux) simulent le champ[27] dans lequel les symboles mémorisent la valeur des variables éponymes. Indépendamment de leur implantation réelle[28], ces fonctions doivent être considérées comme de coût constant.

On remarquera dans la simulation précédente l'évanouissement complet de l'environnement `env` qui ne sert plus à rien. Cette disparition impose que soit modifiée l'implantation des fermetures car elles ne peuvent plus clore l'environnement puisque celui-ci n'existe plus ! La technique, que l'on reverra plus tard, consiste en l'analyse du corps de la fonction pour en extirper les variables libres et les traiter de la belle manière.

La liaison profonde favorise le changement d'environnement et la programmation multi-tâches au détriment de la recherche des valeurs des variables ; la liaison superficielle favorise la recherche des valeurs des variables au détriment de l'appel fonctionnel. Les deux techniques ont été mariées par Henry Baker [Bak78] : c'est la technique de marcottage ou *rerooting*. Rappelons pour finir que liaisons superficielle ou profonde ne sont que des techniques d'implantation qui n'ont rien à voir avec la sémantique des liaisons.

1.7 Environnement global

Un environnement global vide fait pauvre. La plupart des systèmes Lisp procurent des bibliothèques (pour *libraries*) plus qu'étoffées : plus de 700 fonctions pour COMMON LISP(CLtL1), plus de 1 500 pour Le-Lisp, plus de 10 000 pour ZetaLisp etc. Sans bibliothèque, Lisp ne serait qu'une sorte de λ-calcul dans lequel on ne pourrait même pas imprimer les résultats de ses computs. La notion de bibliothèque est importante car, si l'essentiel du langage, pour le réalisateur de l'évaluateur, tient dans les formes spéciales, ce sont les bibliothèques qui souvent font la différence pour l'utilisateur. Le folklore Lisp retient que c'est l'absence des fonctions trigonométriques banales qui ont fait s'écarter de Lisp les numériciens des premiers âges. Savoir différencier

---

27. Le nom de la propriété utilisée, `apval`, est un hommage au nom utilisé dans [MAE+62] lorsque les valeurs étaient *réellement* stockées dans les P-listes.

28. Ces fonctions balayent linéairement la liste des propriétés (ou P-liste) des symboles jusqu'à trouver la propriété concernée. Le coût de cette recherche est linéaire en la longueur de la P-liste sauf si l'implantation associe symboles et propriétés par une table de hachage.

ou intégrer symboliquement [Sla61] des fonctions, c'est bien mais sans sécante, ni cosécante, que peut-on faire ?

On s'attend à trouver dans l'environnement global toutes les fonctions usuelles comme `car`, `cons`, etc. On y trouve aussi de simples variables dont les valeurs sont des données bien connues comme les booléens ou la liste vide. Par commodité et bien que nous n'ayons encore rien dit des macros **[☞ p. 295]** qui sont des phénomènes très complexes que nous analyserons plus tard, nous allons en définir deux qui amélioreront l'élaboration de l'environnement global. On définira l'environnement global comme une extension de l'environnement initial (vide) `env.init`.

```
(define env.global env.init)
(define-syntax definitial
  (syntax-rules ()
    ((definitial name)
     (begin (set! env.global (cons (cons 'name 'void) env.global))
            'name ) )
    ((definitial name value)
     (begin (set! env.global (cons (cons 'name value) env.global))
            'name ) ) ) )
(define-syntax defprimitive
  (syntax-rules ()
    ((defprimitive name value arity)
     (definitial name
        (lambda (values)
          (if (= arity (length values))
              (apply value values)        ; The real apply of Scheme
              (wrong "Incorrect arity"
                     (list 'name values) ) ) ) ) ) ) )
```

On définira tout d'abord quelques constantes bien utiles qui, toutes les trois, n'appartiennent pas au standard Scheme. Notez que `t` est une variable du Lisp en cours de définition tandis que `#t` est une valeur dans le Lisp de définition. Toute valeur autre que celle de `the-false-value` ferait tout aussi bien l'affaire.

```
(definitial t #t)
(definitial f the-false-value)
(definitial nil '())
```

S'il est utile de disposer de variables globales permettant d'obtenir les véritables objets qui représentent les booléens ou la liste vide, une autre solution est de procurer une syntaxe appropriée pour ce faire. Scheme use des syntaxes `#t` et de `#f` dont les valeurs sont les booléens *vrai* et *faux*. L'intérêt de ces syntaxes est qu'elles sont impervertissables et toujours visibles : (*i*) visible globalement, car on peut écrire dans tout contexte `#t` pour dire *vrai*, même si une variable locale se nomme `t` ! ; (*ii*) impervertissable, car de nombreux évaluateurs autorisent d'altérer la valeur de la variable `t`, ce qui conduit à de délicieux casse-têtes où l'on peut voir `(if t 1 2)` valoir 2 comme dans `(let ((t #f)) (if t 1 2))`. Plusieurs solutions sont possibles. La plus simple est de visser l'immuabilité de ces constantes dans l'évaluateur :

```
(define (evaluate e env)
  (if (atom? e)
      (cond ((eq? e 't) #t)
            ((eq? e 'f) #f)
            ...
            ((symbol? e) (lookup e env))
```

```
            ...
            (else (wrong "Cannot evaluate" e)) )
      ... ) )
```

On peut aussi introduire le concept de liaison *muable* ou *immuable*. Une liaison immuable ne peut faire l'objet d'une affectation. Nul ne peut donc altérer la valeur d'une variable ayant une liaison immuable. Ce concept existe de fait, quoique occultement, dans de nombreux systèmes. Le concept de *fonctions intégrables* désigne les fonctions (*inlined* ou *open coded*) **[☞ p. 190]** dont le corps convenablement instancié peut remplacer les appels. Remplacer `(car x)` par le code qui extrait, de la valeur de la paire pointée `x`, le contenu de son champ `car` sous-entend l'hypothèse importante que nul n'altère (n'a altéré ou n'altèrera) la valeur de la variable globale `car`. Songez au malheur qu'illustre le fragment de session suivant :

```
(set! my-global (cons 'c 'd))
   → (c . d)
(set! my-test (lambda () (car my-global)))
   → #<MY-TEST procedure>
(begin (set! car cdr)
       (set! my-global (cons 'a 'b))
       (my-test) )
   → ?????
```

Heureusement encore, la réponse ne peut être que `a` ou `b` ! Si `my-test` utilise la valeur de `car`, qui était courante au moment où `my-test` fut définie, la réponse sera `a`. Si `my-test` utilise la valeur courante de `car`, la réponse sera `b`. On comparera avec fruit le comportement de `my-test` et de `my-global`, sachant que l'on observe plutôt le premier comportement pour `my-test` (quand l'évaluateur est un compilateur) et plutôt le second pour `my-global`. **[☞ p. 51]**

On ajoutera quelques variables de travail usuelles à l'environnement global car il n'y a pas de création dynamique de variables globales dans l'évaluateur présenté. Les noms proposés ci-dessous recouvrent plus de 96% des noms de fonctions auxquels songent spontanément les lispiens soucieux d'éprouver un nouvel évaluateur [29] :

```
(definitial foo)
(definitial bar)
(definitial fib)
(definitial fact)
```

Enfin l'on définira quelques fonctions mais pas toutes car leur énoncé serait soporifique. La principale difficulté est d'adapter les primitives du Lisp sous-jacent au protocole d'appel du Lisp en cours de définition. Sachant que les arguments sont réunis par l'interprète en une liste, il suffit d'utiliser `apply` [30]. On remarquera que le respect de l'arité de la primitive est assuré grâce à l'expansion que procure `defprimitive`.

```
(defprimitive cons cons 2)
(defprimitive car car 1)
(defprimitive set-cdr! set-cdr! 2)
(defprimitive + + 2)
(defprimitive eq? eq? 2)
(defpredicate < < 2)
```

---

29. Ces variables sont malheureusement initialisées. C'est un défaut qui sera corrigé plus tard.
30. Félicitons-nous encore de ce qu'`invoke` ne s'appelle pas `apply` !

1.8 Lancement de l'interprète

Il ne reste plus qu'à indiquer au lecteur, pressé de mettre en œuvre le langage que nous venons de définir, comment « zapper » dans ce nouveau monde.

```
(define (chapter1-scheme)
  (define (toplevel)
     (display (evaluate (read) env.global))
     (toplevel) )
  (toplevel) )
```

On pourra, en guise d'exercice, affubler de quelques invites spirituelles (voire même zozotantes, ce que signifie *to lisp*), cet interprète quelque peu lapidaire ainsi qu'implanter une fonction permettant d'en sortir.

1.9 Conclusion

A-t-on vraiment défini un langage ? Nul ne peut douter que la fonction `evaluate` peut être lancée, qu'on peut lui soumettre des expressions dont elle retourne patiemment la valeur, quand le calcul termine, bien évidemment. Mais la fonction `evaluate` elle-même n'a de sens que par rapport au langage qui a servi de définition et, en absence de définition du langage de définition, rien n'est sûr. Le réflexe d'auto-amorçage circulaire jouant naturellement chez tout lispien de pure souche, il suffit d'identifier le langage de définition à celui qu'on définit. On a donc un langage L défini par une fonction `evaluate` écrite dans le langage L. Le langage défini est donc solution de l'équation en L :

$$\forall \pi \in \textbf{Programme}, L\,\texttt{(evaluate (quote } \pi\texttt{) env.global)} \equiv L\pi$$

Pour n'importe quel programme π, l'évaluation de π en L (notée $L\pi$) doit exhiber le même comportement (et donc la même valeur si π termine) que l'évaluation de l'expression `(evaluate (quote` π`) env.global)` toujours en L. Une conséquence amusante de cette équation est que la fonction `evaluate` est capable de s'auto-interpréter[31] : Les expressions suivantes sont donc équivalentes :

```
(evaluate (quote π) env.global) ≡
  (evaluate (quote (evaluate (quote π) env.global)) env.global)
```

Existe-t-il des solutions à cette équation ? La réponse est oui et elles sont légion ! Comme on l'a déjà vu plus haut, l'ordre d'évaluation n'est pas nécessairement apparent dans la définition d'`evaluate` et beaucoup d'autres propriétés du langage sont inconsciemment *héritées* par le langage défini. On ne peut en fait presque rien dire car il existe aussi des solutions triviales à cette équation comme le langage L_{2001}, qui a pour sémantique que tout programme écrit en L_{2001} a pour valeur le nombre 2001. Le langage L_{2001} satisfait trivialement l'équation précédente. Il faudra donc recourir à d'autres méthodes si l'on souhaite vraiment définir un langage et ce sera l'objet de chapitres ultérieurs.

---

31. Il faut cependant débarrasser la définition d'`evaluate` des abréviations syntaxiques ou macros (`define`, `case`, etc.) et lui fournir un environnement global contenant les variables `evaluate`, `evlis`, etc.

1.10 Exercices

Exercice 1.1 : Modifiez la fonction `evaluate` en un traceur. Tous les appels fonctionnels verront leurs arguments et leur résultat imprimés. On songera avec fruit à l'extension de ce traceur rudimentaire en un metteur au point pas à pas susceptible de modifier le cheminement de l'exécution du programme ainsi contrôlé.

Exercice 1.2 : L'évaluation par `evlis` d'une liste contenant une seule expression nécessite une récursion un peu inutile. Trouvez-la puis éliminez-la.

Exercice 1.3 : La fonction `extend` est maintenant définie comme suit :

```
(define (extend env names values)
  (cons (cons names values) env) )
```

Définissez les fonctions associées `lookup` et `update!` ; comparez-les avec les précédentes définitions.

Exercice 1.4 : Une autre implantation de liaison superficielle est suggérée par le concept de *rack* [SS80]. À chaque symbole est associée non plus un champ pour contenir la valeur de la variable de même nom, mais une pile. À tout instant la valeur de la variable est la valeur qui se trouve au sommet de la pile des valeurs associée. Réécrivez les fonctions `s.make-function`, `s.lookup` et `s.update!` en profitant de cette nouvelle représentation.

Exercice 1.5 : La définition de la primitive `<` est fausse ! Elle retourne en effet un booléen de l'implantation au lieu d'un booléen du langage en cours de définition. Corrigez ce défaut.

Exercice 1.6 : Définissez la fonction `list`.

Exercice 1.7 : Pour les amoureux des continuations, définissez `call/cc`.

Exercice 1.8 : Définissez `apply`.

Exercice 1.9 : Définissez une fonction `end` permettant de sortir proprement de l'interprète présenté dans ce chapitre.

Exercice 1.10 : Mesurez le rapport de vitesse entre Scheme et `evaluate` puis entre `evaluate` et `evaluate` interprétée par `evaluate`.

Exercice 1.11 : La séquence `begin` avait été définie **[☞ p. 10]** par le biais de `lambda` mais usait de `gensym` afin d'éviter une éventuelle capture. Redéfinissez `begin` dans le même esprit sans user de `gensym`.

1.11 Lectures recommandées

Toutes les références aux interprètes mentionnés en début de chapitre sont intéressantes mais, s'il ne faut lire que quelques documents, les plus enrichissants sont probablement :

— parmi les « *λ-papers* » : [SS78a] ;
— le plus court article présentant un évaluateur qui ait jamais été écrit : [McC78b] ;
— pour déjà goûter un brin de formalisme non pédant : [Rey72] ;
— enfin, pour connaître la genèse, [MAE+62].

2

Lisp 1, 2 . . . ω

Parce que la fonction occupe une place centrale en Lisp et que son efficacité est primordiale, de nombreuses variantes ont été expérimentées autour d'elle dont certaines sont encore observables aujourd'hui. Ce chapitre décrit diverses conceptions des fonctions et des applications fonctionnelles, menant notamment à ce qu'il est convenu de nommer les Lisp_1 ou Lisp_2 ainsi qu'à la notion d'espaces séparés de noms. Le chapitre se poursuivra par une étude de la récursion et de sa mise en œuvre dans ces divers contextes.

Parmi tous les objets que peut manipuler un évaluateur, les fonctions représentent un cas singulier. C'est un type de base ayant un créateur très particulier, `lambda`, et au minimum une seule opération licite : l'application. On ne peut guère moins contraindre un type sans lui faire perdre tout intérêt et c'est d'ailleurs ce peu de qualités qui rendent les fonctions particulièrement attractives pour des spécifications ou des encapsulations car, opaques, elles ne permettent que ce pour quoi elles sont programmées. On peut, par exemple, user de fonctions pour représenter des objets, dotés de champs et de méthodes [AR88] ; les schemeurs les prisent par-dessus tout.

Accroître l'efficacité des fonctions a motivé de nombreuses variantes souvent incompatibles. Historiquement, Lisp 1.5 [MAE+62] ne (re)connaissait pas la notion d'objet fonctionnel. Sa réalisation interne n'excluait d'ailleurs pas que, sous un même nom, puissent coexister une variable, une fonction et une macro, toutes trois représentées par des propriétés différentes (`APVAL`, `EXPR` ou `MACRO` [1]) sur la P-liste du symbole associé. MacLisp a privilégié les fonctions nommées et, récemment seulement, son descendant COMMON LISP (CLtL 2 [Ste90]) a introduit les objets fonctionnels de première classe. En COMMON LISP, `lambda` est un mot-clé syntaxique déclarant, en quelque sorte : « Attention : définition de fonction anonyme. » Les formes `lambda` n'ont pas de valeur et ne peuvent apparaître qu'en des emplacements syntaxiques bien particuliers : en premier terme d'une application ou en premier paramètre de la forme spéciale `function`.

D'un autre côté, Scheme, dès sa naissance en 1975, diffusa le concept d'objet fonctionnel et d'unique espace de valeur en conférant à presque tout un statut de *ci-*

1. Ces propriétés servent respectivement à stocker la valeur globale d'une variable (*A Permanent VALue*), une définition globale de fonction (définie par une *EXPR*ession) ou encore une macro.

toyen de première classe. Un citoyen de première classe peut être argument ou valeur de fonction, stocké dans une variable, une liste, un tableau, etc. La philosophie de Scheme est, par ailleurs, confortée par les langages fonctionnels de la classe de ML et c'est celle que nous adoptons.

2.1 Lisp$_1$

L'interprète du chapitre précédent était conforme à cette philosophie : la notion d'*objet fonctionnel* y existe (`make-function` en est son créateur) et le processus d'évaluation des termes d'une application ne différencie pas la fonction de ses arguments, c'est-à-dire l'expression en *position fonctionnelle* de celles situées dans les positions suivantes dites *paramétriques*. Rappelons de l'interprète précédent les seuls fragments qui nous intéressent :

```
(define (evaluate e env)
  (if (atom? e) ...
      (case (car e)
        ...
        ((lambda) (make-function (cadr e) (cddr e) env))
        (else (invoke (evaluate (car e) env)
                      (evlis (cdr e) env) )) ) ) )
```

Les points saillants sont que :

1. `lambda` est une forme spéciale créant des objets de première classe : des fermetures capturant leur environnement de définition.
2. Tous les termes d'une application sont évalués par le même évaluateur : `evaluate` (et `evlis` qui n'est qu'un `evaluate` distribué sur une liste d'expressions).

Cette dernière caractéristique fait de Scheme un Lisp$_1$.

2.2 Lisp$_2$

Les programmes Lisp sont généralement tels que la plupart des applications fonctionnelles qui y figurent ont, en position fonctionnelle, le nom d'une fonction globale. C'est notamment le cas de tous les programmes des chapitres précédents. On peut restreindre la grammaire de Lisp pour toujours imposer un symbole en `car` de toute forme : cela ne modifie pas sensiblement l'aspect du langage mais implique que l'évaluation qui est faite du terme en position fonctionnelle n'a pas besoin de toute la complexité de `evaluate` et peut se suffire d'un mini-évaluateur pour expressions en position fonctionnelle qui ne saurait traiter que des noms de fonctions. Mettre en œuvre cette idée aboutit à modifier la clause finale de l'interprète précédent en :

```
...
(else (invoke (lookup (car e) env)
              (evlis (cdr e) env) )) ...
```

Il y a donc maintenant deux évaluateurs différents. À un identificateur peut correspondre plusieurs comportements suivant sa position, ici fonctionnelle ou paramétrique. Particulariser la position fonctionnelle s'accompagnerait volontiers de la présence d'un environnement supplémentaire uniquement dédié aux fonctions Rechercher la fonction liée à un nom serait a priori plus simple puisque cet environnement

serait débarrassé des variables normales. L'interprète de base peut donc être réécrit en tenant compte de ces indications : un environnement fonctionnel `fenv` et un évaluateur spécifique des formes, `evaluate-application` y sont clairement identifiés. On dispose donc maintenant de deux environnements d'où le nom de Lisp$_2$ [SG93]. On obtient :

```
(define (f.evaluate e env fenv)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e) (string? e) (char? e)
                 (boolean? e) (vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote)  (cadr e))
        ((if)     (if (f.evaluate (cadr e) env fenv)
                      (f.evaluate (caddr e) env fenv)
                      (f.evaluate (cadddr e) env fenv) ))
        ((begin)  (f.eprogn (cdr e) env fenv))
        ((set!)   (update! (cadr e)
                           env
                           (f.evaluate (caddr e) env fenv) ))
        ((lambda) (f.make-function (cadr e) (cddr e) env fenv))
        (else     (evaluate-application (car e)
                                        (f.evlis (cdr e) env fenv)
                                        env
                                        fenv )) ) ) )
```

L'évaluateur particulier des formes est `evaluate-application` qui reçoit le terme fonctionnel non évalué, les arguments évalués et les deux environnements courants. On observera que la création de fonction (`lambda`) enclôt les deux environnements `env` et `fenv` de manière à fixer les valeurs des variables libres apparaissant dans les corps des fonctions, qu'elles apparaissent en position fonctionnelle ou paramétrique. Autrement, cette nouvelle version ne diffère de la précédente que par le fait que `fenv` accompagne, comme son ombre, l'environnement des variables `env`. Les fonctions `eprogn` et `evlis` sont bien sûr mises à jour pour propager `fenv` et deviennent :

```
(define (f.evlis exps env fenv)
  (if (pair? exps)
      (cons (f.evaluate (car exps) env fenv)
            (f.evlis (cdr exps) env fenv) )
      '() ) )
(define (f.eprogn exps env fenv)
  (if (pair? exps)
      (if (pair? (cdr exps))
          (begin (f.evaluate (car exps) env fenv)
                 (f.eprogn (cdr exps) env fenv) )
          (f.evaluate (car exps) env fenv) )
      empty-begin ) )
```

Les fonctions, lorsqu'elles sont invoquées, lient leurs variables dans l'environnement des variables, mais seule la création des fonctions est modifiée ; leur invocation que réalise `invoke` ne l'est pas.

```
(define (f.make-function variables body env fenv)
  (lambda (values)
    (f.eprogn body (extend env variables values) fenv) ) )
```

L'évaluateur fonctionnel a pour tâche d'analyser le terme fonctionnel afin de préparer l'invocation finale. Si l'on retient la grammaire de COMMON LISP, alors ne peuvent intervenir en position fonctionnelle qu'un symbole ou une forme `lambda`.

```
(define (evaluate-application fn args env fenv)
  (cond ((symbol? fn)
         (invoke (lookup fn fenv) args) )
        ((and (pair? fn) (eq? (car fn) 'lambda))
         (f.eprogn (cddr fn)
                   (extend env (cadr fn) args)
                   fenv ) )
        (else (wrong "Incorrect functional term" fn)) ) )
```

Qu'a-t-on gagné ou perdu ? Un premier gain est que la recherche d'une fonction associée à un nom est beaucoup plus légère qu'avant car elle ne requiert plus qu'un simple appel à `lookup` et élimine donc un appel à `f.evaluate` suivi d'un décodage syntaxique déterminant que c'est une référence. Comme de plus l'environnement fonctionnel est débarrassé de toutes les variables, il est certainement moins encombré et les recherches peuvent y être plus rapides. Un second gain est que le traitement des formes où figure une forme `lambda` en position fonctionnelle est grandement amélioré. Considérez l'exemple (financier) suivant :

```
(let ((VAT 1.186))
  ((lambda (x) (* VAT x)) (read)) )
```

On voit sur cet exemple que la fermeture correspondant à `(lambda (x) (* VAT x))` n'est pas créée ; son corps est directement évalué dans le bon environnement.

L'ennui est que ces deux gains n'en sont pas vraiment puisqu'une analyse assez simple permet de les retrouver dans un Lisp_1. La vraie seule différence est de nature implémentatoire. Un Lisp_2 est un petit peu plus performant car l'on peut assurer que tout nom présent dans `fenv` est lié à une fonction. Il suffit, chaque fois qu'une liaison enrichit l'environnement fonctionnel, de vérifier que la valeur présente dans la liaison est bien une fonction. On peut donc ainsi ne jamais vérifier à l'emploi que c'est bien une fonction. Comme tout nom se doit d'appartenir à l'environnement fonctionnel initial, on le liera à une fonction appelant `wrong` si d'aventure elle est invoquée. En termes plus programmatiques et sachant que notre langage d'implantation est Scheme (car ce qui suit n'est pas légal en COMMON LISP du fait de la présence d'une fonction calculée dans la forme `((lookup fn fenv) args)`), la fonction `evaluate-application` est plus précisément :

```
(define (evaluate-application fn args env fenv)
  (cond ((symbol? fn) ((lookup fn fenv) args))
        ... ) )
```

On gagne donc le test `(procedure? fn)` présent dans `invoke`. Le nombre d'appels fonctionnels est tel, en Lisp, qu'un quelconque gain sur l'appel fonctionnel est toujours bon à prendre et peut influencer grandement les performances totales. Toutefois ce gain n'est pas très important car il ne joue que sur les seules applications fonctionnelles à *fonction calculée*, c'est-à-dire non connue statiquement et il y en a peu.

En revanche, du côté des pertes, on a justement perdu la possibilité de calculer la fonction à appliquer. L'expression `(if` *`condition`* `(+ 3 4) (* 3 4))` peut se factoriser en Scheme et par rapport aux arguments `3` et `4`, en `((if` *`condition`* `+ *) 3 4)`. La règle est simple et très algébrique, c'est quasiment une identité remarquable mais,

en Lisp$_2$, ce programme n'est pas licite puisque le premier terme n'est ni un symbole ni une forme `lambda`.

2.2.1 Évaluation du terme fonctionnel

Tel qu'ici introduit, l'environnement fonctionnel est loin de procurer les facilités de l'environnement paramétrique, en particulier, comme on l'a vu dans l'exemple précédent, il ne permet pas que soit calculée la fonction à appliquer. L'astuce ancienne, qui prévalait dans la lignée MacLisp, était d'enrichir l'évaluateur fonctionnel afin de sous-traiter à `f.evaluate` les expressions qu'il ne comprend pas. On a donc :

```
(define (evaluate-application2 fn args env fenv)
  (cond ((symbol? fn)
         ((lookup fn fenv) args) )
        ((and (pair? fn) (eq? (car fn) 'lambda))
         (f.eprogn (cddr fn)
                   (extend env (cadr fn) args)
                   fenv ) )
        (else (evaluate-application2                    ; ** Modified **
               (f.evaluate fn env fenv) args env fenv )) ) )
```

On peut donc résoudre notre problème et écrire :

`(if` *condition* `(+ 3 4) (* 3 4))` ≡ `((if` *condition* `'+ '*) 3 4)`

La transformation est loin d'être élégante puisque l'on a dû ajouter des apostrophes disgracieuses. Mais enfin, cela marche ! Cela marche : oui, mais trop bien puisque maintenant l'évaluateur peut boucler !

`('''''''''1789` *arguments*`)`

L'expression `'''''''''1789` est évaluée tant que des citations subsistent puis boucle sur le nombre 1789, toujours égal à lui-même et jamais à une fonction. La sous-traitance est donc un peu forte et nécessite un peu plus de contrôle. Une variante de ce problème a aussi existé lorsque `evaluate-application` ressemblait à :

```
(define (evaluate-application3 fn args env fenv)
  (cond
    ((symbol? fn)
     (let ((fun (lookup fn fenv)))
       (if fun (fun args)
           (evaluate-application3 (lookup fn env) args env fenv)) ) )
      ... ) )
```

Dans cette variante, tous les symboles ne sont pas prédéfinis dans l'environnement fonctionnel initial `fenv.global` et, lorsqu'un symbole n'est pas défini dans l'environnement fonctionnel, sa valeur est recherchée dans l'environnement des variables. Las ! Si l'on suppose qu'aucune fonction `foo` n'existe, on obtient encore des programmes pouvant boucler, sur la valeur d'une variable, comme :

```
(let ((foo 'foo))
   (foo arguments) )
```

Et l'on a beau ajouter, dans `evaluate-application`, la détection des variables qui ont pour valeur le symbole qui porte leur nom, on peut encore leurrer l'évaluateur fonctionnel et le faire boucler, un peu plus vicieusement, sur :

```
(let ((flip 'flop)
      (flop 'flip) )
   (flip) )
```

La seule solution propre est donc la première définition que nous avions donnée de l'évaluateur fonctionnel `evaluate-application` **[☞ p. 32]** et il nous faut rechercher une nouvelle méthode pour accepter que le terme fonctionnel soit calculé.

2.2.2 Dualité des mondes

Pour résumer les problèmes, il est des calculs appartenant au monde paramétrique que l'on voudrait accomplir dans le monde fonctionnel et vice versa. Plus précisément, on peut vouloir passer une fonction en argument ou en résultat ou encore vouloir que la fonction qui sera appliquée soit le résultat d'un calcul non immédiat. S'il est nécessaire d'indiquer une fonction dans le terme fonctionnel et si l'on veut calculer la fonction à appliquer, alors il suffit d'avoir une fonction prédéfinie sachant appliquer des fonctions. Introduisons donc la fonction `funcall` (pour *function call*) qui applique son premier argument (qui doit être une fonction) à ses arguments suivants. Tentons une première écriture :

```
(if condition (+ 3 4) (* 3 4))                                    WRONG
≡ (funcall (if condition + *) 3 4)
```

Les arguments, notamment le premier d'entre eux, sont évalués avec l'évaluateur normal `f.evaluate` ; la fonction `funcall` se saisit de tous et réalise l'application. On peut facilement définir `funcall` comme :

```
(lambda (args)
  (if (> (length args) 1)
      (invoke (car args) (cdr args))
      (wrong "Incorrect arity" 'funcall) ) )
```

La fonction `funcall` représente, en Lisp_2, l'appel calculé. Dans tous les autres cas, la fonction est connue et la vérification de fonctionnalité n'est plus de mise. On notera, dans la définition de `funcall`, l'appel à `invoke` qui effectue cette vérification contrairement à `evaluate-application` qui l'avait supprimée. La fonction `funcall` rappelle un peu `apply` ; toutes deux prennent une fonction en premier argument et des arguments après. La différence est que, dans une forme `funcall`, on connaît statiquement le nombre d'arguments qui seront fournis à la fonction finalement invoquée.

Il reste malheureusement encore un problème. Lorsque l'on écrivit `(if condition + *)`, on désirait obtenir en résultat la fonction d'addition ou de multiplication. Ce que l'on obtient maintenant, c'est la valeur de la variable `+` ou `*` ! En COMMON LISP, ces variables n'ont aucun rapport avec de quelconques opérations arithmétiques mais sont liées par le mécanisme d'interaction à la dernière expression lue et au dernier résultat retourné par la boucle fondamentale d'interaction (la boucle *toplevel*) !

L'introduction de `funcall` répondait au désir qu'une évaluation normale conduisît à un résultat devant se conduire comme une fonction. L'inverse existe qui est que l'on aimerait disposer, dans l'interprète normal, d'un résultat issu de l'évaluateur fonctionnel. On souhaite la valeur de la variable fonctionnelle `+`, telle que l'obtiendrait `evaluate-application`. On introduit donc, encore une fois, un nouveau dispositif linguistique : `function`. La forme spéciale `function` prend un nom de fonction et retourne sa valeur fonctionnelle. On peut ainsi jongler entre les deux espaces et écrire en toute sécurité :

```
(if condition (+ 3 4) (* 3 4)) ≡
        (funcall (if condition (function +) (function *)) 3 4)
```

Pour définir `function`, on ajoutera une clause supplémentaire à notre interprète `f.evaluate`. La définition de `function` qui va suivre n'a rien à voir avec celle de la page 20 qui définissait la syntaxe `(function (lambda` *`variables corps`*`))` tandis qu'ici nous définissons `(function` *`nom-de-fonction`*`)` pour convertir un nom de fonction en une valeur fonctionnelle.

```
...
((function)
 (cond ((symbol? (cadr e))
        (lookup (cadr e) fenv) )
       (else (wrong "Incorrect function" (cadr e))) ) ) ...
```

La définition de `function` pourrait être étendue, comme en COMMON LISP, au traitement des formes `(function (lambda ...))` semblablement à ce que nous avions vu en page 20. C'est superflu dans le langage que nous avons jusqu'ici défini car nous disposons directement de `lambda` pour ce faire. C'est indispensable en COMMON LISP car `lambda` n'est pas une forme spéciale mais une sorte de marqueur ou mot-clé syntaxique annonçant que ce qui suit est une définition de fonction. Une forme préfixée par `lambda` ne peut apparaître qu'en position fonctionnelle ou en argument de la forme spéciale `function`.

La fonction `funcall` permet d'injecter le résultat d'un calcul venant du monde paramétrique comme valeur du monde fonctionnel. Inversement, la forme spéciale `function` permet d'obtenir la valeur d'une variable du monde fonctionnel. Il y a un parallèle saisissant entre l'application fonctionnelle et la référence à une variable avec les opérateurs (fonction et forme spéciale) `funcall` et `function`. En bref, l'existence simultanée des deux mondes et l'impératif de nécessaire interaction entre eux, imposent la présence de telles passerelles.

Notez qu'il n'y a actuellement pas de modifications possibles de l'environnement fonctionnel ; aucune forme d'affectation existe. Cette propriété permet aux compilateurs d'*intégrer* (*inlining*) les appels fonctionnels de manière sémantiquement saine. C'est un des attraits des espaces multiples de noms que de pouvoir leur conférer des vertus spécifiques.

2.2.3 Jouer en Lisp$_2$

Afin de rendre notre description d'un Lisp$_2$ autonome, il nous faut indiquer quel est l'environnement fonctionnel global et comment lancer l'interprète `f.evaluate`. L'environnement fonctionnel global est codé de façon semblable à l'environnement des variables. Nous changerons juste la macro `defprimitive` pour qu'elle enrichisse celui-là et non celui-ci.

```
(define fenv.global '())
(define-syntax definitial-function
  (syntax-rules ()
    ((definitial-function name)
     (begin (set! fenv.global (cons (cons 'name 'void) fenv.global))
            'name ) )
    ((definitial-function name value)
     (begin (set! fenv.global (cons (cons 'name value) fenv.global))
            'name ) ) ) )
(define-syntax defprimitive
```

```
  (syntax-rules ()
   ((defprimitive name value arity)
    (definitial-function name
      (lambda (values)
        (if (= arity (length values))
            (apply value values)
            (wrong "Incorrect arity" (list 'name values)) ) ) ) ) ) )
(defprimitive car car 1)
(defprimitive cons cons 2)
```

Enfin l'on plongera dans ce monde grâce à :

```
(define ( un certain Lisp2 )
  (define (toplevel)
    (display (f.evaluate (read) env.global fenv.global))
    (toplevel) )
  (toplevel) )
```

2.2.4 Enrichir l'environnement fonctionnel

Les environnements, de quelque nature qu'ils soient, sont des instances d'un type abstrait. On attend d'un environnement qu'il contienne des liaisons, qu'on puisse y chercher la liaison associée à un nom mais aussi qu'on puisse l'étendre. Tout comme l'application fonctionnelle, ou la forme `let`, peut étendre l'environnement des variables, on peut vouloir disposer de fonctions locales et, pour cela, étendre localement l'environnement fonctionnel. Actuellement l'environnement fonctionnel est complètement figé et gagnerait donc à être étendu. Une nouvelle forme spéciale `flet` pour `f`*unctional* `let` sert ce dessein. Sa syntaxe est :

```
(flet ( (nom1 liste-de-variables1 corps1 )
        (nom2 liste-de-variables2 corps2 )
        ...
        (nomn liste-de-variablesn corpsn ) )
  expressions... )
```

Comme la forme `flet` ne sait créer que des fonctions locales, il n'est nul besoin d'indiquer le mot-clé `lambda` qui est implicite. La forme spéciale `flet` évalue les différentes formes `(lambda` *liste-de-variables*$_i$ *corps*$_i$`)` correspondant aux fonctions locales indiquées et les lie aux *nom*$_i$ dans l'environnement fonctionnel. Les *expressions...* formant le corps du `flet` sont évaluées dans cet environnement fonctionnel enrichi. Tous les *nom*$_i$ peuvent être employés en position fonctionnelle et peuvent être soumis à `function` si d'aventure la fermeture associée était nécessaire à un quelconque calcul.

Ajouter `flet` à `f.evaluate` est simple :

```
...
((flet)
 (f.eprogn
   (cddr e)
   env
   (extend fenv
           (map car (cadr e))
           (map (lambda (def)
                  (f.make-function (cadr def) (cddr def) env fenv) )
                (cadr e) ) ) )) ...
```

Grâce à `flet`, les possibilités de l'environnement fonctionnel s'accroissent et la clôture de `fenv` et `env` par la forme `lambda` s'explique. Considérez, par exemple :

```
(flet ((square (x) (* x x)))
  (lambda (x) (square (square x))) )
```

La valeur de cette expression est une fonction anonyme élevant un nombre à la puissance quatrième. La fermeture créée clôt la fonction locale `square` qui lui est utile pour ses calculs.

2.3 Autres extensions

Une fois individualisé l'évaluateur particulier des termes fonctionnels, de nouvelles variantes viennent immédiatement à l'esprit. Il a, par exemple, été proposé que les entiers se voient conférer une valeur fonctionnelle les assimilant à des accesseurs de listes. Ainsi :

```
(2 '(foo bar hux wok))  → hux
(-2 '(foo bar hux wok)) → (hux wok)
```

L'entier n est assimilé à `cad`^{n}`r` si positif et à `cd`^{n}`r` si négatif. Les accesseurs de base `car` et `cdr` sont respectivement `1` et `-1`. On peut alors imaginer réécrire algébriquement `(-1 (-2 π))` en `(-3 π)` et `(2 (-3 π))` en `(5 π)`.

Une autre variante revient à conférer un sens aux listes en position fonctionnelle sous réserve qu'elles soient des listes de fonctions. Ainsi :

```
((list + - *) 5 3)        → (8 2 15)
```

Appliquer une liste de fonctions revient à retourner la liste des valeurs que retournent chacune de ces fonctions sur ces mêmes arguments. L'extrait précédent est donc ni plus ni moins équivalent à :

```
(map (lambda (f) (f 5 3))
     (list + - *) )
```

Enfin, on peut même permettre que la fonction soit en seconde position afin de simuler une écriture infixe. Dans ce cas `(1 + 2)` retournerait `3`. DWIM (pour *Do What I Mean*) [Tei74, Tei76] savait récupérer ce genre d'erreurs.

Toutes ces innovations sont dangereuses car, diminuant le nombre de formes erronées, elles dissimulent l'occurrence d'erreurs qui seraient facilement détectées et corrigées. Elles ne conduisent pas non plus à des économies fantastiques de code et restent, somme toute, assez rares d'emploi. Elles suppriment aussi l'affinité qui régnait entre les fonctions et les objets fonctionnellement applicables, c'est-à-dire les objets qui pouvaient figurer en position fonctionnelle : une liste ou un nombre sont maintenant applicables sans pour autant être devenus des fonctions. On peut donc additionner, entre autres, des objets applicables sans encourir d'erreur :

```
(apply (list 2 (list 0 (+ 1 2)))
       '(foo bar hux wok) )
  → (hux (foo wok))
```

Nous recommandons donc de ne pas les incorporer dans un langage tel que Lisp.
[☞ Ex. 2.3]

2.4 Comparaison entre Lisp$_1$ et Lisp$_2$

Au terme de notre exploration des Lisp$_1$ et Lisp$_2$, exploration qui n'a pas été sans vagabondages, que peut-on retenir de ces deux philosophies ? Scheme est un Lisp$_1$, plaisant à programmer et à enseigner car le processus d'évaluation est simple et uniforme. Par comparaison, un Lisp$_2$ est plus difficile car la coexistence de deux mondes impose de bénéficier de formes permettant de transiter de l'un vers l'autre et réciproquement. COMMON LISP n'est pas simplement un Lisp$_2$: d'autres espaces de liaisons existent, comme l'environnement des échappements lexicaux, des étiquettes de `tagbody`, etc. C'est pourquoi l'on parle, parfois, de Lisp$_n$ car à un même nom peuvent être associés de multiples comportements dépendant de sa position syntaxique. Les langages à syntaxe forte, voire pesante, ont souvent de multiples espaces de noms ou environnements (de variables, de fonctions, de types). Ces environnements multiples ont des propriétés adaptées : il n'y a, par exemple, pas de modification possible (c'est-à-dire pas d'affectation) dans l'environnement fonctionnel local, il est ainsi simple d'optimiser l'appel aux fonctions locales.

Un programme écrit en Lisp$_2$ sépare bien le monde des fonctions du reste des calculs. C'est une différentiation payante que tous les bons compilateurs Scheme font [Sén89]. Ils réécrivent, en interne, les programmes Scheme en une sorte de Lisp$_2$ qu'ils peuvent après mieux compiler. Ils font notamment apparaître tous les endroits où il faut insérer des `funcall`, c'est-à-dire des appels calculés. Un utilisateur de Lisp$_2$ mâche donc le travail du compilateur et, partant, appréhende mieux ce qu'il lui livre en pâture.

Nous avons tant donné de variantes dans ce qui précède qu'il nous semble utile de fournir la définition d'un Lisp$_2$, inspiré de COMMON LISP, la plus simple possible. La seule modification que nous y avons incorporée est l'introduction de la fonction `f.lookup` pour chercher un nom dans l'environnement fonctionnel. Si le nom n'est pas trouvé, une fonction appelant `wrong` est retournée. Cet artifice permet d'assurer que `f.lookup` retourne toujours en temps fini une fonction. D'un autre côté, elle introduit un mécanisme d'erreur différé car l'erreur n'est pas de référencer une fonction inexistante mais de l'utiliser ce qui peut survenir plus tard ou même jamais.

```
(define (f.evaluate e env fenv)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e) (string? e) (char? e)
                 (boolean? e) (vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote)  (cadr e))
        ((if)     (if (f.evaluate (cadr e) env fenv)
                      (f.evaluate (caddr e) env fenv)
                      (f.evaluate (cadddr e) env fenv) ))
        ((begin)  (f.eprogn (cdr e) env fenv))
        ((set!)   (update! (cadr e)
                           env
                           (f.evaluate (caddr e) env fenv) ))
        ((lambda) (f.make-function (cadr e) (cddr e) env fenv))
        ((function)
```

```
 (cond ((symbol? (cadr e))
        (f.lookup (cadr e) fenv) )
       ((and (pair? (cadr e)) (eq? (car (cadr e)) 'lambda))
        (f.make-function
         (cadr (cadr e)) (cddr (cadr e)) env fenv ) )
       (else (wrong "Incorrect function" (cadr e))) ) )
((flet)
 (f.eprogn (cddr e)
           env
           (extend fenv
                   (map car (cadr e))
                   (map (lambda (def)
                          (f.make-function (cadr def)
                                           (cddr def)
                                           env fenv ) )
                        (cadr e) ) ) ) )
((labels)
 (let ((new-fenv (extend fenv
                         (map car (cadr e))
                         (map (lambda (def) 'void)
                              (cadr e) ) )))
   (for-each (lambda (def)
               (update! (car def)
                        new-fenv
                        (f.make-function (cadr def)
                                         (cddr def)
                                         env new-fenv ) ) )
             (cadr e) )
   (f.eprogn (cddr e) env new-fenv ) ) )
       (else     (f.evaluate-application (car e)
                                         (f.evlis (cdr e) env fenv)
                                         env
                                         fenv )) ) ) )
(define (f.evaluate-application fn args env fenv)
  (cond ((symbol? fn)
         ((f.lookup fn fenv) args) )
        ((and (pair? fn) (eq? (car fn) 'lambda))
         (f.eprogn (cddr fn)
                   (extend env (cadr fn) args)
                   fenv ) )
        (else (wrong "Incorrect functional term" fn)) ) )
(define (f.lookup id fenv)
  (if (pair? fenv)
      (if (eq? (caar fenv) id)
          (cdar fenv)
          (f.lookup id (cdr fenv)) )
      (lambda (values)
        (wrong "No such functional binding" id) ) ) )
```

D'autres considérations plus pragmatiques sont liées à la lisibilité comparées des programmes écrits en Lisp$_1$ ou Lisp$_2$, [GP88]. Il est certain que les lispiens chevronnés éviteront d'écrire :

```
(defun foo (list)
  (list list) )
```

Vu d'un Lisp$_1$, `(list list)` est une auto-application licite[2] dont le sens est très

2. Les auto-applications qui ont du sens ne sont pas nombreuses mais existent bel et bien : `(number?`

différent de celui que lui accorde un $Lisp_2$ comme COMMON LISP. En COMMON LISP, ces deux noms sont évaluées dans deux environnements différents et ne sont donc nullement en conflit. Il est cependant de bon ton d'éviter de nommer ses variables locales de noms de fonctions globales bien connues : les faiseurs de macros vous en remercient et vos programmes seront moins dépendants du Lisp ou du Scheme que vous utilisez.

Une autre différence concerne justement les macros. Une macro s'expansant en une forme `lambda`, par exemple pour implanter un système à objets, est problématique en COMMON LISP du fait des restrictions grammaticales que COMMON LISP impose en ce qui concerne les endroits où peuvent survenir les `lambda`. Une forme `lambda` ne peut apparaître qu'en position fonctionnelle, ce qui rend erronée l'expression suivante : `(...(lambda...)...)`. Une forme `lambda` peut aussi apparaître dans la forme spéciale `function` mais cette forme ne peut apparaître qu'en position paramétrique : `((function (lambda...)) ...)`est donc aussi erroné. Une macro qui s'expanse ne sait dans quel contexte fonctionnel ou paramétrique elle s'insérera ; partant, elle ne peut être écrite sans induire une transformation de programmes plus complexe et plus globale. Pour l'exemple sus-cité d'un système à objets, on pourra adopter une expansion vers `(function (lambda ...))` et ajouter `funcall` en tête de toutes les formes susceptibles de contenir un objet en position fonctionnelle.

Mentionnons enfin un point moyen qu'adoptent certains langages. On peut limiter les risques de confusion, même avec de multiples espaces de valeurs, en interdisant qu'un même nom figure sur plusieurs espaces simultanément. Dans le précédent exemple, la variable `list` serait interdite d'emploi car figurant déjà dans l'environnement fonctionnel global. Un autre exemple est l'interdiction, présente dans presque tous les systèmes Lisp ou Scheme, qu'un nom soit à la fois le nom d'une fonction et d'une macro. Cette limitation rend certainement la vie moins infernale.

2.5 Espaces de noms

Un *environnement* associe des entités à des noms. On a précédemment vu deux sortes d'environnements : l'environnement normal `env` et un environnement fonctionnel `fenv` associant à des noms des fonctions. Avoir séparé ces deux espaces de valeurs avait eu l'intérêt d'améliorer le traitement des appels fonctionnels et de distinguer clairement le monde des fonctions de celui des variables. Inversement, cela introduit deux évaluateurs différents ainsi que des moyens de passage d'un monde à l'autre et, par là même, complique la sémantique du langage. La discussion sur les variables dynamiques avait suggéré que, dans les dialectes récents de Lisp (Ilog Talk, EULISP, IS-Lisp), les variables dynamiques étaient placées dans un espace de noms séparé. C'est cette variante que nous allons ici explorer et qui illustrera la notion d'*espace de noms*.

Un environnement est une sorte de type abstrait ; un environnement contient des *liaisons* entre des noms et des *entités* référençables par ces noms. Ces entités peuvent être soit des valeurs, c'est-à-dire des objets que l'utilisateur peut manipuler comme

---

`number?)` en est une autre.

tout autre objet de première classe, soit de réelles entités, c'est-à-dire des objets de seconde classe que l'on ne peut manipuler que par l'intermédiaire de leur nom et généralement seulement à travers un jeu approprié de syntaxes ou de formes spéciales. Pour l'instant, vous ne connaissez d'entités que les liaisons ; ces êtres existent puisque capturables au sein de fermetures ! Les qualités de ces liaisons feront l'objet d'autres commentaires lorsque nous étudierons les effets de bord.

On peut rechercher si un nom apparaît dans un environnement, rechercher l'entité qui lui est associée ou modifier cette association. On peut aussi étendre un environnement avec de nouvelles associations (ou liaisons) que ce soit l'environnement courant, local ou global. Bien sûr, toutes ces opérations ne doivent pas nécessairement être procurées et de nombreux environnements n'ont d'intérêt que parce qu'ils limitent ces opérations. Le tableau suivant recense les qualités de l'environnement des variables de Scheme :

Référence	*x*
Valeur	*x*
Modification	`(set! x ...)`
Extension	`(lambda (...x ...) ...)`
Définition	`(define x ...)`

Comme nous utiliserons assez souvent cette présentation pour discuter des propriétés des environnements, paraphrasons quelque peu ce tableau. La première ligne indique la syntaxe avec laquelle on référence une variable pour la clore dans une fermeture. La seconde ligne correspond à la syntaxe qui permet d'obtenir la valeur d'une variable. Dans le cas des variables, ces deux syntaxes sont les mêmes mais ce n'est pas toujours le cas. La troisième ligne montre comment la liaison associée à la variable peut être modifiée. La quatrième ligne exhibe la façon d'étendre l'environnement des variables lexicales, par une forme `lambda` (ou bien sûr des macros comme `let` ou `let*` qui s'expansent en `lambda`). Enfin dans la dernière ligne figure comment définir une liaison globale. Ces multiples cas peuvent paraître spécieux et s'apparenter à d'occultes pratiques ayant des répercussions quant à la vertu des mouches, toutefois les prochains tableaux de ce chapitre et du suivant devraient rendre claires les différentes intentions qui contrôlent l'emploi des variables.

Dans le Lisp_2, esquissé en début de chapitre, l'espace des fonctions pouvait se décrire comme :

Référence	`(f ...)`
Valeur	`(function f)`
Modification	pas de ça ici !
Extension	`(flet (...(f ...) ...) ...)`
Définition	non traité précédemment (cf. `defun`)

2.5.1 Variables dynamiques

Les *variables dynamiques* correspondent à un concept si différent des variables lexicales que nous nous proposons de fortement les différencier. Le tableau suivant décrit les qualités que nous souhaitons pour ce nouvel environnement :

Référence	non capturable
Valeur	`(dynamic d)`
Modification	`(dynamic-set! d ...)`
Extension	`(dynamic-let ( ...(d ...) ...) ...)`
Définition	non traité ici.

Ce nouvel espace de noms rend compte d'un certain nombres de faits : les variables dynamiques [3] peuvent être liées localement par `dynamic-let`, de syntaxe similaire à `let` (et `flet`), on peut obtenir leur valeur par `dynamic` et les modifier par `dynamic-set!`. Ces trois formes sont des formes spéciales encore que l'on verra plus loin une autre mise en œuvre. Nous allons donc reprendre l'interprète `f.evaluate` et lui adjoindre un nouvel environnement `denv` qui ne contiendra que les seules variables dynamiques. Ce nouvel interprète, un Lisp_3, utilisera des fonctions dont le nom sera préfixé par `df.` afin d'éviter les confusions. Voici donc cet évaluateur (duquel nous avons retiré `flet` pour ne pas le surcharger) :

```
(define (df.evaluate e env fenv denv)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e) (string? e) (char? e)
                 (boolean? e) (vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote) (cadr e))
        ((if) (if (df.evaluate (cadr e) env fenv denv)
                  (df.evaluate (caddr e) env fenv denv)
                  (df.evaluate (cadddr e) env fenv denv) ))
        ((begin) (df.eprogn (cdr e) env fenv denv))
        ((set!) (update! (cadr e)
                         env
                         (df.evaluate (caddr e) env fenv denv) ))
        ((function)
         (cond ((symbol? (cadr e))
                (f.lookup (cadr e) fenv) )
               ((and (pair? (cadr e)) (eq? (car (cadr e)) 'lambda))
                (df.make-function
                 (cadr (cadr e)) (cddr (cadr e)) env fenv ) )
               (else (wrong "Incorrect function" (cadr e))) ) )
        ((dynamic) (lookup (cadr e) denv))
        ((dynamic-set!)
         (update! (cadr e)
                  denv
                  (df.evaluate (caddr e) env fenv denv) ) )
        ((dynamic-let)
         (df.eprogn (cddr e)
                    env
                    fenv
                    (extend denv
                            (map car (cadr e))
                            (map (lambda (e)
                                   (df.evaluate e env fenv denv) )
                                 (map cadr (cadr e)) ) ) ) ) )
```

3. Les variables dynamiques portent mal leur nom puisque, en un sens, ce ne sont pas une spécialisation des variables (lexicales).

```
        (else (df.evaluate-application
               (car e)
               (df.evlis (cdr e) env fenv denv)
               env
               fenv
               denv )) ) ) )
(define (df.evaluate-application fn args env fenv denv)
  (cond ((symbol? fn) ((f.lookup fn fenv) args denv) )
        ((and (pair? fn) (eq? (car fn) 'lambda))
         (df.eprogn (cddr fn)
                    (extend env (cadr fn) args)
                    fenv
                    denv ) )
        (else (wrong "Incorrect functional term" fn)) ) )
(define (df.make-function variables body env fenv)
  (lambda (values denv)
    (df.eprogn body (extend env variables values) fenv denv) ) )
(define (df.eprogn e* env fenv denv)
  (if (pair? e*)
      (if (pair? (cdr e*))
          (begin (df.evaluate (car e*) env fenv denv)
                 (df.eprogn (cdr e*) env fenv denv) )
          (df.evaluate (car e*) env fenv denv)  )
      empty-begin ) )
```

L'introduction d'un nouvel environnement `denv` conduit à augmenter la signature de `df.evaluate` ainsi que celle de `df.eprogn` pour le prendre en compte. Par ailleurs `df.evaluate` introduit trois nouvelles formes spéciales pour la manipulation de `denv`, l'*environnement dynamique*. Une modification plus subtile apparaît dans `df.evaluate-application` : une fonction est maintenant appliquée non seulement à ses arguments mais aussi à l'environnement dynamique courant. Cette situation avait déjà été vue **[☞ p. 18]** lorsqu'il avait fallu passer l'environnement courant à la fonction invoquée. Lors d'une application fonctionnelle, de multiples environnements sont en jeu : l'environnement des variables et des fonctions qui avaient été capturées à la définition de la fonction, enfin l'environnement des variables dynamiques courant au moment de l'application. Ce dernier environnement n'est donc pas capturable (encore qu'il le soit, pour certaines implantations de Scheme, lors de la création de *thread*) et toute reférence à une variable dynamique entraîne une recherche de sa valeur dans l'environnement dynamique courant. Cette recherche peut être infructueuse auquel cas nous signalons une erreur. D'autres choix seraient possibles comme d'avoir un environnement dynamique global unique (comme en IS-Lisp), de multiples environnements globaux, en fait un par module, comme en EuLisp ou encore l'environnement global des variables lexicales comme en Common Lisp. **[☞ p. 45]**

Un des intérêts de ce nouvel espace est la claire vision de ce qui est dynamique et de ce qui ne l'est point. Tout attouchement pratiqué sur l'environnement dynamique se singularise par une forme spéciale de nom préfixé par `dynamic`. C'est donc une excellente annotation qui attire l'œil et permet de juger immédiatement de l'interface d'une fonction. En effet et dans ce Lisp_3, le comportement d'une fonction est stipulé non seulement par la valeur de ses variables mais aussi par l'environnement dynamique. Parmi les emplois usuels de l'environnement dynamique, le plus important concerne le traitement des erreurs. Lorsqu'une erreur (ou situation exceptionnelle) survient dans

un calcul, un objet décrivant l'exception est formé sur lequel on applique la fonction courante de traitement (et éventuellement de récupération) des exceptions. Ce pourrait être une fonction globale mais qui alors requerrait d'odieuses affectations pour être particularisée le temps d'un calcul. L'étanchéité des environnement lexicaux ne semble guère propice à la spécification de fonctions de traitement d'erreur. Par contre, un calcul a une durée de vie qu'enserre parfaitement une forme `let` ou `dynamic-let`. Cette dernière forme a pour avantage que *(i)* les liaisons qu'elle introduit ne sont pas capturables ; *(ii)* qu'elles ne sont accessibles que pendant la durée de vie du calcul de son corps ; *(iii)* qu'elles sont défaites automatiquement à la fin du calcul. C'est donc le concept idéal permettant d'instaurer temporairement une fonction de traitement d'erreur.

Un autre exemple de variable dynamique est le suivant. Les fonctions d'impression de COMMON LISP sont gouvernées par différentes variables dynamiques telles que `*print-base*`, `*print-circle*`, etc., qui spécifient la base de numération selon laquelle les nombres doivent être imprimés, si la donnée à imprimer comporte des cycles, etc. Certes il serait possible que chaque fonction d'impression prenne toutes ces spécifications en arguments : au lieu de seulement écrire `(print` *`expression`*`)`, on écrirait à chaque fois `(print` *`expression print-escape print-radix print-base print-circle print-pretty print-level print-length print-case print-array print-gensym`*`)`. Les variables dynamiques sont donc une technique pour paramétrer des calculs et éviter de fastidieuses énumérations de paramètres qui, le plus souvent, ont une valeur par défaut acceptable.

Scheme, sans trop le dire, use d'un mécanisme similaire pour la spécification des flux d'entrée et de sortie. On peut écrire `(display` *`expression`*`)` ou `(display` *`expression flux`*`)` [4]. La première des formes imprime la valeur d'*expression* sur le flux de sortie courant tandis que la seconde précise explicitement quel flux de sortie utiliser. La fonction `with-output-to-file` permet de spécifier le flux de sortie courant pendant toute la durée d'un calcul. Le flux courant peut être connu grâce à la fonction `(current-output-port)`. On pourra ainsi écrire une fonction d'impression de listes cycliques indépendante du flux d'impression [5] :

```
(define (display-cyclic-spine list)
  (define (scan l1 l2 flip)
    (cond ((atom? l1)  (unless (null? l1) (display " . ")
                               (display l1) )
                       (display ")") )
          ((eq? l1 l2) (display "...)") )
          (else        (display (car l1))
                       (when (pair? (cdr l1)) (display " "))
                       (scan (cdr l1)
                             (if (and flip (pair? l2)) (cdr l2) l2)
                             (not flip) ) ) ) )
  (display "(")
  (scan list (cons 123 list) #f) )
(display-cyclic-spine                          ; prints (1 2 3 4 1 ...)
```

4. Un des principes libertaires de Lisp est « permets à autrui ce que toi-même tu t'autorises ». On peut remarquer que la fonction `display` ne peut prendre qu'un ou deux arguments, ce qu'aucun mécanisme linguistique ne permet de réaliser en Scheme ; Lisp a en revanche la notion de variables optionnelles.

5. Voir aussi la fonction `list-length` de COMMON LISP.

```
(let ((l (list 1 2 3 4)))
  (set-cdr! (cdddr l) l)
  l ) )
```

Si l'on ressort notre petit tableau et qu'on l'adapte à l'espace des flux de sorties de Scheme l'on obtient :

Référence	référencé chaque fois que non mentionné dans une fonction d'impression !
Valeur	`(current-output-port)`
Modification	non modifiable
Extension	`(with-output-to-file` *`file-name thunk`*`)`
Définition	— non applicable —

En COMMON LISP, le mécanisme use explicitement de variables dynamiques. Le flux de sortie qu'utilisent par défaut les fonctions d'impression `print`, `write`, etc. est la valeur de la variable dynamique `*standard-output*` [6]. On peut donc simuler `with-output-to-file` par [7]

```
(define (with-output-to-file filename thunk)
  (dynamic-let ((*standard-output* (open-input-file filename)))
     (thunk) ) )
```

2.5.2 Variables dynamiques en COMMON LISP

Si COMMON LISP procure séparément les concepts de variables lexicales et dynamiques, il tente cependant de les unifier syntaxiquement. La forme `dynamic-let` n'existe pas mais peut-être ainsi simulée :

```
(dynamic-let ((x α))  ≡ (let ((x α))
   β )                    (declare (special x))
                          β )
```

Mais l'innovation est que l'obtention, dans β, de la valeur de la variable dynamique `x` ne passe pas par la forme `dynamic` mais s'écrit tout simplement `x` ! La raison est que la déclaration `(declare (special x))` stipule à la fois que la liaison que doit établir `let` doit être dynamique et que toute référence au nom `x` dans le corps du `let` doit être considérée comme étant équivalente à ce que nous avions nommé `(dynamic x)`.

Ce schéma a l'inconvénient de ne plus permettre de référencer la variable lexicale nommée `x` au sein de β mais seulement la variable dynamique homonyme. Inversement, on peut référencer la variable dynamique `x` en tout contexte, en écrivant `(locally (declare (special x)) x)`, ce qui correspond à notre forme spéciale `dynamic`.

Le schéma de COMMON LISP spécifie donc lexicalement la nature d'une référence. Nous pouvons expliciter ce mécanisme en modifiant notre précédent interprète :

```
(define (df.evaluate e env fenv denv)
  (if (atom? e)
```

---

6. L'usage prévaut d'entourer d'étoiles le nom des variables dynamiques. Ce n'est qu'une convention permettant d'attirer l'œil sur elles.

7. Ce n'est toutefois ni du COMMON LISP ni de l'IS-Lisp mais du Lisp_3 tel que défini un peu plus haut.

```
(cond ((symbol? e) (cl.lookup e env denv))
      ((or (number? e) (string? e) (char? e)
           (boolean? e) (vector? e) )
       e )
      (else (wrong "Cannot evaluate" e)) )
(case (car e)
  ((quote) (cadr e))
  ((if) (if (df.evaluate (cadr e) env fenv denv)
            (df.evaluate (caddr e) env fenv denv)
            (df.evaluate (cadddr e) env fenv denv) ))
  ((begin) (df.eprogn (cdr e) env fenv denv))
  ((set!) (cl.update! (cadr e)
                      env
                      denv
                      (df.evaluate (caddr e) env fenv denv) ))
  ((function)
   (cond ((symbol? (cadr e))
          (f.lookup (cadr e) fenv) )
         ((and (pair? (cadr e)) (eq? (car (cadr e)) 'lambda))
          (df.make-function
           (cadr (cadr e)) (cddr (cadr e)) env fenv ) )
         (else (wrong "Incorrect function" (cadr e))) ) )
  ((dynamic) (lookup (cadr e) denv))
  ((dynamic-let)
   (df.eprogn (cddr e)
              (special-extend env                    ;** Modified **
                              (map car (cadr e)) )
              fenv
              (extend denv
                      (map car (cadr e))
                      (map (lambda (e)
                             (df.evaluate e env fenv denv) )
                           (map cadr (cadr e)) ) ) ) )
  (else (df.evaluate-application
         (car e)
         (df.evlis (cdr e) env fenv denv)
         env
         fenv
         denv )) ) ) )
(define (special-extend env variables)
  (append variables env) )
(define (cl.lookup var env denv)
  (let look ((env env))
    (if (pair? env)
        (if (pair? (car env))
            (if (eq? (caar env) var)
                (cdar env)
                (look (cdr env)) )
            (if (eq? (car env) var)
                ;; lookup in the current dynamic environment
                (let lookup-in-denv ((denv denv))
                  (if (pair? denv)
                      (if (eq? (caar denv) var)
                          (cdar denv)
                          (lookup-in-denv (cdr denv)) )
                      ;; default to the global lexical environment
                      (lookup var env.global) ) )
```

```
                (look (cdr env)) ) )
        (wrong "No such binding" var) ) ) )
```

Le mécanisme s'explique comme suit : lorsque des variables dynamiques sont liées par `dynamic-let`, on les lie normalement dans l'environnement dynamique mais on les marque aussi comme étant dynamiques dans l'environnement lexical. On code cette information en empilant leur nom seul dans l'environnement lexical. Le processus qui associe à une référence sa valeur est donc modifié pour premièrement parachever l'analyse syntaxique afin de déterminer la nature de la référence : lexicale ou dynamique, puis on cherche la valeur associée dans l'environnement concerné. De plus, si l'on ne trouve pas de valeur dynamique, alors on se rabat sur l'environnement lexical global qui sert aussi d'environnement global dynamique en COMMON LISP.

Pour donner un bref exemple de ce Lisp_3 imitant quelque peu COMMON LISP, on pourra évaluer :

```
(dynamic-let ((x 2))
   (+ x                              ; dynamic
      (let ((x (+                    ; lexical
                  x x )))            ; dynamic
        (+ x                         ; lexical
           (dynamic x) ) ) ) )       ; dynamic
→ 8
```

2.5.3 Variables dynamiques sans forme spéciale

Le mécanisme de variables dynamiques que nous avons présenté usait de trois formes spéciales. La coquetterie de Scheme étant de limiter leur nombre, on peut songer à d'autres mécanismes. Sans avoir la prétention de brosser tous ceux qui sont possibles et qui sont notamment étudiés pour Scheme, nous pouvons proposer le suivant qui n'utilise que deux fonctions. La première fonction permet d'associer deux valeurs ; la seconde fonction permet de retrouver la seconde de ces valeurs lorsqu'on lui fournit la première d'entre elles. Nous utiliserons des symboles pour nommer les variables dynamiques. Enfin, si l'on souhaite modifier ces associations, il faudra passer par une donnée modifiable comme, par exemple, une paire pointée. On retrouve ainsi l'austérité chère à Scheme.

Nous expliciterons cette variante en introduisant un nouvel interprète avec deux environnements `env` et `denv`. Il correspond au précédent sauf que nous y avons enlevé toutes les formes spéciales superflues, l'espace des fonctions et les références aux variables à la COMMON LISP. Il ne reste plus donc que l'essence de l'environnement dynamique qui ne semble apparemment plus guère utile puisque celui-ci n'est modifié nulle part. Il est par contre toujours fourni aux fonctions ce qui va nous permettre d'achever nos desseins. Pour toujours différencier cette variante des autres, nous préfixerons les fonctions modifiées par `dd`.

```
(define (dd.evaluate e env denv)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e) (string? e) (char? e)
                 (boolean? e) (vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
```

```
(case (car e)
  ((quote) (cadr e))
  ((if) (if (dd.evaluate (cadr e) env denv)
            (dd.evaluate (caddr e) env denv)
            (dd.evaluate (cadddr e) env denv) ))
  ((begin) (dd.eprogn (cdr e) env denv))
  ((set!) (update! (cadr e)
                   env
                   (dd.evaluate (caddr e) env denv)))
  ((lambda) (dd.make-function (cadr e) (cddr e) env))
  (else (invoke (dd.evaluate (car e) env denv)
                (dd.evlis (cdr e) env denv)
                denv )) ) ) )
(define (dd.make-function variables body env)
  (lambda (values denv)
    (dd.eprogn body (extend env variables values) denv) ) )
(define (dd.evlis e* env denv)
  (if (pair? e*)
      (if (pair? (cdr e*))
          (cons (dd.evaluate (car e*) env denv)
                (dd.evlis (cdr e*) env denv) )
          (list (dd.evaluate (car e*) env denv)) )
      '() ) )
(define (dd.eprogn e* env denv)
  (if (pair? e*)
      (if (pair? (cdr e*))
          (begin (dd.evaluate (car e*) env denv)
                 (dd.eprogn (cdr e*) env denv) )
          (dd.evaluate (car e*) env denv) )
      empty-begin ) )
```

Nommons la première des deux fonctions que nous souhaitons introduire `bind-with-dynamic-extent` ou, sans particule, `bind/de`. Elle prend en première valeur une clef `tag`, une seconde qui lui sera associée `value` et enfin un calcul représenté par une suspension ou `thunk`. La fonction `bind/de` invoque la suspension après avoir enrichi l'environnement dynamique.

```
(definitial bind/de
  (lambda (values denv)
    (if (= 3 (length values))
        (let ((tag (car values))
              (value (cadr values))
              (thunk (caddr values)) )
          (invoke thunk '() (extend denv (list tag) (list value))) )
        (wrong "Incorrect arity" 'bind/de) ) ) )
```

La seconde fonction exploite l'environnement dynamique. Comme il faut prévoir qu'il se pourrait que l'on ne trouve rien, la fonction `assoc/de` prend en premier argument une clef et en second argument, une fonction qu'elle invoquera sur la clef si celle-ci n'est pas présente dans l'environnement dynamique :

```
(definitial assoc/de
  (lambda (values current.denv)
    (if (= 2 (length values))
        (let ((tag     (car values))
              (default (cadr values)) )
          (let look ((denv current.denv))
            (if (pair? denv)
```

```
                (if (eqv? tag (caar denv))
                    (cdar denv)
                    (look (cdr denv)) )
                (invoke default (list tag) current.denv) ) ) )
        (wrong "Incorrect arity" 'assoc/de) ) ) )
```

De nombreuses variantes sont possibles suivant que l'on emploie `eqv?` ou `equal?` comme comparateurs. **[☞ Ex. 2.4]**

Pour reprendre un exemple précédent, on évaluera :

```
(bind/de 'x 2
  (lambda () (+ (assoc/de 'x error)
                (let ((x (+ (assoc/de 'x error)
                            (assoc/de 'x error) )))
                  (+ x (assoc/de 'x error)) ) )) )
→ 8
```

Et si l'on veut associer des données changeantes à `x`, on pourra écrire :

```
(bind/de 'x (list 2)
  (lambda () (set-car! (assoc/de 'x error) 3)
             (car (assoc/de 'x error)) ) )
→ 3
```

On a ainsi montré qu'il n'est pas besoin de formes spéciales pour procurer l'équivalent des variables dynamiques, avec de plus un certain gain puisqu'ici l'on peut associer n'importe quoi à n'importe quoi. Ce gain peut aussi se muer en inconvénient car il existe des techniques efficaces d'implantation de variables dynamiques (en l'absence de parallélisme) par exemple la liaison superficielle qui exige que les valeurs possibles de la clef soient des symboles [8]. Au profit de cette variante l'on peut aussi compter sa transparence : que l'accès à une variable dynamique soit coûteux est clair car ce sont des appels à des fonctions spécialisées ; on est loin de la fusion qu'introduisait COMMON LISP.

Au chapitre des inconvénients vient la syntaxe qui nécessite d'utiliser `bind/de`, une suspension et `assoc/de` pour manipuler ces associations ; de judicieuses macros peuvent toutefois masquer ces problèmes. Le second inconvénient vient de ce que pour compiler correctement ces fonctions, le compilateur doit les connaître intimement, savoir leur comportement et leur propriétés. Il est vrai que la rareté des accès à ce type de variables ainsi que la présence de cette interface fonctionnelle permettent au compilateur de ne pas se casser la tête.

2.5.4 Conclusions

Au terme de cette digression sur les variables dynamiques, il est important de retenir la notion d'espace de noms qui correspond de fait à un environnement spécialisé dans la manipulation d'un certain type d'objets. On aura vu un Lisp_3 à l'œuvre et un peu plus précisément la méthode qu'utilise COMMON LISP pour ces variables dynamiques.

Toutefois un nouveau problème est soulevé par la dernière variante qui procurait la même fonctionnalité mais incarnée par deux fonctions. Si c'est un Lisp_n, de quel n s'agit-il ? Nous sommes parti de Scheme et, dans l'implantation que nous avons

8. De nombreux Lisp interdisent que soient utilisés comme noms de variables des mots réservés comme `nil` ou `if`.

donnée, il est clair que deux environnements sont en jeu `env` et `denv`. Par contre, ces fonctionnalités étant offertes par des fonctions, il n'y a qu'une seule règle d'évaluation uniforme : c'est toujours du Scheme donc un Lisp_1. Mais ce n'est pas si simple parce que les fonctions `bind/de` et `assoc/de` ont nécessité pour être définies de modifier profondément la définition (comparer `evaluate` et `dd.evaluate`). Nous assistons là à la reconnaissance de l'existence de fonctions primitives qu'il ne serait pas possible de recréer en Scheme pur si elles n'existaient pas et dont la seule présence dans la bibliothèque des fonctions disponibles modifie grandement la sémantique du langage. On verra dans un prochain chapitre que c'est aussi le cas de la fonction `call/cc`.

En définitive, nous avons toujours affaire à un Lisp_1 si l'on comptabilise les règles d'évaluation, mais c'est un Lisp_2 si l'on se focalise sur le nombre des espaces de noms. Une généralisation de ce fait est que certains prétendent que la présence de listes de propriétés est le signe d'un Lisp_n avec n non borné ! S'agissant d'un espace de noms global mis en œuvre par des fonctions non primitives, **[☞ Ex. 2.6]** nous récuserons cette appellation.

Nous pouvons tirer une seconde morale de cette étude des variables lexicales et dynamiques. COMMON LISP essayait d'unifier l'accès à deux espaces différents sous une même syntaxe et devait donc énoncer les règles permettant de déterminer pour une référence l'espace souhaité. Toujours en COMMON LISP, l'espace dynamique global se confond avec l'espace lexical global. En Lisp 1.5, existait la notion de constante définie par la forme spéciale `csetq` où le `c` signifiait constante.

Référence	*x*
Valeur	*x*
Modification	`(csetq` *x forme*`)`
Extension	non possible
Définition	`(csetq` *x forme*`)`

L'introduction des constantes rend la syntaxe ambiguë. Lorsque l'on écrit `foo`, ce peut être une constante ou une variable. La règle était la suivante : si `foo` est une constante, alors on retourne sa valeur sinon on va chercher la valeur de la variable lexicale de même nom. Mais les constantes peuvent être modifiées (*sic*) par la forme même qui aussi les crée : `csetq`. Les constantes s'apparentent donc aux variables globales de Scheme à ceci près que la préséance est inverse puisqu'en Scheme la valeur lexicale est d'abord considérée et seulement à défaut sa valeur globale.

Le problème est donc général. Lorsque plusieurs espaces différents sont référençables par une syntaxe ambiguë, des règles doivent être énoncées afin de lever cette ambiguïté.

2.6 Récursion

La récursion est essentielle en Lisp, pourtant rien de ce qui précède n'a expliqué comment elle était mise en œuvre. Nous allons analyser successivement les différents types de récursion et les problèmes variés qu'elle soulève.

2.6.1 Récursion simple

La plus connue des fonctions simplement récursives est probablement la factorielle, rappelée ci-dessous :

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1))) ) )
```

Le langage décrit au précédent chapitre ignore le sens de `define`, supposons, pour l'instant, que c'est une macro s'expansant en :

```
(set! fact (lambda (n)
             (if (= n 1) 1
                 (* n (fact (- n 1))) ) ))
```

Nous voici ramené à considérer une affectation, c'est-à-dire une modification portant sur la variable `fact`, ce qui ne peut avoir de sens que si cette variable existe déjà ! On peut caractériser l'environnement global comme le lieu où préexistent toutes les variables. C'est bien sûr une *réalité virtuelle*, l'implantation effective (et plus précisément le mécanisme de lecture des programmes) se dépêchant, dès qu'une variable est nommée, de créer la liaison qui lui est associée dans l'environnement global et de faire comme si elle avait toujours été là. Toute variable est donc réputée préexister : la définir n'est que modifier sa valeur. Cette position pose néanmoins le problème de ce que peut être la valeur d'une variable qui n'a pas encore été affectée. Il faut là que l'implantation s'arrange pour attraper ce nouveau cas d'erreur correspondant à une liaison qui existe mais qui n'est pas initialisée. On le voit, la notion de liaison est complexe et sera analysée plus en détail au chapitre 4. **[☞ p. 105]**

On peut évacuer le problème lié à l'existence dérangeante d'une liaison existante et non initialisée en adoptant un autre point de vue. L'affectation ne peut que modifier une liaison déjà existante or les liaisons ne préexistent pas : il est donc nécessaire de pouvoir les créer quand on le désire et, pour cette tâche, `define` peut alors être considérée comme une nouvelle forme spéciale dont c'est le rôle. On ne peut donc référencer une variable ou modifier sa valeur que si elle a été auparavant créée. Mais à chaque médaille, son revers ! C'est ce qu'illustre le court extrait suivant :

```
(define (display-pi)
  (display pi) )
(define pi 2.7182818285)                                    ;WRONG
(define (print-pi)
  (display pi) )
(define pi 3.1415926536)
```

La définition de `display-pi` est-elle légale ? Son corps référence la variable `pi` alors même que celle-ci n'existe pas encore. La quatrième définition modifie la seconde (qu'elle corrige) mais, si le sens de `define` est de créer une nouvelle liaison, a-t-on le droit d'en créer une nouvelle de même nom ?

Là encore plusieurs réponses sont possibles. Deux visions au moins s'affrontent : la première correspond à l'extension des principes lexicaux aux définitions globales (comme en ML), la seconde à une vision plus dynamique et qui est celle plutôt utilisée par Lisp.

Dans un monde *global purement lexical* (que l'on nomme plus volontiers *hyperstatique*), on ne peut parler d'une variable c'est-à-dire la référencer, l'évaluer ou la

modifier que si elle existe déjà. La fonction `display-pi` est donc erronée puisque référençant la variable `pi` qui n'existe pas encore : tout appel à la fonction `display-pi` provoquera inéluctablement l'erreur « variable `pi` inconnue ». Ce ne sera pas le cas de la fonction `print-pi` qui affichera indéfiniment la valeur de la variable `pi` qui était valide au moment de sa définition à savoir `2.7182818285`. Dans ce monde, la « redéfinition » de `pi` est tout à fait légale et crée une nouvelle variable de même nom ayant pour portée le reste des interactions. Une vision imagée de ce mécanisme est de considérer la séquence précédente comme équivalente à :

```
(let ((display-pi (lambda () (display pi))))
  (let ((pi 2.7182818285))
    (let ((print-pi (lambda () (display pi))))
      (let ((pi 3.1415926536))
        ...
```

Les points de suspension marquent la suite des interactions ou le restant des définitions globales.

La vision plus dynamique qu'utilise Lisp est de ne considérer qu'il ne peut exister au plus qu'une seule variable globale par nom et que celle-ci est visible partout et qu'elle permet en particulier les références en avant sans qu'elles aient à être soulignées syntaxiquement (comme en Pascal avec la directive `forward` ou avec les prototypes d'ISO C [ISO90]).

La discussion n'est pas si oiseuse. On se sentira certainement interpellé par le choix de l'environnement dans lequel est évaluée la fonction qui sera la valeur de `fact` ? Si c'est l'environnement global, avant que ne soit créée la liaison de `fact`, alors `fact` ne peut être récursive. La raison en est simple, la référence à `fact` dans le corps de la fonction doit être recherchée dans l'environnement clos par la fermeture créée et cet environnement ne contient pas la variable `fact`. Il faut donc que l'environnement qui sera clos comprenne non seulement l'environnement global mais aussi la variable `fact`. Cette constatation milite en faveur d'un environnement global où toutes les variables préexistent car alors on ne se pose pas la précédente question. Si l'on adhère à la seconde vision hyperstatique, que l'on peut avoir de l'environnement global, alors il faut prendre garde à ce que `define` crée la liaison de `fact` avant d'évaluer la fermeture qui deviendra la valeur de cette liaison. En définitive, la récursion simple incombe à l'environnement global.

2.6.2 Récursion mutuelle

Supposons maintenant avoir à définir deux fonctions mutuellement récursives à savoir `odd?` et `even?` qui permettent de tester, fort inefficacement, la parité d'un entier naturel. Elles se définissent tout simplement comme :

```
(define (even? n)
  (if (= n 0) #t (odd? (- n 1))) )
(define (odd? n)
  (if (= n 0) #f (even? (- n 1))) )
```

Quel que soit l'ordre dans lequel sont énumérées ces deux définitions, la première ignore la seconde et, dans le cas présent, `even?` ne connaît aucunement `odd?`. Là encore, la vision de l'environnement global dans lequel toutes les variables préexistent semble gagnante car alors les deux fermetures enclosent l'environnement global, donc

toutes les variables et donc en particulier `odd?` et `even?`. On laisse, bien sûr, le soin à l'implantation de faire son affaire de la représentation d'un environnement global qui contiendrait une infinité (dénombrable tout de même) de variables.

Il est plus difficile d'adopter le point de vue des définitions globales purement lexicales car nécessairement la première définition ignore la seconde. Une solution serait d'introduire les deux définitions ensemble, simultanément, ce qui assurerait leur connaissance mutuelle. Nous reviendrons sur ce point lorsque nous aurons étudié les récursions locales. Par exemple et en réexhumant l'antique `define` de Lisp 1.5, on écrirait :

```
(define ((even? (lambda (n) (if (= n 0) #t (odd? (- n 1)))))
         (odd? (lambda (n) (if (= n 0) #f (even? (- n 1))))))
))
```

Les récursions mutuelles peuvent donc être exprimées par l'environnement global à condition que celui-ci soit doué des qualités ad-hoc. Qu'en est-il si, maintenant, l'on souhaite disposer de fonctions récursives locales ?

2.6.3 Récursion locale en Lisp$_2$

Les problèmes soulevés lors de la définition de `fact` dans l'environnement global se retrouvent partiellement lorsque l'on désire définir localement `fact` : il faut faire en sorte que l'appel à `fact` dans le corps de `fact` soit récursif, c'est-à-dire que la fonction associée à ce nom soit justement la factorielle valeur de `fact` dans l'environnement fonctionnel. Une différence majeure existe cependant : l'inexistence d'une liaison dans un environnement local se détecte simplement et est tout bonnement une erreur. Si l'on écrit en Lisp$_2$:

```
(flet ((fact (n) (if (= n 0) 1
                     (* n (fact (- n 1))) )))
  (fact 6) )
```

La fonction `fact` est liée dans l'environnement fonctionnel courant. Cette fermeture enclôt les environnements fonctionnels et paramétriques qui sont courants à l'endroit où la forme `flet` est évaluée. En conséquence, la fonction `fact` qui apparaît dans le corps de `fact` réfère la fonction `fact` valable à l'extérieur de la forme `flet`. Cette fonction n'a aucune raison d'être la factorielle et la récursion n'est donc pas obtenue.

Le problème était déjà connu en Lisp 1.5 où une forme spéciale, nommée `label`, permettait de définir une fonction récursive locale. On aurait alors écrit :

```
(label fact (lambda (n) (if (= n 0) 1
                            (* n (fact (- n 1))) ) ))
```

Cette forme retourne une fonction anonyme, calculant la factorielle. Cette fonction anonyme est, de plus, la valeur de la fonction `fact` qui apparaît dans son corps. On ne peut toutefois assurer que Lisp 1.5 ait été un authentique Lisp$_2$ [9] et, aussi intéressante qu'elle soit, la forme `label` n'est malheureusement pas apte à traiter simplement le cas des récursions mutuelles. Une version n-aire a donc été inventée beaucoup plus tard [HS75], c'est la forme spéciale `labels`. Cette dernière a la même syntaxe que la forme `flet` mais assure que les fermetures qui seront créées, le seront dans un environnement

9. C'est d'ailleurs ce qui a permis le foisonnement des dialectes de Lisp.

fonctionnel où toutes les fonctions locales se connaissent. On peut donc aussi bien définir localement la fonction récursive `fact` que les fonctions mutuellement récursives `even?` et `odd?`, comme le montre l'extrait suivant :

```
(labels ((fact (n) (if (= n 0) 1
                       (* n (fact (- n 1))) ) ))
   (fact 6) )     → 720
(funcall (labels ((even? (n) (if (= n 0) #t (odd? (- n 1))))
                  (odd? (n) (if (= n 0) #f (even? (- n 1)))) )
            (function even?) )
         4 )    → #t
```

On dispose donc en Lisp_2 de deux formes, `flet` et `labels`, permettant d'enrichir l'environnement fonctionnel local.

2.6.4 Récursion locale en Lisp_1

Le problème de la définition de fonctions locales récursives se pose aussi en Lisp_1 et se résoud semblablement. Une forme particulière nommée `letrec` procure un effet similaire à `labels`. Elle est ainsi nommée car c'est un « `let` récursif ».

La forme `let` a, en Scheme, la syntaxe suivante :

```
(let ((variable1 expression1)
      (variable2 expression2)
      ...
      (variablen expressionn) )
  expressions... )
```

Son effet est équivalent à l'expression :

```
((lambda (variable1 variable2 ...variablen) expressions...)
 expression1 expression2 ...expressionn )
```

Une explication plus discursive est : en premier lieu les $expression_1$, $expression_2$ ...$expression_n$ sont évaluées puis les variables $variable_1$, $variable_2$ ...$variable_n$ sont liées aux valeurs précédemment obtenues, enfin, dans l'environnement ainsi étendu, le corps du `let` est évalué (dans un `begin` implicite), sa valeur devenant celle de la forme `let` toute entière.

La forme `let` ne semble pas a priori utile puisque elle peut être simulée par `lambda` et donc n'être qu'une simple macro. Ce n'est d'ailleurs pas une forme spéciale de Scheme mais une syntaxe primitive. La forme `let` est cependant, sur le plan stylistique, bien commode car elle permet de placer côte-à-côte des variables et leur valeur d'initialisation à la façon d'un bloc Algol. Ce qu'il importe de remarquer est que les variables d'une forme `let` sont initialisées par des valeurs calculées dans l'environnement courant ; seul le corps du `let` est évalué dans l'environnement enrichi.

Ce fonctionnement ne permet pas d'écrire simplement des fonctions mutuellement récursives pour les mêmes raisons qu'en Lisp_2, aussi utilise-t-on `letrec` à cette fin. La syntaxe de la forme `letrec` est semblable à celle de la forme `let`. Par exemple, l'on écrira :

```
(letrec ((even? (lambda (n) (if (= n 0) #t (odd? (- n 1)))))
         (odd? (lambda (n) (if (= n 0) #f (even? (- n 1))))) )
   (even? 4) )
```

La différence d'avec `let` est que les expressions d'initialisation des variables sont évaluées dans l'environnement même où est évalué le corps de la forme `letrec`. Les

opérations qu'effectue `letrec` sont semblables à celles de `let` mais pas dans le même ordre. En premier lieu, l'environnement courant est étendu par les variables du `letrec` puis, dans cet environnement, sont évaluées les expressions d'initialisation de ces mêmes variables, enfin le corps est évalué toujours dans ce même environnement enrichi. Cette description suggère simplement comment implanter `letrec`, il suffit, pour obtenir ce même effet, d'écrire :

```
(let ((even? 'void)
      (odd? 'void) )
   (set! even? (lambda (n) (or (= n 0) (odd? (- n 1)))))
   (set! odd? (lambda (n) (or (= n 1) (even? (- n 1)))))
   (even? 4) )
```

Les liaisons concernant `even?` et `odd?` sont créées (les variables sont, au passage, liées à une valeur sans importance car `let` (ou `lambda`) ne permet pas de créer des liaisons non initialisées) puis les deux variables sont initialisées avec des valeurs qui sont calculées dans un environnement connaissant les variables `even?` et `odd?`. Le mot « connaissant » est juste car, si les liaisons des variables `even?` et `odd?` existent, leur valeur est sans relation avec ce que l'on en attend, tant qu'elles n'ont pas toutes été initialisées. Les liaisons concernant `even?` et `odd?` existent suffisamment pour être capturées mais pas assez pour être valablement évaluées.

La transformation n'est toutefois pas tout-à-fait correcte car, si une forme `let` est équivalente à une application fonctionnelle, alors l'ordre d'évaluation des formes d'initialisation, qui deviennent, au cours de la transcription, les arguments de l'application fonctionnelle engendrée, ne doit pas être spécifié. Malheureusement notre expansion force un ordre gauche-droite. **[☞ Ex. 2.9]**

Équations et `letrec`

Un problème majeur de `letrec` est que sa syntaxe n'est pas contraignante et permet que les formes d'initialisation soient quelconques et pas seulement des fonctions. Observez que la syntaxe de la forme `labels` de COMMON LISP interdit, elle, qu'il soit défini autre chose que des fonctions. Il est, en Scheme, possible d'écrire :

```
(letrec ((x (/ (+ x 1) 2))) x)
```

Observez que la variable `x` est définie en termes d'elle-même d'après la sémantique de `letrec`. C'est donc véritablement une équation qui est ainsi écrite : $x = \frac{x+1}{2}$. Il paraît logique de lier `x` à la solution de cette équation. La valeur finale de la forme `letrec` serait donc `1`.

Mais que devient cette interprétation si l'équation n'a pas de solution ou bien en admet plusieurs ?

```
(letrec ((x (+ x 1))) x)                ; x = x + 1
(letrec ((x (+ 1 (power x 37)))) x)     ; x = x^37 + 1
```

À côté des nombres d'autres domaines existent, tout aussi familiers de Lisp comme les S-expressions, où l'on peut parfois assurer qu'il y a toujours une solution unique [MS80]. Par exemple, on peut construire une liste infinie sans effet de bord apparent, un peu comme dans un langage paresseux, en écrivant :

```
(letrec ((foo (cons 'bar foo))) foo)
```

La valeur de cette forme pourrait alors être la liste infinie `(bar bar bar ...)` paresseusement calculée [FW76, PJ87] ou encore la structure circulaire (et moins coûteuse) obtenue par :

```
(let ((foo (cons 'bar 'wait)))
  (set-cdr! foo foo)
  foo )
```

Il nous faut donc adopter une règle plus pragmatique pour `letrec` qui interdise d'utiliser la valeur d'une variable du `letrec` lors des initialisations de ces mêmes variables. Ainsi les deux exemples précédents exigeaient, pour initialiser `x`, que soit connue la valeur de `x`. Cette situation est erronée et sera sanctionnée. Il faut toutefois remarquer que l'ordre d'initialisation n'étant pas spécifié en Scheme, certains programmes peuvent être erronés sur certaines implantations et ne pas provoquer d'erreur sur d'autres. C'est le cas par exemple de :

```
(letrec ((x (+ y 1))
         (y 2) ) )
    x )
```

Si la forme d'initialisation de `y` est évaluée avant celle concernant `x`, alors tout se passe bien. Dans le cas contraire, il y a erreur car l'on cherche à incrémenter `y` dont la liaison existe mais qui n'a pas encore de valeur. Quelques compilateurs Scheme ou ML analysent les expressions d'initialisation et les trient topologiquement afin de déterminer l'ordre selon lequel les évaluer. Ce tri n'est pas toujours possible lorsque l'on introduit des dépendances mutuelles du genre [10] :

```
(letrec ((x y)(y x)) (list x y))
```

Ceci nous rappelle furieusement certaine discussion sur l'environnement global et la sémantique de `define`. Nous tenons là un problème de même nature à savoir l'existence d'une liaison non initialisée.

2.6.5 Créer des liaisons non initialisées

La sémantique officielle de Scheme fait de `letrec` une syntaxe, c'est-à-dire une abréviation commode mais nullement nécessaire. Ainsi toute forme `letrec` est équivalente à une forme qui aurait pu être écrite directement en Scheme. C'est ce que nous avons tenté de faire plus haut toutefois nous avions lié temporairement les variables du `letrec` à `void`, ce qui malheureusement initialise les liaisons et interdit de détecter l'erreur sus-mentionnée. Notre infortune est encore pire car aucune des quatre formes spéciales de Scheme ne permet de créer des liaisons non initialisées.

Une première recherche de solution serait d'utiliser l'objet `#<UFO>` **[☞ p. 13]** à la place de `void`. On ne peut faire grand chose de `#<UFO>`, on ne peut l'additionner ni prendre son `car` mais, comme c'est un objet de première classe, on peut le faire apparaître dans un `cons`. Le programme suivant ne serait donc pas faux et retournerait `#<UFO>` :

```
(letrec ((foo (cons 'foo foo))) (cdr foo))
```

La raison profonde est que la non initialisation d'une liaison est une propriété de la liaison et non de son contenu. L'emploi de la valeur de première classe `#<UFO>` n'est donc pas une solution.

---

10. Encore que retourner en valeur la liste `(34 34)` ne viole pas l'énoncé de cette forme.

De manière implémentatoire, on marque souvent qu'une liaison est non initialisée en lui accordant une valeur très particulière. Supposons que cette valeur, que l'on nommera `#<uninitialized>`, soit, pour l'instant, de première classe. Lorsqu'elle apparaît comme valeur d'une variable, alors la variable doit être considérée comme non initialisée. On remplacera donc salutairement `void` par `#<uninitialized>` et on obtiendra ainsi la détection d'erreur que l'on cherchait. Toutefois, ce mécanisme est trop puissant car n'importe qui peut fournir `#<uninitialized>` comme argument à n'importe quelle fonction et l'on perd donc l'importante propriété que toute variable a une valeur. Ainsi la brave vieille factorielle ne peut même plus assumer que sa variable `n` est liée et doit donc tester explicitement que c'est une valeur régulière :

```
(define (fact n)
  (if (eq? n '#<uninitialized>)
      (wrong "Uninitialized n")
      (if (= n 0) 1
          (* n (fact (- n 1))) ) ) )
```

Le surcoût est trop important et donc `#<uninitialized>` ne peut être une valeur de première classe, ce ne doit être qu'un témoin interne réservé à l'implanteur que l'utilisateur ne peut manipuler. Cette seconde voie est donc condamnée comme solution.

Une troisième solution est d'introduire une nouvelle forme spéciale capable de créer des liaisons non initialisées. On profitera d'une variante syntaxique de `let`, existant en COMMON LISP mais pas en Scheme, pour ce faire :

```
(let ( variable ... )
     ... )
```

Lorsqu'une variable figure seule, sans forme d'initialisation, dans la liste des variables locales d'une forme `let`, alors la liaison sera créée non initialisée. Toute évaluation (ou tentative d'évaluation) de cette variable devra vérifier qu'elle est bien initialisée. L'expansion d'une forme `letrec` ne fait maintenant plus appel à quoi que ce soit d'étranger. Dans ce qui suit, les variables $temp_i$ sont hygiéniques, c'est-à-dire qu'elles ne peuvent provoquer de quelconques conflits avec les variables $variable_i$ et celles qui sont libres dans π.

```
(letrec ((variable1 expression1)
         ...
         (variablen expressionn) )
   corps )
```

$\equiv$

```
(let ( variable1 ...variablen )
  (let (( temp1 expression1 )
        ...
        ( tempn expressionn ) )
    (set! variable1 temp1)
    ...
    (set! variablen tempn)
    corps ) )
```

On a ainsi résolu le problème d'une manière satisfaisante car seules les variables non initialisées payent le surcoût dû à la non-initialisation. En revanche, la forme `let` n'est plus une syntaxe mais une forme spéciale primitive qui doit donc apparaître dans l'interprète de base. On ajoutera donc la clause suivante à `evaluate` :

```
...
((let)
 (eprogn (cddr e)
         (extend env
                 (map (lambda (binding)
                        (if (symbol? binding) binding
```

```
                    (car binding) ) )
              (cadr e) )
        (map (lambda (binding)
               (if (symbol? binding)
                   the-non-initialized-marker
                   (evaluate (cadr binding) env) ) )
              (cadr e) ) ) ) ) ...
```

La variable `the-non-initialized-marker` appartient au langage de définition et peut être définie comme :

```
(define the-non-initialized-marker (cons 'non 'initialized))
```

Bien sûr ce témoin interne est exploité par la fonction `lookup` qui doit être adaptée. La fonction `update!` n'est pas, quant à elle, affectée par ce changement.

```
(define (lookup id env)
  (if (pair? env)
      (if (eq? (caar env) id)
          (let ((value (cdar env)))
            (if (eq? value the-non-initialized-marker)
                (wrong "Uninitialized binding" id)
                value ) )
          (lookup id (cdr env)) )
      (wrong "No such binding" id) ) )
```

Ainsi donc, au terme de ces vagabondages syntaxico-sémantiques, avons-nous obtenu une forme `letrec` autorisant la co-définition de fonctions locales mutuellement récursives.

2.6.6 Récursion sans affectation

La forme `letrec` ci-dessus analysée use d'affectations pour assurer que les formes d'initialisation sont évaluées dans le bon environnement. Les langages dits *purement fonctionnels* n'ont pas cette ressource à leur disposition : les effets de bord y sont inconnus et qu'est donc l'affectation sinon un effet de bord sur la valeur d'une variable. Interdire l'affectation est une philosophie qui offre un énorme avantage ; la transparence référentielle du langage est préservée laissant ainsi libre champ à de nombreuses transformations de programmes telles que déplacement de code, évaluation parallèle ou paresseuse, etc. En revanche, ne pas disposer d'effets de bord nuit à la clarté de certains algorithmes et ne permet pas l'introspection d'une machine réelle qui, elle, ne fonctionne qu'avec les effets de bord qu'opèrent ses instructions.

La première solution qui vient à l'esprit est de faire de `letrec` une forme spéciale de plus, comme dans la plupart des langages de la classe de ML. On enrichira `evaluate` pour traiter le cas de `letrec` :

```
...
((letrec)
 (let ((new-env (extend env
                        (map car (cadr e))
                        (map (lambda (binding)
                               the-non-initialized-marker )
                             (cadr e) ) )))
   (map (lambda (binding)              ; map to preserve chaos !
          (update! (car binding)
                   new-env
                   (evaluate (cadr binding) new-env) ) )
```

```
        (cadr e) )
    (eprogn (cddr e) new-env) ) ) ...
```

On retrouve notamment l'effet de bord, naguère perpétré par affectation, aujourd'hui effectué par `update!`. On remarquera aussi que le désordre est préservé puisque `map`, au contraire de `for-each`, ne garantit rien sur l'ordre de traitement des termes des listes qu'elle parcourt [11].

`letrec` et environnement global purement lexical

Un environnement global purement lexical ne permet d'utiliser une variable que si elle a précédemment été définie. L'ennui est que l'on ne pouvait définir de fonctions simplement récursives ou des groupes de fonctions mutuellement récursives. Avec `letrec`, nous pouvons suggérer comment résoudre ces problèmes en procurant la possibilité de définir plus d'une fonction à la fois et en indiquant si la définition est récursive ou non, on écrira donc :

```
(letrec ((fact (lambda (n)
                 (if (= n 0) 1 (* n (fact (- n 1))))  )))
   (letrec ((odd? (lambda (n) (if (= n 0) #f (even? (- n 1)))))
            (even? (lambda (n) (if (= n 0) #t (odd? (- n 1))))) )
     ...
```

Combinateur paradoxal

Si l'on a des lettres, on pourra se souvenir avec fruit que le λ-calcul permet d'écrire des *combinateurs de point fixe* et parmi eux le célèbre *combinateur paradoxal* **Y**. Une fonction f possède un point fixe lorsqu'il existe un élément x de son domaine de définition tel que $x = f(x)$. Le combinateur **Y** prend n'importe quelle fonction (du λ-calcul) et retourne un point fixe de celle-ci. C'est un des plus beaux et plus profonds théorèmes du λ-calcul :

Théorème du point fixe : $\exists \mathbf{Y}, \forall F,\ \mathbf{Y}F = F(\mathbf{Y}F)$

En termes lispien, **Y** est la valeur de :

```
(let ((W (lambda (w)
           (lambda (f)
             (f ((w w) f)) ) )))
  (W W) )
```

La démonstration est très courte. Si l'on suppose que **Y** est égal à **(WW)** alors comment choisir **W** pour que **(WW)F** soit égal à **F((WW)F)** ? Manifestement, **W** n'est autre que $\lambda W.\lambda F.F((WW)F)$. L'expression ci-dessus n'est que la transcription lispienne de ce qui précède.

L'ennui est que la stratégie d'appel par valeur de Scheme n'est pas compatible avec cette programmation. Il faut lui ajouter une η-conversion superflue (du point de vue du λ-calcul pur) bloquant l'évaluation trop précoce du terme `((w w) f)`. On se rabattra donc sur la définition suivante du combinateur de point fixe dans lequel `(lambda (x) (...x))` marque la η-conversion opérée.

```
(define fix
```

11. Au prix ici du coût de fabrication d'une liste inutile puisqu'oubliée sitôt construite.

```
(let ((d (lambda (w)
           (lambda (f)
             (f (lambda (x) (((w w) f) x))) ) )))
  (d d) ) )
```

Le plus troublant dans cette définition est qu'elle fonctionne comme on va bientôt le montrer. Définissons la fonctionnelle `meta-fact` comme suit :

```
(define (meta-fact f)
  (lambda (n)
     (if (= n 0) 1
         (* n (f (- n 1))) ) ) )
```

Cette fonction entretient un rapport troublant avec la factorielle. On vérifiera, par exemple, que l'expression `(meta-fact fact)` calcule, aussi bien que `fact` mais plus lentement, la factorielle d'un nombre. Plus précisément, supposons que l'on connaisse un point fixe, f, de `meta-fact`, $f =$ `(meta-fact f)`. Ce point fixe a comme propriété d'être, par définition, une solution de l'équation fonctionnelle en f :

```
f = (lambda (n)
       (if (= n 1) 1
           (* n (f (- n 1))) ) )
```

Qu'est-ce que f peut donc être d'autre hors la fameuse factorielle ?

En fait rien ne garantit que l'équation fonctionnelle précédente a une solution et qu'elle est unique. Tous ces termes doivent bien sûr être mathématiquement bien définis et ce n'est pas l'objet de ce livre. L'équation a de nombreuses solutions et par exemple :

```
(define (autre-fact n)
  (cond ((< n 1) (- n))
        ((= n 1) 1)
        (else (* n (autre-fact (- n 1)))) ) )
```

Le lecteur est invité à vérifier que la fonction `autre-fact` est bien un point fixe de `meta-fact`. L'analyse de tous les points fixes montre qu'il y a un ensemble de réponses sur lesquelles elles s'accordent toutes : toutes, elles calculent la factorielle des entiers naturels. Elles ne divergent qu'aux endroits où `fact` divergeait dans des calculs sans fin. Sur les entiers négatifs, `autre-fact` retourne un résultat là où tant d'autres seraient possibles puisque l'équation fonctionnelle ne dit rien sur ces cas. Il existe donc [12] un plus petit point fixe qui est la moins déterminée des fonctions solutions de l'équation fonctionnelle.

Le sens à accorder à une définition comme celle de `fact` dans l'environnement global est qu'elle définit une fonction égale au plus petit des points fixes de l'équation fonctionnelle associée. Ainsi donc en écrivant :

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1))) ) )
```

doit-on penser que l'on vient d'écrire une équation où la variable est nommée `fact`. La forme `define`, quand à elle, résoud l'équation et lie la variable `fact` à la solution de cette équation fonctionnelle. Cette manière de voir nous éloigne des discussions **[☞ p. 51]** sur l'initialisation des variables globales et fait de `define` un résolveur d'équations. En fait, `define` est bien implanté comme suggéré plus haut. La récursion

12. Tout cela se démontre [Man74] !

au travers de l'environnement global et les règles normales d'évaluation font que c'est le plus petit point fixe qui est calculé, ouf !

Revenons donc à notre combinateur de point fixe `fix` et traçons l'exécution de `((fix meta-fact) 3)`. Dans la trace qui suit et sachant qu'il n'y a pas d'effet de bord, nous nous sommes permis de temps à autre, de substituer certaines variables directement par leur valeur.

```
   ((fix meta-fact) 3)
 = (((d d)|
          | d≡ (lambda (w)
                 (lambda (f)
                   (f (lambda (x)
                        (((w w) f) x) )) ) )
     meta-fact )
    3 )
 = (((lambda (f)                                   ; (i)
       (f (lambda (x)
            (((w w) f) x) )) )|
                              | w≡ (lambda (w)
                                     (lambda (f)
                                       (f (lambda (x)
                                            (((w w) f) x) )) ) )
     meta-fact )
    3 )
 = ((meta-fact
     (lambda (x)
       (((w w) meta-fact) x) ))|
                             | w≡ (lambda (w)
                                    (lambda (f)
                                      (f (lambda (x)
                                           (((w w) f) x) )) ) )
    3 )
 =((lambda (n)
     (if (= n 0) 1
         (* n (f (- n 1))) ) )|
                              | f≡ (lambda (x)
                                     (((w w) meta-fact) x))|
                                                           | w≡
                          (lambda (w)                                ⋮
                            (lambda (f)                              ↩
                              (f (lambda (x)
                                   (((w w) f) x) )) ) )
   3 )
 = (* 3 (f 2))|
              | f≡ (lambda (x)
                     (((w w) meta-fact) x))|
                                           | w≡
                                (lambda (w)                ⋮
                                  (lambda (f)              ↩
                                    (f (lambda (x)
                                         (((w w) f) x) )) ) )
 = (* 3 (((w w) meta-fact) 2))|
                              | w≡ (lambda (w)
                                     (lambda (f)
                                       (f (lambda (x)
                                            (((w w) f) x) )) ) )
 = (* 3 (((lambda (f)                                ; (ii)
```

```
          (f (lambda (x)
               (((w w) f) x) )) )
                                    | w≡ (lambda (w)
                                           (lambda (f)
                                             (f (lambda (x)
                                                  (((w w) f) x) )) ) )
           meta-fact )
         2 ) )
```

Nous nous arrêterons là en remarquant que, grossièrement, l'expression de l'étape *(i)* se retrouve à l'étape *(ii)* ce qui de fil en aiguille conduira vers :

```
(* 3 (* 2 (((lambda (f)
                (f (lambda (x)
                     (((w w) f) x) )) )
                                         | w≡ (lambda (w)
                                                (lambda (f)
                                                  (f (lambda (x)
                                                       (((w w) f) x) )) ) )
                   meta-fact )
                 1 )))
= (* 3 (* 2 ((meta-fact (lambda (x)
                          (((w w) meta-fact) x) ))
                                                  | w≡
                                 (lambda (w)                ⋮
                                   (lambda (f)              ↩
                                     (f (lambda (x)
                                          (((w w) f) x) )) ) )
                   1 )))
= (* 3 (* 2 ((lambda (n)
                 (if (= n 0) 1
                     (* n (f (- n 1))) ) )
                                         | f→ ...
                  1 )))
= (* 3 (* 2 (if (= n 0) 1 (* n (f (- n 1))))))
                                         | n→ 1
                                         | f→ ...
= (* 3 (* 2 1))
= 6
```

On peut observer au fil du calcul qu'un objet correspondant à la factorielle a bien été construit : c'est la valeur :

```
(lambda (x)
  (((w w) f) x) )
                  | f≡ meta-fact
                  | w→(lambda (w)
                        (lambda (f)
                          (f (lambda (x)
                               (((w w) f) x) )) ) )
```

Toute l'astuce est dans la recomposition d'une nouvelle instance de cette factorielle chaque fois qu'un appel récursif le nécessite.

Ainsi donc une simple récursion peut-elle être obtenue, sans effet de bord, grâce au combinateur de point fixe `fix`. Grâce à **Y** ou `fix` en Lisp, il est possible de définir `define` comme un résolveur d'équations récursives qui prend en argument une équation et lie sa solution à un nom. Ainsi l'équation définissant la factorielle conduit-elle à lier la variable `fact` à la valeur :

```
(fix (lambda (fact)
```

```
        (lambda (n)
          (if (= n 0) 1
              (* n (fact (- n 1))) ) ) ))
```

On peut étendre cette technique au cas de plusieurs fonctions mutuellement récursives en les regroupant. Les fonctions `odd?` et `even?` peuvent être fusionnées en :

```
(define odd-and-even
  (fix (lambda (f)
         (lambda (which)
           (case which
             ((odd) (lambda (n) (if (= n 0) #f
                                    ((f 'even) (- n 1)) )))
             ((even) (lambda (n) (if (= n 0) #t
                                     ((f 'odd) (- n 1)) ))) ) ) )) ))
(define odd? (odd-and-even 'odd))
(define even? (odd-and-even 'even))
```

L'ennui de cette méthode est qu'elle n'est pas performante comparée à une compilation même imparfaite d'une forme `letrec` (voir cependant [Roz92, Ser93]). Elle a toutefois ses adeptes notamment lorsqu'il s'agit d'écrire des bouquins ! Les langages fonctionnels ne l'adoptent pas non plus [PJ87] car, outre son inefficacité, la définition de `fix` n'est pas simplement typable. En effet sa nature est de prendre une fonctionnelle qui prend en argument la fonction typée $\alpha \to \beta$ et qui retourne un point fixe de cette fonctionnelle. Le type est alors :

$$\texttt{fix: } ((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$$

Mais dans la définition de `fix`, on a l'auto-application `(d d)` qui a pour type γ tel que :

$$\gamma = \gamma \to (\alpha \to \beta)$$

Il faut un système de types non trivial pour contenir ce type récursif ou considérer que `fix` est une fonction primitive qui, heureusement, préexiste car on ne saurait la définir au sein du langage si elle manquait.

2.7 Conclusion

Ce chapitre a présenté successivement quelques-uns des grands schismes qui ont scindé le monde Lisp dans les trente dernières années. Lorsque l'on regarde la « surface » des points d'éclosion des divergences, on s'aperçoit qu'elle est extrêmement réduite. Elle touche principalement le sens de `lambda` et la façon dont est calculée l'application fonctionnelle. Bien que la fonction semble un concept mathématiquement bien fondé, son incarnation au sein du langage fonctionnel (c'est-à-dire usant de fonctions) qu'est Lisp est bien souvent à l'origine de nombre de controverses. Connaître et apprécier ces divergences font partie de la culture lispienne. L'analyse des glorieuses traces des aînés fonde non seulement la compréhension mutuelle mais accroît les vertus roboratives et stylistiques des programmes.

Paradoxalement, ce chapitre a montré l'importance du concept de liaison. Dans un Lisp_1, à une variable, c'est-à-dire à un nom, est associée une unique liaison (éventuellement globale), donc une unique valeur. C'est pourquoi on parle de la valeur d'une variable plutôt que de la valeur associée à la liaison associée à cette variable. Vue

comme un type abstrait, une liaison est créée par une forme liante, est lue ou écrite par évaluation de variable ou affectation, peut enfin être capturée lorsqu'est créée une fermeture dont le corps réfère, en lecture ou écriture, la variable associée à cette même liaison.

Les liaisons ne sont pas des objets de première classe ; elles ne sont manipulées qu'indirectement par les variables auxquelles elles sont associées. Elles ont néanmoins une durée de vie illimitée : elles perdurent tant qu'elles sont utiles.

Une forme liante introduit une notion de portée. La portée d'une variable ou d'une liaison est l'espace textuel où cette variable est visible. La portée d'une variable liée par `lambda` est restreinte au corps de cette forme `lambda`. On parle alors de portée textuelle ou lexicale.

La notion de liaison est compliquée du fait de la présence de l'affectation et sera étudiée plus en détail au chapitre prochain.

2.8 Exercices

Exercice 2.1 : Comment se traduit en Scheme l'expression suivante écrite en COMMON LISP :

```
(funcall (function funcall) (function funcall) (function cons) 1 2)
```

Exercice 2.2 : Que vaut, dans le pseudo COMMON LISP vu dans ce chapitre, le programme suivant et quelles pensées vous inspire-t-il ?

```
(defun test (p)
  (function bar) )
(let ((f (test #f)))
  (defun bar (x) (cdr x))
  (funcall f '(1 . 2)) )
```

Exercice 2.3 : Incorporez à l'interprète Scheme les deux premières innovations présentées en section 2.3. **[☞ p. 37]** Ces innovations concernaient les nombres et les listes en position fonctionnelle.

Exercice 2.4 : La fonction `assoc/de` pourrait être améliorée pour prendre le comparateur (`eq?`, `equal?` ou autre) utilisé en argument. Écrivez cette nouvelle version.

Exercice 2.5 : Écrivez les macros simulant à l'aide de `bind/de` et `assoc/de` les formes spéciales `dynamic-let`, `dynamic` et `dynamic-set!`.

Exercice 2.6 : Écrivez des fonctions `getprop` et `putprop` simulant les listes de propriétés. On peut associer une valeur à une clé dans la liste de propriétés d'un symbole avec `putprop` ; on peut chercher la valeur associée à une clé dans la liste de propriétés d'un symbole avec `getprop`. On a bien sûr la propriété suivante :

```
(begin (putprop 'symbol 'key 'value)
       (getprop 'symbol 'key) )          → value
```

Exercice 2.7 : Définissez la forme spéciale `label` en Lisp_1.

Exercice 2.8 : Définissez la forme spéciale `labels` en Lisp_2.

Exercice 2.9 : Concevez une expansion de `letrec` en fonction de `let` et `set!` qui préserve la non-spécification de l'ordre d'évaluation des expressions d'initialisation.

Exercice 2.10 : Le combinateur de point fixe énoncé en Scheme a le défaut de ne fonctionner que sur des fonctions unaires. Donnez-en une version, `fix2`, fonctionnant correctement sur des fonctions binaires. Enfin écrivez une version, `fixN`, pouvant fonctionner sur des fonctions d'arité quelconque.

Exercice 2.11 : Écrivez maintenant une fonction `NfixN` retournant un point fixe d'une liste de fonctionnelles d'arité quelconque. On l'utilisera par exemple pour définir :

```
(let ((odd-and-even
       (NfixN (list (lambda (odd? even?)    ; odd?
                      (lambda (n)
                        (if (= n 0) #f (even? (- n 1))) ) )
                    (lambda (odd? even?)    ; even?
                      (lambda (n)
                        (if (= n 0) #t (odd? (- n 1))) ) ) )) ))
  (set! odd? (car odd-and-even))
  (set! even? (cadr odd-and-even)) )
```

Exercice 2.12 : La fonction `klop` est-elle un combinateur de point fixe ? On s'attachera à montrer que, tout comme `fix`, `(klop` f`)` retourne un point fixe de f.

```
(define klop
  (let ((r (lambda (s c h e m)
             (lambda (f)
               (f (lambda (n)
                    (((m e c h e s) f) n) )) ) )))
    (r r r r r r) ) )
```

Exercice 2.13 : Soit la fonction `hyper-fact` ainsi définie :

```
(define (hyper-fact f)
  (lambda (n)
    (if (= n 0) 1
        (* n ((f f) (- n 1))) ) ) )
```

Que vaut `((hyper-fact hyper-fact) 5)` ?

2.9 Lectures recommandées

Outre le *λ-paper* [SS78a] déjà mentionné, on pourra méditer l'article historique de [Mos70] ainsi que l'étude comparative Lisp_1/Lisp_2 [GP88]. Une intéressante introduction au λ-calcul figure dans [Gor88], enfin le combinateur **Y** est aussi traité en [Gab88].

3

Échappement, reprise : continuations

Tout calcul a pour but de retourner une valeur à une certaine entité que l'on nomme : continuation. Ce chapitre explique cette notion et ses racines historiques, décrit un nouvel interprète les rendant explicites, présente diverses mises en œuvre en Lisp et en Scheme et, enfin, approfondit le style dit de la programmation par continuations. Lisp se singularise parmi les langages de programmation par l'existence de formes élaborées de manipulation du fil du contrôle de l'exécution. Cette richesse se traduira dans ce chapitre par le grand air du catalogue [Moz87] : les *mil e tre* formes de contrôle que nous étudierons tour à tour. Il est cependant bon de conserver, aussi, à l'esprit la face cachée des continuations : leur réalisation physique. Le nouvel interprète usera d'objets qui marqueront clairement la parenté des continuations et des blocs de contrôle dans la fameuse *pile d'évaluation*.

Les interprètes précédents se saisissaient d'une expression (et d'un environnement) afin d'en déterminer la valeur. Toutefois ils n'étaient point capables de décrire des calculs comportant des *échappements*, structures de contrôle commodes lorsqu'il s'agit de s'évader d'un contexte pour en retrouver un autre jugé préférable. Les échappements servent principalement, en programmation classique, à maîtriser le comportement des programmes en cas d'erreurs inopinées ou encore à programmer par exceptions lorsque l'on décrit un comportement général où la survenue d'un événement spécifique interrompt ce calcul et le force dans une voie appropriée.

Les racines historiques des échappements remontent à Lisp 1.5 avec la forme `prog`. Celle-ci, maintenant obsolète, avait été introduite dans le vain espoir d'attirer à Lisp les adeptes d'Algol, dont on pensait qu'ils ne savaient programmer qu'à grands coups d'affectations, et de débranchements impératifs : le célèbre `goto`. Las, cette forme a plutôt détourné nombre de lispeurs [1] des sains (saints ?) préceptes de la programmation fonctionnelle récursive terminale. La forme `prog` mérite cependant d'être détaillée car elle apportait quelques traits intéressants. Voici la factorielle écrite avec `prog` :

COMMON LISP
```
(defun fact (n)
  (prog (r)
```

1. On appréciera les différences stylistiques qui existent entre les première et troisième éditions de [WH88].

```
      (setq r 1)
loop  (cond ((= n 1) (return r)))
      (setq r (* n r))
      (setq n (- n 1))
      (go loop) ) )
```

La forme spéciale `prog` déclare, en premier, les variables locales utilisées, ici seulement la variable `r`. Les expressions qui suivent sont des instructions (représentées par des listes) ou des étiquettes de branchement (représentées par des symboles). Les instructions sont évaluées séquentiellement comme dans un `progn`, la valeur d'une forme `prog` étant normalement `nil`. Plusieurs instructions spéciales peuvent être utilisées dans une forme `prog` : les débranchements inconditionnels sont spécifiés par `go` (l'étiquette n'est pas calculée) tandis que `return` impose la valeur finale de la forme `prog`. Une restriction existait en Lisp 1.5, formes `go` ou formes `return` ne pouvaient apparaître qu'au premier niveau des `prog` ou des `cond` de premier niveau.

La forme `return` permet d'échapper à la forme `prog` en imposant son résultat. La restriction de Lisp 1.5 ne permettait que de s'échapper d'un contexte simple, ses successeurs améliorèrent ce point en autorisant que `return` puisse être utilisé sans restriction. Cette fonctionnalité d'échappement devint le mode normal selon lequel furent traitées les erreurs. Lorsqu'une erreur survient, le calcul cherche à s'échapper du contexte erroné afin de regagner un contexte plus sain. On put donc réécrire le précédent exemple en le code équivalent suivant où la forme `return` n'apparaît plus au premier niveau :

COMMON LISP

```
(defun fact2 (n)
  (prog (r)
        (setq r 1)
   loop (setq r (* (cond ((= n 1) (return r))
                         ('else n) )
                   r ))
        (setq n (- n 1))
        (go loop) ) )
```

Si l'on réduit les formes `prog` et `return` à leur seuls effets sur le fil de contrôle, on voit bien qu'elles manipulent le point d'appel (et de retour) de la forme `prog`, exactement comme dans le cas d'une application fonctionnelle où la fonction invoquée retourne sa valeur à l'endroit où elle a été précisément appelée. On peut en quelque sorte dire que `prog` *lie* `return` au point précis où il faudra revenir avec une valeur. L'échappement consiste donc à ignorer l'endroit d'où l'on part en spécifiant l'endroit où l'on désire arriver. On peut aussi espérer de ce *saut* qu'il soit efficacement implanté : les échappements deviennent alors une méthode de programmation qui peut avoir ses adeptes. On peut le voir sur l'exemple suivant qui consiste à rechercher un symbole dans un arbre binaire. La programmation la plus naïve de cet algorithme, en Scheme, est :

```
(define (find-symbol id tree)
  (if (pair? tree)
      (or (find-symbol id (car tree))
          (find-symbol id (cdr tree)) )
      (eq? tree id) ) )
```

Supposons que l'on cherche le symbole `foo` au sein de l'expression `((( a . b) . ( foo . c)) . ( d . e ))`. Comme la recherche s'effectue de la gauche vers la droite et en profondeur d'abord, lorsque le symbole aura été trouvé, il faudra que la valeur `#t`

passe au travers de plusieurs formes `or` imbriquées avant de devenir la valeur finale : l'aboutissement de la recherche. C'est ce que manifeste le détail du calcul précédent par formes équivalentes :

```
(find-symbol 'foo '(((a . b) . (foo . c)) . (d . e)))
≡ (or (find-symbol 'foo '((a . b) . (foo . c)))
      (find-symbol 'foo '(d . e)) )
≡ (or (or (find-symbol 'foo '(a . b))
          (find-symbol 'foo '(foo . c)) )
      (find-symbol 'foo '(d . e)) )
≡ (or (or (or (find-symbol 'foo 'a)
              (find-symbol 'foo 'b) )
          (find-symbol 'foo '(foo . c)) )
      (find-symbol 'foo '(d . e)) )
≡ (or (or (find-symbol 'foo 'b)
          (find-symbol 'foo '(foo . c)) )
      (find-symbol 'foo '(d . e)) )
≡ (or (find-symbol 'foo '(foo . c))
      (find-symbol 'foo '(d . e)) )
≡ (or (or (find-symbol 'foo 'foo)
          (find-symbol 'foo 'c) )
      (find-symbol 'foo '(d . e)) )
≡ (or (or #t
          (find-symbol 'foo 'c) )
      (find-symbol 'foo '(d . e)) )
≡ (or #t
      (find-symbol 'foo '(d . e)) )
→ #t
```

Un échappement efficace paraîtrait donc approprié. Cet échappement devrait permettre, sitôt trouvé le symbole recherché, de retourner la valeur idoine sans se soucier des multiples branches de l'arbre restant à explorer.

Un autre exemple d'emploi ressortit de la programmation par exception où un traitement répétitif est appliqué sempiternellement jusqu'à détection d'une situation exceptionnelle auquel cas un échappement est effectué afin de justement *échapper* au non moins sempiternel traitement qui autrement continuerait derechef. Nous en verrons un exemple plus loin avec la fonction `better-map`. **[☞ p. 75]**

Si l'on veut mieux caractériser l'entité correspondant à un point d'appel, on remarquera fort naturellement qu'un calcul précise non seulement l'expression à calculer et l'environnement de variables dans lequel il faut calculer mais aussi à qui l'on doit retourner la valeur qui sera obtenue. Cette dernière entité s'appelle une *continuation* et elle représente tout ce qui reste à faire en un point. Tout calcul a une continuation, par exemple, dans l'expression `(+ 3 (* 2 4))`, la sous-expression `(* 2 4)` a pour continuation le calcul d'une addition où le premier argument est `3` et où le second est attendu. Là où la théorie vient à notre secours, c'est que l'on peut représenter cette continuation sous une forme plus perceptible : celle d'une fonction. Une continuation représente un calcul et ne se déroulera que si on lui procure une valeur : ce protocole ressemble furieusement à celui d'une fonction et c'est donc sous cette forme que seront représentées les continuations. Pour l'exemple précédent, la continuation de la sous-expression `(* 2 4)` sera donc l'équivalent de `(lambda (x) (+ 3 x))`, soulignant fidèlement que le calcul attend la valeur du second argument de l'addition. Une autre représentation existe qui spécifie les continuations par des contextes inspirés du

λ-calcul. On noterait `(+ 3 [])` la précédente continuation, où `[]` marque l'endroit où une valeur est attendue.

Vraiment tout a une continuation : l'évaluation de la condition d'une alternative s'effectue avec une continuation qui est d'exploiter la valeur qui sera retournée afin de décider si l'on prend la branche vraie ou fausse de l'alternative. Dans l'expression `(if (foo) 1 2)`, la continuation de l'application fonctionnelle `(foo)` est donc `(lambda (x) (if x 1 2))` ou `(if [] 1 2)`.

Les échappements, la programmation par exception, etc. ne sont que des formes particulières de manipulation de continuations dont nous allons détailler maintenant les diverses formes et grandes variantes observées ces vingt dernières années.

3.1 Formes de manipulation des continuations

Capturer la continuation permet de manipuler le fil de contrôle. La forme `prog` permettait déjà de le faire mais possédait aussi d'autres effets comme celui d'un `let` pour ses variables locales. Dépouiller `prog` pour ne s'intéresser qu'au contrôle fut le but premier des formes `catch` et `throw`.

3.1.1 Le couple `catch` et `throw`

La forme spéciale `catch` a la syntaxe suivante :

`(catch` *étiquette formes...*`)`

Le premier argument *étiquette* est évalué et sa valeur est associée à la continuation courante. On supposera donc l'existence d'un nouvel espace, l'*espace des échappements dynamiques*, liant valeurs et continuations. Si l'on associe noms et identificateurs, on ne peut plus réellement parler d'espace de noms ; c'en est toutefois bien un si l'on accepte que toute valeur soit un nom légal mais cela pose toutefois le problème de l'égalité dans cet espace : comment reconnaître qu'une valeur nomme ce qu'une autre a associé à une continuation ? Les autres *formes...* forment le corps de la forme `catch` et sont évalués séquentiellement comme dans un `progn` (ou `begin`). Si rien ne se passe, la valeur de la forme `catch` est celle de la dernière des formes de son corps. Ce qui peut toutefois se passer c'est l'évaluation d'une forme `throw`.

La forme `throw` a pour syntaxe :

`(throw` *étiquette forme*`)`

Le premier argument est évalué et doit conduire à un objet auquel une forme `catch` dynamiquement englobante a associé une continuation. Si c'est bien le cas, alors l'évaluation du corps de cette forme `catch` est interrompue et la valeur de *forme* devient celle de la forme `catch` toute entière.

Reprenons notre exemple de recherche de symbole dans un arbre binaire et incorporons-y les formes `catch` et `throw`. La variante qui suit a factorisé le processus de recherche afin de ne pas transmettre inutilement la variable `id` qui est constante dans toute la quête. On usera donc d'une fonction locale pour ce faire.

```
(define (find-symbol id tree)
  (define (find tree)
     (if (pair? tree)
```

```
        (or (find (car tree))
            (find (cdr tree)) )
        (if (eq? tree id) (throw 'find #t) #f) ) )
(catch 'find (find tree)) )
```

Comme son nom l'indique, `catch` attrape la valeur que lui envoit `throw`, l'échappement est ainsi une transmission directe de valeur associée à une manipulation du contrôle. En d'autres termes encore, la forme `catch` est une forme de liaison qui associe à la valeur de l'étiquette, la continuation courante. La référence à cette liaison s'effectue grâce à la forme `throw`. Cette dernière altère aussi le fil du contrôle : `throw` n'a pas vraiment de valeur car elle ne retourne pas de résultat par elle-même mais s'arrange pour que `catch` en retourne un. La forme `catch` capture, ici, la continuation d'appel à `find-symbol`, tandis que `throw` réalise un retour direct à l'appelant de `find-symbol`.

L'espace des échappements dynamiques peut être caractérisé par l'habituel tableau recensant les diverses propriétés qu'il procure :

Référence	`(throw` *étiquette* `...)`
Valeur	non car continuation de seconde classe
Modification	non
Extension	`(catch` *étiquette* `...)`
Définition	non

Comme sus-mentionné, `catch` n'est pas une fonction mais une forme spéciale qui, successivement, évalue son premier argument (l'étiquette), lie sa propre continuation à cette dernière valeur dans l'environnement des échappements dynamiques puis débute l'évaluation des autres arguments. Tous ne seront point nécessairement évalués. Lorsque `catch` retourne une valeur ou lorsque l'on s'en échappe, `catch` désassocie étiquette et continuation. L'effet de `throw` peut être procuré par une fonction ou une forme spéciale. Dans le cas d'une forme spéciale, comme en COMMON LISP, l'étiquette est calculée, l'existence d'un `catch` associé est vérifiée puis la valeur à transmettre est évaluée, enfin, le saut a lieu. Si `throw` est une fonction, les traitements sont ordonnés différemment : les deux arguments de `throw` sont tout d'abord calculés puis l'existence d'un `catch` associé est vérifiée avant que d'y sauter.

Ces différentes sémantiques illustrent bien la faible précision des descriptions textuelles du comportement des formes de contrôle. De nombreuses questions demeurent en suspens : que se passe-t-il quand il n'y a pas de `catch` associé ? Quelle est l'égalité utilisée pour comparer les étiquettes ? Que fait `(throw` α `(throw` β π`))` ? Nous verrons les réponses plus loin.

3.1.2 Le couple `block` et `return-from`

Les échappements que réalisent `catch` et `throw` sont de nature dynamique. Lorsqu'un échappement est demandé par `throw`, il faut vérifier, à l'exécution, si un `catch` associé existe et quelle continuation il référence. En termes d'implantation, il y a un coût non négligeable que l'on peut souhaiter diminuer grâce aux échappements dits *lexicaux* suivant la terminologie de COMMON LISP. Les formes spéciales `block` et `return-from` ressemblent superficiellement à `catch` et `throw`.

La forme `block` a pour syntaxe :

`(block` *label formes*`)`

Le premier argument n'est pas évalué et doit être un nom permettant de nommer l'échappement. La forme `block` lie la continuation courante au nom *label* dans l'*environnement des échappements lexicaux*. Le corps de `block` est alors évalué dans un `progn` implicite, la valeur de ce dernier devenant la valeur de ce premier. Cette évaluation séquentielle peut être rompue par `return-from` qui a pour syntaxe :

`(return-from` *label forme*`)`

Le premier argument n'est pas évalué et doit mentionner le nom d'un échappement lexicalement visible : une forme `return-from` ne peut apparaître que dans le corps d'un `block` associé, exactement comme une variable ne peut apparaître que dans le corps de la forme `lambda` associée. Lorsqu'une forme `return-from` est évaluée, elle fait en sorte que la forme `block` associée retourne la valeur de *forme*. Les échappements lexicaux forment un nouvel espace de noms, dont on synthétisera les propriétés dans l'habituel tableau :

Référence	`(return-from` *label* `...)`
Valeur	non car continuation de seconde classe
Modification	non
Extension	`(block` *label* `...)`
Définition	non

Reprenons alors l'exemple précédent que l'on réécrira en [2] :

```
(define (find-symbol id tree)
  (block find
    (letrec ((find (lambda (tree)
                     (if (pair? tree)
                         (or (find (car tree))
                             (find (cdr tree)) )
                         (if (eq? id tree) (return-from find #t)
                             #f ) ) )))
      (find tree) ) ) )
```

Observez que ce n'est pas une banale traduction de l'exemple précédent où « `catch 'find` » se serait transmué en « `block find` » et similairement pour `throw`. Sans la migration de « `block find` », la forme `return-from` n'aurait pas été lexicalement reliée au `block` convenable.

Le code engendré correspondant à cette forme est extrêmement efficace ; la forme `block`, en gros et en général, mémorise la hauteur de la pile d'exécution sous le nom d'échappement `find`. La forme `return-from` consiste seulement à mettre `#t` là où une valeur est attendue (par exemple dans un registre) et à réajuster le pointeur de pile à la hauteur que `block` avait associée à `find`. Quelques instructions seulement comparées au comportement dynamique de `catch`. Ce dernier déviderait peu à peu la liste de tous les échappements valides jusqu'à en trouver un qui ait la bonne étiquette, ce qui apparaît limpidement si l'on simule `catch` par `block` [3] :

```
(define *active-catchers* '())
```

---

2. Les schemeurs avertis remarqueront la transformation du `define` interne en un `letrec` car `(let () (define ...) ...)` n'est pas licite en Scheme à cause du `()`.

3. La définition de `catch` emploie une forme `block` avec le nom `label`. Les règles d'hygiène de Scheme doivent bien sûr être étendues à cet espace de noms.

```
(define-syntax throw
  (syntax-rules ()
    ((throw tag value)
     (let* ((label tag)                    ; compute once
            (escape (assv label            ; compare with  eqv?
                          *active-catchers* )) )
       (if (pair? escape)
           ((cdr escape) value)
           (wrong "No associated catch to" label) ) ) ) ) )
(define-syntax catch
  (syntax-rules ()
    ((catch tag . body)
     (let* ((saved-catchers *active-catchers*)
            (result (block label
                      (set! *active-catchers*
                            (cons (cons tag
                                        (lambda (x)
                                          (return-from label x) ) )
                                  *active-catchers* ) )
                      . body )) )
       (set! *active-catchers* saved-catchers)
       result ) ) ))
```

Dans cette simulation le coût du couple `catch`/`throw` est entièrement concentré dans l'appel à `assv`[4] qui figure dans l'expansion de `throw`. Mais expliquons auparavant comment fonctionne cette simulation. Une variable globale, ici nommée `*active-catchers*`[5], recense l'ensemble des formes `catch` actives, c'est-à-dire dont l'exécution n'est pas achevée. Cette variable est mise à jour en sortie de `catch` et donc en sortie de `throw`. La valeur de `*active-catchers*` est une A-liste dont les clés sont les étiquettes de `catch` et dont les valeurs sont les continuations associées. Cette A-liste matérialise l'environnement des échappements dynamiques, dont `catch` était la forme liante et dont `throw` était la forme de référence, ce qui apparaît aisément sur le code ci-dessus.

3.1.3 Échappements à durée de vie dynamique

Cette simulation est toutefois imparfaite en ce sens qu'elle interdit l'usage simultané de `block` qui pertuberait le maintien de la valeur de la variable globale `*active-catchers*`. L'art de la simulation ou de l'extension syntaxique d'un dialecte est un art difficile [Bak92b], car souvent l'adjonction de nouveaux traits requiert une architecture complexe interdisant l'accès direct aux ressources mobilisées pour ce faire. On verra plus loin une véritable simulation de `catch` par `block` mais elle demande un moyen linguistique de plus : `unwind-protect`[6].

Comme toutes les entités de Lisp, les continuations ont une certaine durée de vie. On l'a vu dans la simulation de `catch` par `block`, la durée de vie d'une continuation

---

4. Au passage, on voit que la comparaison entre étiquettes s'effectue avec `eqv?`.

5. Les variables globales sont souvent préfixées et suffixées en COMMON LISP par une étoile afin de les rendre plus visibles. Rien, sinon l'habitude, n'impose cette convention.

6. Une autre solution serait de redéfinir de nouvelles formes `block` et `return-from` (à l'aide des anciennes) compatibles de `catch` et `throw`. Cette solution est difficile à mettre en œuvre directement mais peut se faire par adjonction d'un nouveau niveau d'interprétation.

prise par `catch` est limitée à la durée du calcul du corps du `catch` en question. On qualifie cette durée de vie de *dynamique*. L'emploi de ce qualificatif est aussi liée à la liaison dynamique qui n'assure ce qu'est la valeur d'une variable dynamique que pendant l'évaluation du corps de la forme liante correspondante. On peut mettre à profit cette propriété pour proposer une nouvelle définition de `catch` et `throw` en termes de `block` et `return-from` où maintenant la liste des « catcheurs » activables est, sans effort aucun, maintenue dans la variable dynamique `*active-catchers*`. Cette programmation réconcilie `block` et `catch` qui peuvent simultanément être utilisés.

```
(define-syntax throw
  (syntax-rules ()
    ((throw tag value)
     (let* ((label tag)                                    ; compute once
            (escape (assv label (dynamic *active-catchers*))) )
       (if (pair? escape)
           ((cdr escape) value)
           (wrong "No associated catch to" label) ) ) ) ) )
(define-syntax catch
  (syntax-rules ()
    ((catch tag . body)
     (block label
       (dynamic-let ((*active-catchers*
                      (cons (cons tag (lambda (x)
                                        (return-from label x) ))
                            (dynamic *active-catchers*) ) ))
         . body ) ) ) ) )
```

La durée de vie d'un échappement pris par `block`, respectivement `catch`, est dynamique en COMMON LISP ; l'échappement ne peut être utilisé que pendant le calcul du corps de la forme `block`, resp. `catch`. Ceci pose un intéressant problème avec `block` qui n'a pas son équivalent avec `catch`. Si `throw` et `return-from` permettent d'abréger des calculs encore faut-il qu'il reste des calculs à abréger. Considérez le programme suivant :

```
((block foo (lambda (x) (return-from foo x)))
 33 )
```

La valeur du premier terme est un échappement vers `foo`. Mais cet échappement est obsolète lorsqu'il est appliqué et l'on obtient ainsi une erreur d'exécution. Lorsque la fermeture est créée, elle enclôt l'environnement des échappements lexicaux et notamment la liaison de `foo`. Cette fermeture est alors retournée comme valeur de la forme `block`, ce qui fait que l'on est sorti de `block` et qu'il est donc hors de question d'en ressortir à nouveau puisque c'est déjà fait ! Quand cette fermeture est appliquée, on vérifie que la continuation associée à `foo` est toujours valide et on saute dessus si c'est le cas. Que les fermetures enclosent les échappements lexicaux permet de ne pas mélanger les liaisons comme le montre l'exemple suivant :

```
(block foo
  (let ((f1 (lambda (x) (return-from foo x))))
    (* 2 (block foo
           (f1 1) )) ) ) → 1
```

Comparez ce résultat avec celui qui serait obtenu en remplaçant « `block foo` » par « `catch 'foo` » :

```
(catch 'foo
```

```
(let ((f1 (lambda (x) (throw 'foo x))))
  (* 2 (catch 'foo
         (f1 1)) ) ) → 2
```

Le « catcheur » invoqué par la fonction `f1` est le plus récent ayant lié l'étiquette `foo` ; le résultat de `catch` est donc multiplié par deux et retourne finalement `2`.

3.1.4 Comparaison entre `catch` et `block`

Les formes `catch` et `block` ont de nombreux points de comparaison. Les continuations qu'elles capturent ont une durée de vie dynamique : elles ne vivent que l'espace d'une évaluation. En revanche, `return-from` permet de référencer indéfiniment une telle continuation tandis que `throw` est plus limitée. En terme d'efficacité, `block` est meilleure dans la plupart des cas, car elle n'a jamais besoin de vérifier, lors d'un `return-from`, que le `block` correspondant existe puisque c'est assuré par la syntaxe. Il faut par contre vérifier que l'échappement n'est pas obsolète, ce qui est souvent visible syntaxiquement. On peut tracer un parallèle entre les échappements dynamiques et lexicaux et les variables affublées des mêmes qualificatifs : de nombreux problèmes sont communs.

Les échappements dynamiques autorisent des conflits étrangers aux échappements lexicaux. Les échappements dynamiques peuvent être mis en œuvre de partout. Une fonction peut mettre en place un échappement qu'une autre intercepte sans le savoir. Par exemple :

```
(define (foo)
  (catch 'foo (* 2 (bar))) )
(define (bar)
  (+ 1 (throw 'foo 5)) )
(foo)   → 5
```

Indépendamment de la durée de vie dynamique de l'échappement, `block` limite sa référence à son corps tandis que `catch` l'autorise tant qu'il vit. Il est donc possible d'utiliser l'échappement lié à `foo` pendant tout le temps que dure l'évaluation de `(* 2 (bar))`. On ne peut dans l'exemple précédent remplacer « `catch 'foo` » par « `block foo` ». Un accrochage plus dangereux serait le suivant :

```
(catch 'not-a-pair
       (better-map (lambda (x)
                     (or (pair? x)
                         (throw 'not-a-pair x) ) )
                   (hack-and-return-list) ) )
```

Supposons que l'on ait entendu dire que la fonction `better-map` est bien supérieure à `map` ; supposons, en outre, que l'on se soit risqué à l'utiliser afin de tester que le résultat de `(hack-and-return-list)` est bien une liste formée de paires ; supposons de plus que nous ne connaissions pas la définition de `better-map` qui est :

```
(define (better-map f l)
  (define (loop l1 l2 flag)
    (if (pair? l1)
        (if (eq? l1 l2)
            (throw 'not-a-pair l)
            (cons (f (car l1))
                  (loop (cdr l1)
                        (if flag (cdr l2) l2)
```

```
                        (not flag) ) ) ) ) )
    (loop l (cons 'ignore l) #t) )
```

La fonction `better-map` est effectivement intéressante car elle s'arrête même si la liste de second argument est cyclique. Mais si `(hack-and-return-list)` retournait la liste infinie `#1=(( foo . hack ) . #1#)` [7] alors, `better-map` chercherait à s'échapper. Si ce n'est pas spécifié dans son interface d'utilisation, il y aura un conflit de noms et un carambolage d'échappements. Là encore, on peut changer les noms afin de limiter ces conflits, ce qui est simple car `catch` permet que son étiquette soit toute valeur Lisp possible, ce peut être en particulier une paire pointée que l'on a fabriquée soi-même. On réécrirait donc plus sûrement :

```
(let ((tag (list 'not-a-pair)))
  (catch tag
       (better-map (lambda (x)
                     (or (pair? x)
                         (throw tag x) ) )
                   (foo-hack) ) ) )
```

Il est possible de simuler `block` avec `catch` mais sans gagner l'efficacité native de `block`. Il suffit de convertir les échappements lexicaux en échappements dynamiques :

```
(define-syntax block
  (syntax-rules ()
    ((block label . body)
     (let ((label (list 'label)))
       (catch label . body) ) ) ) )

(define-syntax return-from
  (syntax-rules ()
    ((return-from label value)
     (throw label value) ) ) )
```

La macro `block` crée une étiquette unique adaptée et la lie lexicalement à la variable éponyme, ce qui assure que seules les `return-from` présentes dans son corps verront cette étiquette. Ce faisant, on pollue l'espace des variables avec le nom `label`. Pour pallier ce défaut, on peut chercher à utiliser un nom non conflictuel ou encore un nom créé par `gensym` mais à ce moment il faut s'arranger pour que ce soit le même qui apparaisse dans `catch` et `throw`.

3.1.5 Échappement à durée de vie illimitée

Dans la grande reconstruction de Lisp à laquelle s'est livrée Scheme vers 1975, il a été proposé qu'une durée de vie illimitée soit conférée aux continuations prises par `catch` ou `block`. Cette propriété leur apporte de nouvelles caractéristiques très intéressantes. Plus tard, dans le souci de diminuer le nombre de formes spéciales de Scheme, il fut entrepris de capturer la continuation par une fonction ainsi que de représenter une continuation comme une fonction, c'est-à-dire comme un citoyen de première classe. Landin, dès 1965 [Lan65], proposa l'opérateur *J* qui permettait de capturer la continuation. La fonction `call/cc` de Scheme en est la descendante.

---

7. On a ici utilisé la notation COMMON LISP pour les données cycliques. On aurait aussi pu la fabriquer par `(let ((p (list (cons 'foo hack)))) (set-cdr! p p) p)`.

Tentons d'expliquer sa syntaxe pour le moins complexe à première approche. Tout d'abord il s'agit de capturer une continuation : il faut donc une forme qui capture sa continuation k d'appel :

```
k( ... )
```

Comme ce doit être une fonction, nommons-la `call/cc` :

```
k(call/cc ... )
```

Maintenant que l'on a capturé k, il faut la fournir à l'utilisateur, mais comment ? On ne peut la retourner en valeur de la forme `call/cc` puisqu'alors on rendrait k à k ! Si l'utilisateur attend cette valeur pour pouvoir l'employer dans un calcul, c'est qu'il peut matérialiser ce calcul en une fonction unaire [8] puisque c'est justement un objet qui attend une valeur pour l'employer dans un calcul. On peut donc fournir cette fonction en argument à `call/cc` :

```
k(call/cc (lambda (k) ... ) )
```

La continuation k est donc *réifiée* [9] en une valeur de première classe sur laquelle la fonction argument de `call/cc` est invoquée. Le dernier choix dû à Scheme est de ne pas créer un nouveau type d'objets et donc de représenter les continuations par des fonctions unaires indiscernables des fonctions créées par `lambda`. La continuation k est donc réifiée en ce qui devient la valeur de `k` et il suffit d'invoquer la fonction `k` pour transmettre son argument à l'appelant de `call/cc` :

```
k(call/cc (lambda (k) (+ 1 (k 2))))    → 2
```

Une autre solution aurait été de créer un nouveau type d'objets : les continuations. Ce type aurait été disjoint des fonctions et aurait nécessité un invoqueur spécialisé : `continue`. Le précédent exemple se serait donc écrit :

```
k(call/cc (lambda (k) (+ 1 (continue k 2))))    → 2
```

Il est toujours possible de transformer une continuation en une fonction en l'enveloppant dans une abstraction `(lambda (v) (continue k v))`. D'aucuns, par contre, apprécient que soient marqués syntaxiquement les appels aux continuations qui provoquent des ruptures de contrôle.

L'ultime difficulté avec `call/cc` est que son vrai nom est `call-with-current-continuation`, ce qui n'améliore pas la situation. Réécrivons donc notre exemple fétiche pour employer `call/cc` :

```
(define (find-symbol id tree)
  (call/cc
   (lambda (exit)
     (define (find tree)
       (if (pair? tree)
           (or (find (car tree))
               (find (cdr tree)) )
           (if (eq? tree id) (exit #t) #f) ) )
     (find tree) ) ) )
```

La continuation d'appel de la fonction `find-symbol` est capturée par `call/cc`, reifiée en une fonction unaire qui est liée à la variable `exit`. Lorsque le symbole est trouvé, l'échappement est déclenché par appel de la fonction `exit` (qui ne retourne jamais).

---

8. Les schemeurs avertis remarqueront qu'il suffit que l'argument de `call/cc` accepte d'être appelé avec un unique argument. On peut ainsi écrire `(call/cc list)`.

9. De *res fecire* littéralement « rendre chose » c'est-à-dire matérialiser !

La durée de vie illimitée des continuations n'est pas mise en évidence dans cet exemple puisque la continuation n'est utilisée que pendant l'invocation de l'argument de `call/cc`, c'est-à-dire pendant sa durée de vie dynamique. C'est ce qu'illustre le programme suivant :

```
(define (fact n)
  (let ((r 1)(k 'void))
    (call/cc (lambda (c) (set! k c) 'void))
    (set! r (* r n))
    (set! n ( - n 1))
    (if (= n 1) r (k 'recurse)) ) )
```

La continuation réifiée en `c` et stockée en `k` correspond à :

```
k = (lambda (u)
      (set! r (* r n))
      (set! n ( - n 1))
      (if (= n 1) r (k 'recurse)) )
```

r→ 1
k≡ *k*
n

Cette continuation k apparaît elle-même comme valeur de la variable `k` qu'elle enclôt. La récursion, qui nécessite toujours quelque chose de cyclique quelque part, est donc assurée par appel à k, qui se réinvoque jusqu'à ce que `n` atteigne le seuil fatidique. Le tout, bien sûr, calcule la factorielle.

On voit sur cet exemple que la continuation k est utilisée en dehors de sa durée de vie dynamique qui est égale au temps pendant lequel le corps de l'argument de `call/cc` est évalué. Sur cette base, de nombreuses autres variantes peuvent être imaginées, comme de soumettre l'identité à `call/cc` :

```
(define (fact n)
  (let ((r 1))
    (let ((k (call/cc (lambda (c) c))))
      (set! r (* r n))
      (set! n ( - n 1))
      (if (= n 1) r (k k)) ) ) )
```

Il faut que l'appel récursif soit maintenant l'auto-application `(k k)` car la variable `k` est liée à nouveau à chaque appel de la continuation à la valeur de l'argument qui lui est fournie. La continuation peut être représentée par :

```
(lambda (u)
  (let ((k u))
    (set! r (* r n))
    (set! n ( - n 1))
    (if (= n 1) r (k k)) ) )
```

r→ 1
n

Conférer une durée de vie illimitée aux continuations rend leur implantation plus délicate et généralement plus coûteuse (voir cependant [CHO88, HDB90, Mat92]) puisque l'on doit abandonner le modèle d'évaluation en pile pour adopter un modèle en arbre. En effet, lorsque les continuations ont une durée de vie dynamique, on peut les qualifier d'échappements puisque leur seul but est de s'échapper d'un calcul. S'échapper, c'est esquiver des calculs qui restaient à effectuer afin d'imposer une valeur finale à une forme en cours d'évaluation. Un calcul commence lorsque l'évaluation d'une forme démarre et il paraît simple de déterminer quand il se termine.

En présence de continuations à durée de vie dynamique, un calcul a toujours une fin décelable. Avec des continuations à durée de vie illimitée, ce n'est plus aussi simple. Considérez l'expression `(call/cc ...)` de la factorielle précédente, cette expression retourne plusieurs fois un résultat. Le fait qu'une forme puisse retourner multiplement des valeurs [10] implique que la borne finale de son calcul n'est pas nécessairement l'instant où elle retourne un résultat.

La fonction `call/cc` est très puissante et permet en quelque sorte de manipuler le temps. S'échapper, c'est accélérer le temps jusqu'à provoquer l'arrivée d'un événement prévu mais encore lointain. Une fois échappée, une continuation à durée de vie illimitée permet de revenir encore en cet état. Ce n'est pas un moyen alambiqué de boucler car, les valeurs des variables ayant en général changé, revenir en un point de programme déjà connu force le calcul dans une nouvelle voie.

La fonction `call/cc` peut être comparée à l'infâme et si décrié `goto` des langages impératifs mais ne lui est pas équivalente car une continuation ne peut que retourner en un point où l'on est déjà passé et dont on a pris la continuation alors que `goto` permet de se brancher en un point où l'on n'est jamais encore passé !

Au début, la fonction `call/cc` est assez délicate à manipuler car la continuation est unaire comme l'est l'argument de `call/cc`. La règle de réécriture suivante permet d'appréhender sous un autre angle l'effet de `call/cc` :

$_k$`(call/cc` ϕ`)` $\rightarrow$ $_k$`(`ϕ k`)`

Dans ce qui précède, k représente la continuation d'appel à `call/cc` et ϕ une fonction unaire quelconque. `call/cc` procède à la réification de la continuation k, une entité de l'implantation, en une valeur du langage qui peut figurer en deuxième terme d'une application. Notez que la continuation courante quand ϕ est invoquée est toujours k. On peut donc ne pas utiliser la continuation fournie en argument et se reposer sur la même continuation implicitement, comme le montre l'extrait suivant :

```
(call/cc (lambda (k) 1515)) → 1515
```

Certains peuvent regretter ce fait et imposer, comme un père sévère, que lorsque l'on « prend » la continuation, on la prend totalement et on la retire donc à l'évaluateur sous-jacent. Il faut ainsi utiliser cette continuation pour retourner une valeur :

```
(call/cc (lambda (k) (k 1615))) → 1615
```

Dans ce monde hypothétique, `(k 1615)` serait évaluée avec une continuation semblable à un trou noir quelque chose comme $(\lambda u.\bullet)$ qui absorberait la valeur qui lui serait rendue sans émettre d'information ; il n'y aurait donc pas de calcul postérieur ; votre clavier se dissoudrait dans l'éther ; vous n'existeriez plus, l'Olympe non plus ; et ce bouquin encore moins. De grâce, dans un tel monde, n'oubliez pas d'invoquer votre continuation !

3.1.6 Protection

Un dernier effet existe qui est relié aux continuations, il s'agit de la forme spéciale `unwind-protect`. Ce nom provient de l'âge où l'implantation d'un trait imposait son

10. Retourner multiplement des valeurs n'est pas retourner des valeurs multiples (comme en COMMON LISP).

nom [11] et suggère son emploi. La forme spéciale `unwind-protect` a pour syntaxe :

`(unwind-protect` *forme*
formes-de-nettoyage `)`

Le terme *forme* est évalué en premier, sa valeur deviendra celle de la forme `unwind-protect` toute entière. Lorsque cette valeur est obtenue, les *formes de nettoyage* (ou, en grand-breton, *cleanup forms*) sont évaluées (pour leur effet) avant que ne soit finalement rendue cette valeur. L'effet est quasiment celui d'un `prog1` en COMMON LISP ou d'un `begin0` comme dans certains Scheme, c'est-à-dire l'évaluation d'une série de formes (comme `progn` ou `begin`) tout en rendant la valeur de la première de ces formes. La forme spéciale `unwind-protect` garantit que les formes de nettoyage seront toujours évaluées quelle que soit la façon dont le calcul de *forme* s'achève, même si c'est par un échappement. Ainsi :

COMMON LISP
```
(let ((a 'on))
  (cons (unwind-protect (list a)
          (setq a 'off) )
        a ) )   → ((on) . off)
(block foo
  (unwind-protect (return-from foo 1)
     (print 2) ) )  → 1 ; and prints 2
```

Cette forme est intéressante dans le cas d'un sous-sytème où l'on veut assurer qu'une certaine propriété sera restaurée, quelle que soit l'issue de ce sous-système. L'exemple classique est celui du traitement d'un fichier dont on veut assurer qu'il sera fermé même si le traitement erre. Un autre exemple, plus intéressant, concerne la simulation de `catch` par `block`. L'emploi simultané de `block` désynchronisait le maintien de la variable `*active-catchers*`. On corrigera ce défaut grâce à un judicieux `unwind-protect` bien placé :

```
(define-syntax catch
  (syntax-rules
    ((catch tag . body)
     (let ((saved-catchers *active-catchers*))
       (unwind-protect
         (block label
           (set! *active-catchers*
                 (cons (cons tag (lambda (x) (return-from label x)))
                       *active-catchers* ) )
           . body )
         (set! *active-catchers* saved-catchers) ) ) ) ) )
```

Quelle que soit l'issue du calcul qu'effectue `body`, la liste des formes `catch` actives sera toujours remise à jour en fin de calcul. Les formes `block` peuvent donc être librement utilisées conjointement avec les formes `catch`. La simulation n'est toujours pas parfaite car la variable `*active-catchers*` devrait être privée à `catch` et `throw` (ou plutôt à leurs expansions). Dans l'état actuel, on peut l'altérer, accidentellement ou intentionnellement, et ainsi détruire la belle ordonnance de ces macros.

La forme `unwind-protect` protège le calcul d'une forme en assurant que, lorsque celui-ci s'achèvera, un certain traitement sera effectué. La forme `unwind-protect` est donc grandement couplée à la détection de la terminaison d'un calcul et, partant, aux

---

11. Proviennent de ce même âge les noms de `car` et `cdr` fortement liés à l'implantation de Lisp 1.5. Nous ne rappellerons pas le sens des acronymes `car` et `cdr` pour lesquels on consultera les notes en bas de page des bons auteurs.

continuations à durée de vie dynamique. Pour les raisons débattues plus haut, `unwind-protect` se marie mal avec `call/cc` et les continuations à durée de vie illimitée [12].

Comme souvent souligné auparavant, le sens de ces constructions n'est pas précis du tout. Soulevons juste quelques interrogations relatives aux valeurs des formes suivantes :

```
(block foo
  (unwind-protect (return-from foo 1)
    (return-from foo 2) ) )                                  → ?
(catch 'bar
  (block foo
    (unwind-protect (return-from foo (throw 'bar 1))
      (throw 'no-associated-catch (return-from foo 2)) ) ) ) → ?
```

Toutes ces imprécisions militent pour une formalisation soignée et précise de ces concepts. Il faut cependant observer le statut quelque peu particulier des formes de nettoyage et notamment leur continuation. On peut qualifier celle-ci de *flottante* car elle n'est pas connue statiquement et peut correspondre à un échappement en cours. Considérez par exemple :

```
(block bar
   (unwind-protect (return-from bar 1)
      (block foo π2 ) ) )
```

La forme de nettoyage capture la continuation correspondant à l'échappement lexical vers `bar`. Si la forme de nettoyage retourne une valeur, cet échappement vers `bar` sera repris. Si la forme de nettoyage s'échappe elle-même, alors sa propre continuation sera abandonnée. Ces phénomènes seront précisés lorsque l'on détaillera l'implantation.

D'autres formes de contrôle existent, et notamment en COMMON LISP, où la forme antédiluvienne `prog` s'est métamorphosée en `tagbody` et `go` qui peuvent être simulées grâce à `labels` et `block`. **[☞ Ex. 3.3]** On retiendra que dans un monde à continuations à durée de vie dynamique, `block`, `return-from` et `unwind-protect` forment une base des formes spéciales qu'on y trouve usuellement. Similairement, dans un monde Scheme, `call/cc` est tout ce qui est nécessaire. Il est clair que l'on ne peut simuler directement `call/cc` avec des continuations à durée de vie dynamique. L'inverse est faisable mais nécessite de brider `call/cc`, qui est par trop puissant. La méthode pour ce faire sera claire lorsque nous aurons explicité l'interprète à continuations.

Protection et variables dynamiques

Certaines implantations de Scheme procurent des variables dynamiques un peu différentes des variations que nous avons auparavant présentées et dont l'implantation repose sur la présence de la forme `unwind-protect` ou similaire. L'idée tient en une sorte de squat lexical. Ces variables dynamiques sont introduites par la forme `fluid-let` :

```
(fluid-let ((x α))   ≡ (let ((tmp x))
   β... )                 (set! x α)
```

12. Une description opérationnelle d'une extension d'`unwind-protect` pour Scheme (nommée `dynamic-wind` figure en [FWH92] ; voir aussi [Que93c].

```
(unwind-protect
  (begin β...)
  (set! x tmp) ) )
```

Pendant le calcul de β, la variable `x` a pour valeur la valeur de α, la valeur précédente de `x` est sauvegardée dans une variable locale *tmp* et est restaurée en fin de calcul. Cette forme implique l'existence d'une liaison lexicale à emprunter. C'est usuellement une liaison globale, ce qui rend alors la variable visible de partout. Si par contre c'est une liaison locale qui est empruntée, alors, à la différence des variables dynamiques de COMMON LISP, celles-ci ne seront accessibles que dans le corps de la forme `fluid-let` et seulement dans celui-ci avec possiblement des effets pernicieux sur les copropriétaires de cette variable. D'autre part la forme `unwind-protect` et les continuations ne font pas vraiment bon ménage aussi cette forme est-elle plus délicate d'emploi que les variables dynamiques à la COMMON LISP.

3.2 Les acteurs d'un calcul

Un calcul est donc maintenant constitué de trois éléments : une expression, un environnement et une continuation. Si le futur immédiat est l'évaluation de l'expression dans l'environnement, le but est de retourner une valeur au futur lointain qu'est la continuation. Comme tout futur, celui-ci contient l'avenir tout entier y compris les altérations que subira dans le futur cet avenir même.

Nous allons donc décrire un nouvel interprète qui met en relief toutes les continuations nécessaire à tous les niveaux. Parce que les continuations sont usuellement représentées par des blocs en pile ou tas, nous utiliserons des objets pour les représenter au sein de l'interprète.

3.2.1 Rappel sur les objets

Il n'est pas dans le but de cette section de présenter un système d'objets dans tous ses détails **[☞ p. 395]** et toutes ses impressionnantes qualités. Le but n'est de présenter qu'un ensemble de trois macros associées à quelques règles de dénomination. Cet ensemble n'est que l'essence d'un système d'objet et est indépendant de toute réalisation de celui-ci. Nous avons choisi d'utiliser des objets pour ce chapitre afin de suggérer l'implantation des continuations. Les objets munis de leur champs évoquent commodément les blocs d'activation qui servent à implanter des langages. La notion d'héritage, qui sera, ici, mise en œuvre de façon fort élémentaire, permet de factoriser certains effets et partant de diminuer la taille de l'interprète que nous présenterons.

Nous supposons que nos lecteurs sont au fait de la philosophie, de la terminologie et de la pragmatique relative aux objets et nous nous contenterons d'énoncer les quelques tours linguistiques que nous utiliserons.

Les objets sont regroupés en *classes*, les objets d'une même classe répondent aux mêmes *méthodes*, enfin, l'envoi de message s'effectue grâce aux *fonctions génériques* popularisées par Common Loops [BKK+86], CLOS [BDG+88] et TEΛOΣ [PNB93]. L'intérêt du style de programmation par objets est que nous pourrons greffer autour d'un noyau d'évaluateur les différentes formes spéciales ou fonctions primitives de

manipulation du contrôle. L'inconvénient propre à ce style est qu'une vision globale est plus difficile à acquérir du fait de la dissémination des méthodes.

Définition d'une classe

Une classe est définie par `define-class` :

```
(define-class classe super-classe
        ( champs... ) )
```

Cette forme définit une classe ayant pour nom *classe*, héritant des champs et méthodes de *super-classe* et possédant, outre les champs de *super-classe*, les champs supplémentaires *champs*.... Que la classe soit créée permet d'utiliser une tétra-foultitude de fonctions associées. La fonction de nom `make-`*`classe`* permet de créer des objets appartenant à *classe*, cette fonction a autant d'arguments, énoncés dans le même ordre, que la classe a de champs. Les accesseurs en lecture aux champs de ces objets ont un nom préfixés par le nom de la classe et suffixés par le nom du champ ; un tiret les sépare. Les accesseurs en écriture préfixent le nom de l'accesseur en lecture par `set-` et le suffixent par un point d'exclamation. Les accesseurs en écriture ont une valeur de retour non spécifiée. Enfin le prédicat *classe*`?` permet de tester l'appartenance à *classe*.

La racine du graphe d'héritage est la classe `Object` qui ne possède aucun champ.

La définition suivante :

```
(define-class continuation Object (k))      ; exemple
```

permet que soient utilisées les fonctions :

```
(make-continuation k)           ; création
(continuation-k c)              ; accesseur en lecture
(set-continuation-k! c k)       ; accesseur en écriture
(continuation? o)               ; prédicat de reconnaissance
```

Définition d'une fonction générique

Une fonction générique est définie par :

```
(define-generic (fonction variables)
    [ traitement-par-défaut...] )
```

Cette forme définit une fonction générique nommée *fonction*, dont le traitement par défaut, lorsqu'aucune méthode appropriée n'est trouvée, est spécifié par *traitement-par-défaut*. La liste de variables est une liste de variables normales sauf qu'elle spécifie aussi la variable servant à la *discrimination*, celle-ci apparaît cernée de parenthèses.

```
(define-generic (invoke (f) v* r k)
  (wrong "not a function" f r k) )
```

Cette forme définit la fonction générique `invoke` qui pourra être enrichie de méthodes ultérieurement. Cette fonction a quatre variables dont la première est la variable de discrimination : `f`. Si dans un appel à `invoke`, aucune méthode appropriée n'est définie sur la classe de l'argument, alors le traitement par défaut sera d'invoquer `wrong`.

Définition d'une méthode

On étoffe une fonction générique de méthodes spécifiques grâce à `define-method` :

```
(define-method (fonction variables)
      traitement... )
```

Pareillement à la forme `define-generic`, la liste de variables spécifie la classe sur laquelle on définit la méthode. Cette classe apparaît avec la variable de discrimination entre parenthèses. Par exemple, la forme suivante définit une méthode sur la classe `primitive` :

```
(define-method (invoke (ff primitive) vv* rr kk)
  ((primitive-address ff) vv* rr kk) )
```

Voici donc réuni le cocktail de traits que nous utiliserons. Une implantation de ce système d'objets sera vue au chapitre 11 mais quelle qu'elle sera, nous n'en utilisons ici que les caractéristiques les plus connues et les moins sujettes à caution.

3.2.2 L'interprète à continuations

La fonction `evaluate` a maintenant trois arguments, l'expression, l'environnement et la continuation. La fonction `evaluate` commence par effectuer l'analyse syntaxique de l'expression et détermine ainsi le traitement approprié. Chacun de ces traitements est individualisé en une fonction particulière. Avant de nous lancer dans ce nouvel interprète, indiquons les règles que nous allons suivre pour nommer nos variables, sachant qu'une variable suffixée par une étoile représente une liste de ce que représente la même variable non étoilée.

`e`, `et`, `ec`, `ef` ...	expression, forme
`r` ...	environnement
`k`, `kk` ...	continuation
`v` ...	valeur (entier, paire, fermeture, etc.)
`f` ...	fonction
`n` ...	identificateur

Voici donc l'interprète (qui suppose que tout ce qui est atomique et non reconnu comme une variable est une citation implicite) :

```
(define (evaluate e r k)
  (if (atom? e)
    (cond ((symbol? e) (evaluate-variable e r k))
          (else        (evaluate-quote e r k)) )
    (case (car e)
     ((quote)  (evaluate-quote (cadr e) r k))
     ((if)     (evaluate-if (cadr e) (caddr e) (cadddr e) r k))
     ((begin)  (evaluate-begin (cdr e) r k))
     ((set!)   (evaluate-set! (cadr e) (caddr e) r k))
     ((lambda) (evaluate-lambda (cadr e) (cddr e) r k))
     (else     (evaluate-application (car e) (cdr e) r k)) ) ) )
```

L'interprète sera bâti sur trois fonctions : `evaluate`, `invoke` et `resume`. Seules les deux dernières sont génériques et savent invoquer les objets applicables ou traiter les continuations quelle que soit leur nature. L'interprète tout entier ne va être qu'un jeu de passe-passe entre ces trois fonctions. Deux autres fonctions génériques seront cependant utiles pour déterminer ou modifier la valeur d'une variable : `lookup` et `update!`.

```
(define-generic (invoke (f) v* r k)
  (wrong "not a function" f r k) )
```

```
(define-generic (resume (k continuation) v)
  (wrong "Unknown continuation" k) )
(define-generic (lookup (r environment) n k)
  (wrong "not an environment" r n k) )
(define-generic (update! (r environment) n k v)
  (wrong "not an environment" r n k) )
```

Toutes les entités qui seront manipulées dériveront de trois classes virtuelles :

```
(define-class value Object ())
(define-class environment Object ())
(define-class continuation Object (k))
```

Les classes des valeurs manipulées par le langage que l'on définit hériteront de `value` ; les environnements et les continuations manipulées par le langage de définition hériteront de `environment` et `continuation` respectivement.

3.2.3 Citation

La forme spéciale de citation est toujours la plus simple et consiste à rendre à la continuation courante le terme cité :

```
(define (evaluate-quote v r k)
  (resume k v) )
```

3.2.4 Alternative

L'alternative met en jeu deux continuations : celle qui est courante et celle qui consiste à attendre la valeur de la condition afin de déterminer quelle branche de l'alternative choisir. Pour représenter cette continuation, nous définirons une classe appropriée. Lorsque la condition est évaluée, l'alternative choisira entre la branche vraie ou fausse ; ces deux branches doivent donc être mémorisées ainsi que l'environnement des variables nécessaires à leur évaluation. Le résultat de l'une ou l'autre de ces branches devra être retourné à la continuation originale de l'alternative qui doit donc, aussi, être mémorisée. On obtient donc :

```
(define-class if-cont continuation (et ef r))
(define (evaluate-if ec et ef r k)
  (evaluate ec r (make-if-cont k
                               et
                               ef
                               r )) )
(define-method (resume (k if-cont) v)
  (evaluate (if v (if-cont-et k) (if-cont-ef k))
            (if-cont-r k)
            (if-cont-k k) ) )
```

L'alternative décide donc d'évaluer sa condition `ec`, dans l'environnement courant `r` mais avec une nouvelle continuation fabriquée en y mettant tous les ingrédients nécessaires à la poursuite du calcul. Lorsque le calcul de la condition s'achèvera par un appel à `resume`, le calcul de l'alternative reprendra, s'enquerra de la valeur de la condition `v` pour déterminer la branche à choisir mais, de toutes façon l'évaluera dans l'environnement sauvegardé et avec la continuation initiale de l'alternative [13].

13. Les passionnés d'implantation noteront que `make-if-cont` peut être vu comme une forme empi-

3.2.5 Séquence

La séquence met aussi en jeu deux continuations.

```
(define-class begin-cont continuation (e* r))
(define (evaluate-begin e* r k)
  (if (pair? e*)
    (if (pair? (cdr e*))
      (evaluate (car e*) r (make-begin-cont k e* r))
      (evaluate (car e*) r k) )
    (resume k empty-begin-value) ) )
(define-method (resume (k begin-cont) v)
  (evaluate-begin (cdr (begin-cont-e* k))
                  (begin-cont-r k)
                  (begin-cont-k k) ) )
```

Les cas de `(begin)` et `(begin` π`)` sont simples. Lorsque la forme `begin` comporte plusieurs termes, il faut évaluer le premier des termes en lui fournissant une nouvelle continuation `(make-begin-cont k e* r)` qui, quand il retournera une valeur par `resume`, déclenchera la méthode sur `begin-cont`. Cette continuation ignorera la valeur retournée `v` et relancera le calcul des autres formes présentes dans `begin` [14].

3.2.6 Environnement des variables

Les valeurs des variables sont enregistrées dans un environnement. Celui-ci sera aussi représenté comme un objet :

```
(define-class null-env environment ())
(define-class full-env environment (others name))
(define-class variable-env full-env (value))
```

Deux sortes d'environnements sont nécessaires : l'environnement vide pour initialiser les calculs et les instances de `variable-env` correspondant à des environnements non vides. Ces derniers mémorisent une liaison, c'est-à-dire un nom et une valeur ; la suite de l'environnement se trouve dans le champ `others`. Cette organisation même si elle use d'objets est fonctionnellement semblable à la A-liste sauf qu'elle ne consomme qu'un objet à trois champs et non deux paires pointées.

Rechercher la valeur d'une variable est donc :

```
(define (evaluate-variable n r k)
  (lookup r n k) )
(define-method (lookup (r null-env) n k)
  (wrong "Unknown variable" n r k) )
(define-method (lookup (r full-env) n k)
  (lookup (full-env-others r) n k) )
(define-method (lookup (r variable-env) n k)
  (if (eqv? n (variable-env-name r))
    (resume k (variable-env-value r))
    (lookup (variable-env-others r) n k) ) )
```

---

lant `et`, puis `ef` et enfin `r` sur la pile d'exécution dont la partie inférieure est représentée par `k`. Réciproquement `(if-cont-et k)` et les autres désempilent ces mêmes valeurs.

14. Les lecteurs attentifs auront remarqué la forme `(cdr (begin-cont-e* k))` présente dans la méthode `resume`. Il aurait été équivalent de fabriquer directement dans `evaluate-begin` : `(make-begin-cont k (cdr e*) r)`. La raison est qu'en cas d'erreur, l'analyse de la continuation permet de savoir quelle était l'expression en cours.

La fonction générique `lookup` scrute l'environnement jusqu'à trouver la bonne liaison. La valeur qui est alors déterminée est soumise à la continuation grâce à la fonction générique `resume`.

Modifier une variable participe du même processus :

```
(define-class set!-cont continuation (n r))
(define (evaluate-set! n e r k)
  (evaluate e r (make-set!-cont k n r)) )
(define-method (resume (k set!-cont) v)
  (update! (set!-cont-r k) (set!-cont-n k) (set!-cont-k k) v) )
(define-method (update! (r null-env) n k v)
  (wrong "Unknown variable" n r k) )
(define-method (update! (r full-env) n k v)
  (update! (full-env-others r) n k v) )
(define-method (update! (r variable-env) n k v)
  (if (eqv? n (variable-env-name r))
    (begin (set-variable-env-value! r v)
           (resume k v) )
    (update! (variable-env-others r) n k v) ) )
```

Il est cependant nécessaire d'introduire une continuation particulière car l'évaluation d'une affectation se fait en deux temps : calcul de la valeur à affecter puis modification de la variable concernée. C'est la classe `set!-cont` qui implante cette nouvelle sorte de continuation, la méthode `resume` adaptée se contente d'appeler l'environnement afin de le mettre à jour.

3.2.7 Fonctions

La création d'une fonction est un processus simple dont s'occuppe `make-function`.

```
(define-class function value (variables body env))
(define (evaluate-lambda n* e* r k)
  (resume k (make-function n* e* r)) )
```

Ce qui est à peine plus compliqué est de décrire comment sont invoquées les fonctions. On notera le `progn` (ou `begin`) implicite des corps de fonction :

```
(define-method (invoke (f function) v* r k)
  (let ((env (extend-env (function-env f)
                         (function-variables f)
                         v* )))
    (evaluate-begin (function-body f) env k) ) )
```

Il peut sembler aberrant que les fonctions prennent en argument l'environnement courant `r` alors qu'elles ne s'en servent pas. Nous l'avons laissé pour deux raisons : d'une part il y a souvent un registre dédié à l'environnement dans les implantations qui, comme tout registre, est disponible. D'autre part, certaines fonctions (et nous en verrons lorsque nous parlerons de reflexivité) peuvent influer sur l'environnement lexical courant comme par exemple les fonctions de mise au point.

La fonction suivante étend un environnement. On n'a pas besoin de tester s'il y a assez de valeurs ou assez de noms car les fonctions l'ont déjà vérifié.

```
(define (extend-env env names values)
  (cond ((and (pair? names) (pair? values))
         (make-variable-env
```

```
      (extend-env env (cdr names) (cdr values))
      (car names)
      (car values) ) )
    ((and (null? names) (null? values)) env)
    ((symbol? names) (make-variable-env env names values))
    (else (wrong "Arity mismatch")) ) )
```

Il ne reste plus qu'à indiquer comment se définit une application fonctionnelle, sachant que les fonctions sont invoquées sur des arguments réunis en une liste.

```
(define-class evfun-cont continuation (e* r))
(define-class apply-cont continuation (f r))
(define-class argument-cont continuation (e* r))
(define-class gather-cont continuation (v))
(define (evaluate-application e e* r k)
  (evaluate e r (make-evfun-cont k e* r)) )
(define-method (resume (k evfun-cont) f)
  (evaluate-arguments (evfun-cont-e* k)
                      (evfun-cont-r k)
                      (make-apply-cont (evfun-cont-k k)
                                       f
                                       (evfun-cont-r k) ) ) )
(define (evaluate-arguments e* r k)
  (if (pair? e*)
    (evaluate (car e*) r (make-argument-cont k e* r))
    (resume k no-more-arguments) ) )
(define no-more-arguments '())
(define-method (resume (k argument-cont) v)
  (evaluate-arguments (cdr (argument-cont-e* k))
                      (argument-cont-r k)
                      (make-gather-cont (argument-cont-k k) v)) )
(define-method (resume (k gather-cont) v*)
  (resume (gather-cont-k k) (cons (gather-cont-v k) v*)) )
(define-method (resume (k apply-cont) v)
  (invoke (apply-cont-f k)
          v
          (apply-cont-r k)
          (apply-cont-k k) ) )
```

La technique est un peu déroutante de prime abord. L'évaluation a lieu de la gauche vers la droite, le terme fonctionnel est donc évalué en premier avec une continuation de classe `evfun-cont`. Lorsque cette continuation prend le contrôle, elle procède à l'évaluation des arguments en laissant une continuation qui, lorsqu'ils auront tous été calculés, leur appliquera la fonction. Pendant l'évaluation des arguments, sont laissées des continuations de classe `gather-cont` dont le rôle sera d'agréger les arguments en une liste.

Donnons un exemple, en examinant les différentes continuations qui apparaissent lors du calcul de `(cons foo bar)`, en supposant que `foo` et `bar` vaillent respectivement 33 et -77. Pour illustrer ce calcul, les continuations seront dessinées comme des piles horizontales ; k, resp. r, sera la continuation, resp. l'environnement courant. On notera *cons* la valeur de la variable gobale `cons`, c'est-à-dire la fonction d'allocation de paires pointées.

evaluate `(cons foo bar)` r k

evaluate `cons` r evfun-cont `(foo bar)` r k

resume cons				evfun-cont `(foo bar)` *r* *k*
evaluate-arguments `(foo bar)` *r*				apply-cont *cons* *k*
evaluate `foo` *r*		argument-cont `(foo bar)` *r*		apply-cont *cons* *k*
resume 33		argument-cont `(foo bar)` *r*		apply-cont *cons* *k*
evaluate-arguments `(bar)` *r*			gather-cont 33	apply-cont *cons* *k*
evaluate `bar` *r*		argument-cont `()` *r*	gather-cont 33	apply-cont *cons* *k*
resume -77		argument-cont `()` *r*	gather-cont 33	apply-cont *cons* *k*
evaluate-arguments `()` *r*	gather-cont -77		gather-cont 33	apply-cont *cons* *k*
resume ()	gather-cont -77		gather-cont 33	apply-cont *cons* *k*
resume (-77)			gather-cont 33	apply-cont *cons* *k*
resume (33 -77)				apply-cont *cons* *k*
invoke cons (33 -77)				*k*

3.3 Initialisation de l'interprète

Avant de nous plonger dans les mystérieux arcanes des formes de contrôle, indiquons de suite les quelques détails qui autorisent l'exécution de cet interprète. Cette section est assez semblable à la section 1.7. **[☞ p. 23]** Il faut en premier habiller l'interprète de quelques variables bien choisies et pour cela définir les macros pour enrichir l'environnement global.

```
(define-syntax definitial
  (syntax-rules ()
    ((definitial name)
     (definitial name 'void) )
    ((definitial name value)
     (begin (set! r.init (make-variable-env r.init 'name value))
            'name ) ) ) )
(define-class primitive value (name address))
(define-syntax defprimitive
  (syntax-rules ()
    ((defprimitive name value arity)
     (definitial name
       (make-primitive
        'name (lambda (v* r k)
                (if (= arity (length v*))
                    (resume k (apply value v*))
                    (wrong "Incorrect arity" 'name v*) ) ) ) ) ) ) )
(define r.init (make-null-env))
(defprimitive cons cons 2)
(defprimitive car car 1)
```

Les primitives qui sont créées doivent pouvoir être invoquées, tout comme les fonctions de l'utilisateur, par `invoke`. Ces primitives comportent deux champs. Le premier est là pour faciliter la mise au point car il indique le nom original de la primitive. Ce n'est bien sûr qu'une indication, car vous pouver lier la primitive à d'autres noms globaux si vous le souhaitez [15]. Le second champ contient l'« adresse » de la

15. C'est la même astuce qui permet dans de nombreux systèmes d'obtenir l'effet suivant : `(begin (set! foo car)(set! car 3) foo)` a un résultat qui s'imprime `#<car>` où le nom évocateur de la primitive suit l'objet fonctionnel.

primitive, c'est-à-dire quelque chose d'exécutable par la machine sous-jacente. Une primitive est donc lancée par :

```
(define-method (invoke (f primitive) v* r k)
  ((primitive-address f) v* r k) )
```

Enfin, pour démarrer l'interprète et jouir de ses talents, on définira de manière semblable à `null-env` une continuation initiale qui imprimera les résultats qu'on lui fournit.

```
(define-class bottom-cont continuation (f))
(define-method (resume (k bottom-cont) v)
  ((bottom-cont-f k) v) )
(define (chapter3-interpreter)
  (define (toplevel)
    (evaluate (read)
              r.init
              (make-bottom-cont 'void display) )
    (toplevel) )
  (toplevel) )
```

On observera que l'interprète tout entier pourrait être facilement réécrit dans un vrai langage d'objets comme Smalltalk [GR83] et ainsi bénéficier de son fameux fouineur ou *browser* et de son metteur au point. Il ne resterait plus qu'à ajouter ce qu'il faut pour avoir des tas de petites fenêtres partout.

3.4 Implantation des formes de contrôle

Commençons par la plus puissante des formes de contrôle, à savoir `call/cc`, qui est paradoxalement la plus simple à implanter si l'on mesure la simplicité en nombre de lignes. Le style de programmation par objets que nous avons suivi et l'explicitation des continuations va rendre la tâche presque totalement triviale.

3.4.1 Implantation de `call/cc`

La fonction `call/cc` prend la continuation courante `k`, la transforme en un objet que l'on peut soumettre à `invoke`, enfin applique le premier argument, une fonction unaire, dessus. C'est ce qu'exprime fort élégamment ce qui suit.

```
(definitial call/cc
 (make-primitive
  'call/cc
  (lambda (v* r k)
    (if (= 1 (length v*))
        (invoke (car v*) (list k) r k)
        (wrong "Incorrect arity" 'call/cc v*) ) ) ) )
```

Malgré le faible nombre de lignes, explicitons un peu. Bien qu'étant une fonction, `call/cc` est définie par `definitial` car elle a intensément besoin d'avoir accès à sa continuation. La variable `call/cc` (nous sommes ici dans un Lisp_1) est donc liée à un objet de la classe `primitive`. Le protocole d'appel de ces objets est de les associer à une « adresse » qui est représentée dans le Lisp de définition par une fonction

de signature `(lambda (v* r k) ...)` [16]. Enfin le premier argument est appliqué à la continuation. Notez que la continuation a été livrée telle quelle. Nous nous empressons donc, puisqu'elle va être possiblement soumise à `invoke`, de conférer à cette dernière la méthode ad-hoc.

```
(define-method (invoke (f continuation) v* r k)
  (if (= 1 (length v*))
      (resume f (car v*))
      (wrong "Continuations expect one argument" v* r k) ) )
```

3.4.2 Implantation de `catch`

La forme `catch` est intéressante à implanter car les mécanismes qu'elle met en œuvre sont assez différents de ceux de `block` que nous présenterons plus loin. Comme d'habitude, on ajoutera les clauses nécessaires à `evaluate` pour analyser les formes `catch` et `throw` :

```
...
((catch) (evaluate-catch (cadr e) (cddr e) r k))
((throw) (evaluate-throw (cadr e) (caddr e) r k)) ...
```

On a ici pris l'option de faire de `throw` une forme spéciale et non une fonction prenant une suspension, pour mieux simuler COMMON LISP. La forme spéciale `catch` se définit ainsi :

```
(define-class catch-cont continuation (body r))
(define-class labeled-cont continuation (tag))
(define (evaluate-catch tag body r k)
  (evaluate tag r (make-catch-cont k body r)) )
(define-method (resume (k catch-cont) v)
  (evaluate-begin (catch-cont-body k)
                  (catch-cont-r k)
                  (make-labeled-cont (catch-cont-k k) v) ) )
```

Il est maintenant apparent que `catch` évalue son premier argument, le lie à sa continuation en créant un bloc étiqueté, puis poursuit sa tâche en évaluant séquentiellement son corps. Lorsqu'une valeur est retournée au bloc étiqueté, celui-ci s'efface et, tout simplement, la transmet. La forme `throw` va mieux utiliser ce bloc étiqueté.

```
(define-class throw-cont continuation (form r))
(define-class throwing-cont continuation (tag cont))
(define (evaluate-throw tag form r k)
  (evaluate tag r (make-throw-cont k form r)) )
(define-method (resume (k throw-cont) tag)
  (catch-lookup k tag k) )
(define-method (resume (k throw-cont) tag)
  (catch-lookup k tag k) )
(define-generic (catch-lookup (k) tag kk)
  (wrong "Not a continuation" k tag kk) )
(define-method (catch-lookup (k continuation) tag kk)
  (catch-lookup (continuation-k k) tag kk) )
```

---

16. Il peut sembler tout à fait inutile de donner l'environnement de variables courant aux fonctions primitives. Nous l'avons laissé par souci de ressembler aux implantations réelles où souvent une fonction primitive, lorsqu'elle est appelée, trouve, dans un certain registre, l'environnement. Cette propriété aide à l'écriture de metteurs au point ou de fonctions réflexives.

```
(define-method (catch-lookup (k bottom-cont) tag kk)
  (wrong "No associated catch" k tag kk) )
(define-method (catch-lookup (k labeled-cont) tag kk)
  (if (eqv? tag (labeled-cont-tag k)) ; comparator
      (evaluate (throw-cont-form kk)
                (throw-cont-r kk)
                (make-throwing-cont kk tag k) )
      (catch-lookup (labeled-cont-k k) tag kk) ) )
(define-method (resume (k throwing-cont) v)
  (resume (throwing-cont-cont k) v) )
```

La forme `throw` évalue, en premier, son premier argument puis recherche si une continuation ainsi étiquetée existe. Si ce n'est pas le cas, une erreur est déclenchée sinon la valeur à transmettre est alors calculée et sera transmise à la continuation auparavant découverte (qui ne peut avoir changé). On notera que l'évaluation de la valeur à transmettre a une continuation un peu particulière ; une instance de `throwing-cont`. La raison est que si, d'aventure, une erreur ou effet de contrôle surgissait dans le calcul de cette valeur, le bon contexte serait le contexte courant et non celui qui sera adopté si tout se passe bien. On peut donc écrire :

```
(catch 2
  (* 7 (catch 1
         (* 3 (catch 2
                (throw 1 (throw 2 5)) )) )) )
```

Le résultat est `(* 7 3 5)` et non 5. Cette définition de `throw` peut conduire à détecter des erreurs qui n'en seraient point si `throw` était une fonction :

```
(catch 2 (* 7 (throw 1 (throw 2 3))))
```

Cette forme conduit non à 3 mais à l'erreur `"No associated catch"`.

3.4.3 Implantation de `block`

Deux problèmes sont à résoudre pour implanter les échappements lexicaux. Le premier est de conférer une durée de vie dynamique aux continuations. Le second problème est de donner une portée lexicale aux étiquettes des échappements lexicaux. On utilisera pour ce faire l'environnement lexical qui justement confère des portées appropriées. On définira une nouvelle sorte d'environnement pour y associer étiquettes et continuations.

On ajoutera à la fonction `evaluate`, la clause reconnaissant les formes spéciales `block` et l'on définira tout ce qui suit :

```
(define-class block-cont continuation (label))
(define-class block-env full-env (cont))
(define (evaluate-block label body r k)
  (let ((k (make-block-cont k label)))
    (evaluate-begin body
                    (make-block-env r label k)
                    k ) ) )
(define-method (resume (k block-cont) v)
  (resume (block-cont-k k) v) )
```

Tout est en place maintenant pour `return-from`. On ajoutera donc une clause d'analyse des formes `return-from` au sein d'`evaluate` :

...

```
((block)       (evaluate-block (cadr e) (cddr e) r k))
((return-from) (evaluate-return-from (cadr e) (caddr e) r k)) ...
```

Et l'on définira :

```
(define-class return-from-cont continuation (r label))
(define (evaluate-return-from label form r k)
  (evaluate form r (make-return-from-cont k r label)) )
(define-method (resume (k return-from-cont) v)
  (block-lookup (return-from-cont-r k)
                (return-from-cont-label k)
                (return-from-cont-k k)
                v ) )
(define-generic (block-lookup (r) n k v)
  (wrong "not an environment" r n k v) )
(define-method (block-lookup (r block-env) n k v)
  (if (eq? n (block-env-name r))
      (unwind k v (block-env-cont r))
      (block-lookup (block-env-others r) n k v) ) )
(define-method (block-lookup (r full-env) n k v)
  (block-lookup (variable-env-others r) n k v) )
(define-method (block-lookup (r null-env) n k v)
  (wrong "Unknown block label" n r k v) )
(define-method (resume (k return-from-cont) v)
  (block-lookup (return-from-cont-r k)
                (return-from-cont-label k)
                (return-from-cont-k k)
                v ) )
(define-generic (unwind (k) v ktarget))
(define-method (unwind (k continuation) v ktarget)
  (if (eq? k ktarget) (resume k v)
      (unwind (continuation-k k) v ktarget) ) )
(define-method (unwind (k bottom-cont) v ktarget)
  (wrong "Obsolete continuation" v) )
```

Une fois la valeur à transmettre obtenue, la continuation associée à l'étiquette du `return-from` est recherchée par `block-lookup` dans l'environnement lexical. Si `block-lookup` la trouve, alors on vérifie que la continuation associée est toujours valide en la recherchant dans la continuation courante grâce à la nouvelle fonction `unwind`.

La recherche dans l'environnement lexical d'un bloc ad hoc est effectuée par la fonction générique `block-lookup`. On a défini sur elle les méthodes nécessaires afin qu'elle saute les environnements de variables, de peu d'intérêt pour elle, pour ne s'intéresser qu'aux instances de `block-cont`. Réciproquement, on a étendu la fonction générique `lookup` pour ignorer les instances de `block-cont`. Ces méthodes ont été définies sur la classe virtuelle `full-env` afin qu'elles soient partagées au cas où d'autres classes d'environnement seraient créées.

La fonction générique `unwind`, quant à elle, cherche à transmettre une valeur à une certaine continuation qui doit être vivante, c'est-à-dire que l'on doit trouver dans la continuation courante.

3.4.4 Implantation de `unwind-protect`

C'est la forme la plus complexe à traiter et elle implique de modifier les définitions des formes `catch` et `block` précédentes car elles doivent être adaptées à la présence

de `unwind-protect`. Ceci est un exemple d'un trait dont la seule introduction altère la définition de ce qui existe déjà. Ne pas se servir de `unwind-protect` implique déjà un coût. Ce qui suit est la définition de `unwind-protect`, apparemment peu différente de `prog1`.

```
(define-class unwind-protect-cont continuation (cleanup r))
(define-class protect-return-cont continuation (value))
(define (evaluate-unwind-protect form cleanup r k)
  (evaluate form
            r
            (make-unwind-protect-cont k cleanup r) ) )
(define-method (resume (k unwind-protect-cont) v)
  (evaluate-begin (unwind-protect-cont-cleanup k)
                  (unwind-protect-cont-r k)
                  (make-protect-return-cont
                   (unwind-protect-cont-k k) v ) ) )
(define-method (resume (k protect-return-cont) v)
  (resume (protect-return-cont-k k) (protect-return-cont-value k)) )
```

Il faut maintenant modifier `catch` et `block` pour tenir compte des nettoyages programmés lorsqu'une forme `unwind-protect` est franchie par un échappement. Pour `catch`, on modifiera la définition de `throwing-cont` :

```
(define-method (resume (k throwing-cont) v)              ; ** Modified **
  (unwind (throwing-cont-k k) v (throwing-cont-cont k)) )
(define-class unwind-cont continuation (value target))
(define-method (unwind (k unwind-protect-cont) v target)
  (evaluate-begin (unwind-protect-cont-cleanup k)
                  (unwind-protect-cont-r k)
                  (make-unwind-cont
                   (unwind-protect-cont-k k) v target ) ) )
(define-method (resume (k unwind-cont) v)
  (unwind (unwind-cont-k k)
          (unwind-cont-value k)
          (unwind-cont-target k) ) )
```

Transmettre la valeur d'échappement consiste à reparcourir une seconde fois la continuation jusqu'à trouver la continuation `target`. Lors du parcours, si des `unwind-protect` sont franchis, les formes de nettoyage associées seront évaluées. Observez la continuation des ces formes de ménage, de classe `unwind-cont` qu'il est donc possible de capturer. Elles correspondent à la poursuite de la scrutation de la continuation : les continuations flottantes que nous avions mentionnées.

En ce qui concerne `block`, la modification est de même nature mais ne concerne que `block-lookup` :

```
(define-method (block-lookup (r block-env) n k v) ; ** Modified **
  (if (eq? n (block-env-name r))
      (unwind k v (block-env-cont r))
      (block-lookup (block-env-others r) n k v) ) )
```

On recherche la continuation du `block` associé dans l'environnement lexical puis l'on redéroule (`unwind`) la continuation jusqu'à ce bloc cible.

On pourrait croire qu'en présence de `unwind-protect`, la forme `block` n'est pas plus rapide que `catch` puisque toutes deux partagent le coût (important) de `unwind`. En vérité et comme `unwind-protect` est une forme spéciale et que donc son utilisation ne peut être dissimulée, le gain est conservé pour tous les `return-from` non séparés

de leur `block` associé par un quelconque `unwind-protect` ou `lambda`.

Au rayon des curiosités, COMMON LISP (CLtL2 [Ste90]) a introduit une restriction sur les échappements dans les formes de nettoyage. Ceux-ci ne peuvent aller *moins loin* que l'échappement en cours. L'intention était d'empêcher tout programme de se coincer dans une situation d'où aucun échappement ne pouvait l'en tirer. **[☞ Ex. 3.9]** Ainsi le programme suivant est-il erroné car la forme de nettoyage veut en faire moins que l'échappement ciblé vers 1 :

COMMON LISP
```
(catch 1
  (catch 2
    (unwind-protect (throw 1 'foo)
      (throw 2 'bar) ) ) )          → error!
```

3.5 Comparaison entre `call/cc` et `catch`

Grâce aux objets, les continuations apparaissent comme des listes chaînées de blocs. Certains de ces blocs sont accessibles depuis l'environnement lexical, d'autres ne se peuvent retrouver qu'en parcourant la continuation de bloc en bloc. D'autres enfin induisent des traitements variés lorsqu'ils sont franchis.

Dans un langage tel que Lisp, doté de continuations à durée de vie dynamique, la notion de pile et de continuation sont synonymes. Lorsque l'on écrit `(evaluate ec r (make-if-cont k et ef r))`, on signifie que l'on empile un nouveau bloc stipulant quoi faire lors du retour de la valeur de la condition d'une alternative. Réciproquement, l'expression `(evaluate-begin (cdr (begin-cont-e* k)) (begin-cont-r k) (begin-cont-k k))` indique explicitement que le bloc courant est abandonné, dépilé au profit du bloc juste en dessous : `(begin-cont-k k)`. On pourra vérifier que dans un tel langage, aucune structure de données ne conserve des morceaux de continuation obsolètes, c'est-à-dire qui ont disparus de la pile. C'est pourquoi lorsque l'on sort d'un `block`, la continuation associée (et peut-être capturée ailleurs) est invalidée. En terme d'implantation, les continuations peuvent être conservées dans une pile (ou plusieurs piles synchronisées) et compilées en les primitives C : `setjmp`/`longjmp`. **[☞ p. 382]**

Il existe dans le dialecte EULISP[PE92], une forme spéciale nommée `let/cc` ayant la syntaxe suivante :

EULISP
```
(let/cc variable formes...)
```

Dylan [App92b] quant à lui écrit la même chose comme :

Dylan
```
(bind-exit (variable) formes...)
```

Cette forme spéciale lie la continuation courante à *variable* avec une portée égale au corps de la forme `let/cc`, resp. `bind-exit`. La continuation est donc de première classe avec une interface de fonction unaire. Toutefois sa durée de vie *utile* est dynamique : son emploi est donc limité au temps d'évaluation du corps de la forme liante `let/cc` ou `bind-exit`. De manière plus précise, l'objet valeur de *variable* a une durée de vie illimitée mais un effet intéressant limité à une durée de vie dynamique. Cette caractéristique est liée à EULISP ou Dylan et ne se retrouve ni en Scheme, où les continuations ont une durée de vie illimitée, ni en COMMON LISP, où les continuations ne sont pas des objets de première classe. On peut toutefois simuler ce comportement en écrivant :

```
(define-syntax let/cc
  (syntax-rules ()
    ((let/cc variable . body)
     (block variable
       (let ((variable (lambda (x) (return-from variable x))))
         . body ) ) ) ) )
```

Dans le monde de Scheme, les continuations ne peuvent plus être mises en pile car leur rétention dans des structures de données externes est possible. Il faut donc adopter un autre modèle, arborescent. Le plus naïf est de se passer de la pile et d'allouer les blocs de continuation directement dans le monceau comme diraient les Canadiens. Cette technique permet une allocation uniforme et une bonne portabilité [AS94]. En revanche, elle diminue la localité des références en mémoire et exige le maintien explicite des liens de chaînage entre blocs (voir cependant [MB93] qui propose une solution à ces problèmes). Les implanteurs travaillent en général dur pour garder le plus possible de choses en pile et, dans ce cas, l'implantation canonique de `call/cc` est de recopier la pile dans le tas : la continuation est donc cette copie de la pile. D'autres schémas ont bien sûr été étudiés [CHO88, HDB90] pour partager les copies, retarder les copies, copier partiellement, etc. Toutes introduisent cependant des cas coûteux.

Les formes `block` et `call/cc` sont plus proches que ne l'étaient `catch` et `call/cc`. Elles partagent une même discipline lexicale et ne se distinguent que par la durée de vie de la continuation. Une restriction de `call/cc` existe dans certains dialectes [IM89], nommée `call/ep` (pour *call with exit procedure*) fortement apparentée à `block`/`return-from` ainsi qu'au précédent `let/cc`. La fonction `call/ep` a la même interface que `call/cc` :

```
(call/ep (lambda (exit) ...))
```

La variable `exit` de la fonction unaire, argument de `call/ep`, est liée à la continuation de la forme `call/ep` limitée à sa durée de vie dynamique. La ressemblance avec `block` est donc flagrante sauf qu'au lieu d'user d'un espace de noms disjoints (l'environnement des échappements lexicaux), on emprunte l'espace des variables. La principale différence qu'apporte `call/ep` est que la continuation d'échappement est de première classe et peut donc être manipulée comme peut l'être tout objet de première classe. Au contraire, il faut explicitement construire cet objet avec `block` en écrivant `(lambda (x) (return-from` *`label`* `x))`. La puissance d'expression est la même mais tous les sites d'appels (les formes `return-from`) sont connus statiquement dans une forme `block` ce qui n'est pas toujours le cas lors d'un appel à `call/ep`. Par exemple, `(call/ep foo)` ne dit rien sur l'emploi de l'échappement sauf analyse plus fine, non locale de `foo`. La fonction `call/ep` complique donc le travail du compilateur par rapport à celui qu'exige une forme spéciale comme `block`.

Comparer `block` et `call/ep` révèle donc certaines différences. Une exécution efficace ne devrait pas créer la fermeture argument de `call/ep` si celui-ci est une forme `lambda` explicite. Il faut donc distinguer, à la compilation, le cas `(call/ep (lambda ...))` des autres puisque celui-ci peut être mieux compilé. Ce cas particulier se compare au traitement d'une forme spéciale puisque les deux induisent un traitement différencié dans le compilateur. La fonction est la forme véhiculaire la plus prisée des adeptes de Scheme tandis que la forme spéciale correspond plus à une sorte de déclaration facilitant la vie du compilateur. Toutes deux procurent un même pouvoir bien qu'une complexité différente à la fois pour l'utilisateur et l'implanteur.

En résumé, si la puissance sous le plus faible volume est recherchée alors `call/cc` est pour vous puisqu'elle permet en effet de coder toutes les structures de contrôle connues : échappement, coroutine, continuation partielle ... Si vous n'avez que des besoins normaux (et Lisp a montré que l'on pouvait déjà écrire de nombreuses applications intéressantes sans `call/cc`), alors choisissez les formes de COMMON LISP dont la compilation est simple et dont le code engendré est efficace.

3.6 Programmation par continuations

Un style de programmation existe, bâti sur les continuations. Il consiste à fournir explicitement à un calcul le receveur du résultat. Lorsque le calcul s'achève, l'exécutant applique le receveur au résultat plutôt que de le rendre comme valeur normale. Cela revient, si l'on a le calcul `(foo (bar))`, à modifier la fonction `bar` en `new-bar` pour prendre un argument, qui sera justement ce que l'on appellera la continuation et, dans le cas présent, la fonction `foo`. Le calcul modifié apparaîtra donc comme `(new-bar foo)`. Donnons un exemple avec notre vieille connaissance la factorielle en supposant que l'on veuille calculer $n(n!)$:

```
(define (fact n k)
  (if (= n 0) (k 1)
      (fact (- n 1) (lambda (r) (k (* n r)))) ) )
(fact n (lambda (r) (* n r)))  → n(n!)
```

La factorielle prend donc un nouvel argument `k`, le receveur du résultat final. Lorsque le résultat est `1`, il est simple d'appliquer `k` dessus. En revanche, lorsque le résultat n'est pas immédiat, on se rappelle récursivement comme de bien entendu. Le problème est maintenant à la fois de transmettre le receveur et de multiplier par `n` la factorielle de `n`-1. Et bien que la multiplication soit commutative, on doit faire ces choses dans l'ordre ! D'abord multiplier par `n` puis appeler le receveur. Comme en final celui-ci est appliqué à `1`, il ne reste plus qu'à composer le receveur et le traitement supplémentaire et ainsi obtenir `(lambda (r) (k (* n r)))`.

L'avantage qu'offre cette nouvelle factorielle est que la même définition permet de calculer de nombreux résultats et notamment la factorielle `(fact` n `(lambda (x) x))`, le double de la factorielle `(fact` n `(lambda (x) (* 2 x)))`, etc.

3.6.1 Valeurs multiples

User de continuations est aussi la clé des valeurs multiples. Lorsqu'un calcul a plusieurs résultats à retourner, les continuations sont une technique intéressante. En COMMON LISP, la division retourne une valeur multiple composée du quotient et du reste. Un opérateur semblable, que l'on nommera `divide`, peut être procuré qui prend deux nombres et une continuation, calcule le quotient et le reste euclidien de ces deux nombres et leur applique la continuation de troisième argument. Ainsi, si l'on veut vérifier qu'une division est juste, on pourra écrire :

```
(let ((p (read)) (q (read)))
  (divide p q (lambda (quotient remainder)
                (= p (+ (* quotient q) remainder)) )) )
```

Un exemple plus probant est le calcul des nombres de Bezout. L'identité de Bezout stipule que si p et q sont premiers entre eux, alors il existe des nombres u et v tels que $up + vq = 1$. Le calcul des nombres de Bezout passe par le calcul du pgcd qui vérifie au passage que p et q sont bien premiers entre eux.

```
(define (bezout n p k)      ; assume n > p
  (divide
   n p (lambda (q r)
         (if (= r 0)
             (if (= p 1)
             (k 0 1)   ; since 0 × 1 − 1 × 0 = 1
             (error "not relatively prime" n p) )
(bezout
 p r (lambda (u v)
       (k v (- u (* v q))) ) ) ) ) ) ) )
```

La fonction `bezout` utilise `divide` qui remplit `q` et `r` comme les quotient et reste de la division de `n` par `p`. Si les nombres sont finalement premiers, une solution triviale est fournie par 0 et 1, autrement on « remonte » ces valeurs jusqu'aux `n` et `p` initiaux. On pourra vérifier les mathématiques sous-jacentes, il suffit de connaître la théorie du pgcd ou l'on vérifiera (c'est la « preuve » par le feu) que

```
(bezout 1991 1960 list) → (-569 578)
```

3.6.2 Récursion terminale

Dans l'exemple de la factorielle avec continuations, un appel à `fact` se résolvait généralement à ne rien être d'autre qu'un appel à `fact`. Si l'on trace le calcul de `(fact 3 list)` et en sautant les étapes évidentes, on obtient :

```
(fact 3 list)
≡ (fact 2 (lambda (r) (k (* n r))))
                                    | n→ 3
                                    | k≡ list
≡ (fact 1 (lambda (r) (k (* n r))))
                                    | n→ 2
                                    | k→ (lambda (r)
                                           (k (* n r)))
                                                        | n→ 3
                                                        | k≡ list
≡ (k (* n 1))
             | n→ 2
             | k→ (lambda (r) (k (* n r)))
                                          | n→ 3
                                          | k≡ list
≡ (k (* n 2))
             | n→ 3
             | k≡ list
→ (6)
```

Lorsque l'appel à `fact` se traduit directement en un appel à `fact`, le nouvel appel s'effectue avec la même continuation que l'ancien appel. Cette propriété s'appelle la *récursion terminale*. Récursion car récursion il y a, et terminale car c'est la dernière chose qu'il reste à faire. La récursion terminale est un cas particulier de l'*appel terminal*, lorsqu'un calcul se résoud en un autre sans nécessiter de revenir au calcul abandonné. Un appel en *position terminale* s'effectue à *continuation constante*.

Dans l'exemple de la fonction `bezout`, `bezout` appelle en position terminale la fonction `divide` qui elle-même appelle sa continuation en position terminale qui appelle récursivement `bezout` en position terminale encore. Dans la factorielle classique, l'appel récursif `(* n (fact (- n 1)))` est dit *enveloppé* puisque la valeur de `(fact (- n 1))` est reprise dans l'environnement courant pour être multiplié par `n`.

Un appel terminal permet d'abandonner complètement l'environnement courant qui n'est plus nécessaire, qui donc ne mérite pas d'être sauvegardé et qui, par conséquent, ne sera pas restauré, procurant ainsi un gain appréciable. Ces techniques ont été très finement étudiées par l'école lispienne française [Gre77, Cha80, SJ87] et ont permis la réalisation des interprètes les plus véloces du monde.

La récursion terminale est une qualité très recherchée et l'interprète lui-même peut en user avec quelque bonheur. Le fondement de l'optimisation liée à la récursion terminale est très simple et se voit bien sur la définition de la séquence. On avait écrit :

```
(define (evaluate-begin e* r k)
  (if (pair? e*)
    (if (pair? (cdr e*))
      (evaluate (car e*) r (make-begin-cont k e* r))
      (evaluate (car e*) r k) )
    (resume k empty-begin-value) ) )
(define-method (resume (k begin-cont) v)
  (evaluate-begin (cdr (begin-cont-e* k))
                  (begin-cont-r k)
                  (begin-cont-k k) ) )
```

On aurait pu aussi bien écrire plus simplement :

```
(define (evaluate-begin e* r k)
  (if (pair? e*)
      (evaluate (car e*) r (make-begin-cont k e* r))
      (resume k empty-begin-value) ) )
(define-method (resume (k begin-cont) v)
  (let ((e* (cdr (begin-cont-e* k))))
   (if (pair? e*)
       (evaluate-begin e* (begin-cont-r k) (begin-cont-k k))
       (resume (begin-cont-k k) v) ) ) )
```

La première écriture est préférable car elle permet, lorsque l'on évalue le dernier terme d'une séquence, de ne pas construire la continuation `(make-begin-cont k e* r)`, dont l'allocation et le traitement coûtent en temps et espace pour finalement s'avérer être équivalente à `k`. Conformément au principe de bonne économie qui veut que l'on ne garde pas indûment les objets inutiles (et notamment l'environnement `r` présent dans `(make-begin-cont k e* r)`), il vaut mieux se débarrasser au plus tôt de cette continuation superflue. Ce cas est très important car toutes les séquences ont un dernier terme !

Le même effet peut être appliqué à l'évaluation des arguments. On écrira donc et pour les mêmes raisons qu'au dessus :

```
(define-class no-more-argument-cont continuation ())
(define (evaluate-arguments e* r k)
  (if (pair? e*)
      (if (pair? (cdr e*))
          (evaluate (car e*) r (make-argument-cont k e* r))
```

```
            (evaluate (car e*) r (make-no-more-argument-cont k)) )
        (resume k no-more-arguments) ) )
(define-method (resume (k no-more-argument-cont) v)
  (resume (no-more-argument-cont-k k) (list v)) )
```

On a ici créé un nouveau type de continuation qui permet de mettre en liste la valeur du dernier terme d'une liste d'arguments sans pour cela nécessiter la rétention de l'environnement `r`. La découverte de cet effet est attribuée à Friedman et Wise [Wan80b].

3.7 Continuations partielles

Parmi les questions que peuvent soulever les continuations, l'une porte sur la constatation suivante : lors d'un échappement, que devient la partie qui disparaît ? En d'autres termes, la portion de la continuation (ou « pile d'exécution ») qui se situe entre l'endroit d'où l'on s'échappe et l'endroit où l'on saute représente une tranche de continuation. Cette tranche prend une valeur en entrée et comme elle a une fin, elle fournit un résultat en sortie, elle est donc équivalente à une fonction unaire. Il a été proposé des formes de contrôle élaborées permettant de réifier ces portions de continuations ou *continuations partielles* [FFDM87, FF87, Fel88, DF90, HD90, QS91, MQ94].

Considérez un instant les simplissimes interactions suivantes :

```
(+ 1 (call/cc (lambda (k) (set! foo k) 2)))   → 3
(foo 3)                                       → 4
```

Conformément à ce que l'on a préalablement annoncé, la continuation `k`, affectée à `foo`, est $\lambda u.1 + u$. Mais alors, que vaut `(foo (foo 4))` ?

```
(foo (foo 4))                                 → 5
```

Le résultat est 5 et non 6 comme la composition des continuations aurait pu le laisser croire. En effet, appeler une continuation correspond à un abandon du calcul en cours et donc un seul appel au plus peut être réalisé : l'appel intérieur à `foo` force le calcul de la continuation $\lambda u.1 + u$ sur 4 avec pour mission de retourner la valeur obtenue comme valeur finale du calcul tout entier. On ne revient donc jamais d'une continuation ! Plus précisément, la continuation `k` est d'attendre une valeur, de lui ajouter 1 et de retourner le résultat comme résultat *final* et *définitif*.

Le même exemple est peut-être plus parlant avec des contextes. La continuation de l'exemple était `(+ 1 [])`. Puisque nous sommes en appel par valeur, `(foo (foo 4))` correspond à l'élimination du contexte `(foo [])` autour de `(foo 4)` pour se réécrire en `(+ 1 4)`, qui vaut finalement `5`.

Les continuations partielles représentent la suite des calculs qui restent à effectuer jusqu'à un certain point dûment identifié. Plusieurs propositions existent qui permettent de rendre les continuations partielles et donc composables [FWFD88, DF90, QS91]. Supposons que la continuation liée à `foo` soit maintenant `[(+ 1 [])]`, où les crochets externes indiquent qu'il faut retourner une valeur. Alors `(foo (foo 4))` est `(foo [(+ 1 [4])])` puis `(+ 1 5)` et vaut enfin 6. La continuation capturée `[(+ 1 [])]` ne définit pas toute la suite du calcul mais seulement ce qui restait jusqu'au retour à la boucle fondamentale d'interaction non comprise. Cette continuation a donc une fin et, par là même, est donc fonctionnelle et composable.

Une autre façon de voir cet effet est de reprendre l'exemple plus haut qui faisait un effet de bord sur la variable globale `foo` ainsi qu'une interaction avec la boucle fondamentale. Condensons-donc les deux expressions en une seule et écrivons :

```
(begin (+ 1 (call/cc (lambda (k) (set! foo k) 2)))
       (foo 3) )
```

Patatras ! Cela boucle car `foo` est maintenant liée au contexte `(begin (+ 1 []) (foo 3))` qui s'appelle récursivement. Nous pouvons en conclure que l'agrégation de formes que nous venons d'effectuer n'est pas neutre vis-à-vis des continuations, pas plus que la boucle fondamentale d'interaction ne l'est. Si l'on souhaite simuler mieux l'effet de la boucle fondamentale d'interactions, on pourrait écrire :

```
(let (foo sequel print?)
  (define-syntax toplevel
    (syntax-rules ()
      ((toplevel e) (toplevel-eval (lambda () e))) ) )
  (define (toplevel-eval thunk)
    (call/cc (lambda (k)
               (set! print? #t)
               (set! sequel k)
               (let ((v (thunk)))
                 (when print? (display v)(set! print? #f))
                 (sequel v) ) )) )
  (toplevel (+ 1 (call/cc (lambda (k) (set! foo k) 2))))
  (toplevel (foo 3))
  (toplevel (foo (foo 4))) )
```

Chaque fois que `toplevel` se saisit d'une forme pour l'évaluer, on mémorise dans la variable `sequel` la continuation menant à la prochaine forme à évaluer. Toute continuation prise pendant l'évaluation est donc de ce fait limitée à la forme courante. Comme on peut le constater, lorsqu'un `call/cc` est utilisé hors de sa durée de vie dynamique, il faut soit un effet de bord, soit analyser la valeur reçue afin d'éviter de boucler.

Les continuations partielles spécifient jusqu'où prendre une continuation, ce qui permet de ne pas en prendre trop et d'obtenir des effets intéressants. On peut alors réécrire l'exemple précédent pour redéfinir `call/cc` pour ne capturer de continuations que jusqu'à `toplevel` et pas au-delà. La notion d'échappement est aussi nécessaire pour éliminer une tranche de continuation. Les continuations partielles ne sont cependant pas vraiment convaincantes, à mon sens, car on ne connaît pas de programme les utilisant de façon profitable et qui soit plus simple que ce que l'on pourrait directement écrire. Ce qui par contre est essentiel, toujours à mon sens, est que tous les opérateurs proposés pour les continuations partielles peuvent se simuler en Scheme avec `call/cc` et affectations.

3.8 Conclusion

Les continuations sont omniprésentes ; bien les appréhender permet à la fois de se doter d'un nouveau style de programmation, de maîtriser les arcanes des formes de contrôle et d'estimer leur coût d'utilisation. Les continuations sont liées au contrôle de l'exécution et à tout instant représentent dynamiquement le travail restant à effectuer.

Elles sont de ce fait très utiles pour le traitement des exceptions mais également pour l'écriture d'applications Web [Que04, MFG+04].

L'interprète présenté dans ce chapitre est très précis et encore assez lisible localement. La profusion de petits morceaux de code habituelle à la programmation par objets rend par contre plus délicate la compréhension du tout. L'interprète est très modulaire et supporte bien l'expérimentation de nouveaux traits linguistiques. Il est peu véloce car il consomme de façon effrénée des objets pour les abandonner, le plus souvent, immédiatement. C'est un des rôles de la compilation que de déterminer les seules entités utiles à construire.

3.9 Exercices

Exercice 3.1 : Que vaut `(call/cc call/cc)` ? L'ordre d'évaluation influence-t-il votre réponse ?

Exercice 3.2 : Que vaut `((call/cc call/cc) (call/cc call/cc))` ?

Exercice 3.3 : Écrivez une simulation du couple `tagbody`/`go` avec `block`, `catch`, `labels`. On rappelle la syntaxe de `tagbody` telle que la définit COMMON LISP :

```
(tagbody
        expressions0...
  label1  expressions1...
                  ...
  labeli  expressionsi...
                  ... )
```

Toutes les $expressions_i$ et seules les $expressions_i$ peuvent comporter des formes de débranchement inconditionnel `(go` *label*`)` ou les échappements `(return` *value*`)`. Si aucun `return` n'est rencontré, la valeur finale de `tagbody` est `nil`.

Exercice 3.4 : On aura pu remarquer, lors de l'invocation des fonctions, que celles-ci vérifient leur arité effective, c'est-à-dire le nombre d'arguments qui leur est soumis. Modifiez la création des fonctions afin de précalculer leur arité pour accélérer cette vérification. On ne traitera que le cas des fonctions avec une arité fixe.

Exercice 3.5 : Définissez la fonction `apply` appropriée à l'interprète de ce chapitre.

Exercice 3.6 : Étendez les fonctions reconnues par l'interprète de ce chapitre afin qu'il accepte les fonctions d'arité variable.

Exercice 3.7 : Modifiez le lancement de l'interprète de manière à n'appeler la fonction `evaluate` qu'une unique fois.

Exercice 3.8 : Le codage des continuations tel que présenté au paragraphe 3.4.1 mélangeait continuations et valeurs. Comme des instances de la classe `continuation` pouvaient apparaître en valeur, on avait été obligé de définir une méthode pour `invoke` sur `continuation`. Redéfinissez `call/cc` afin de créer une nouvelle sous-classe de `value` correspondant aux continuations réifiées.

Exercice 3.9 : Écrivez en COMMON LISP une fonction `eternel-retour` prenant une suspension en argument, l'appelant sans cesse et faisant en sorte qu'aucun échappement tenté par la suspension ne réussisse à sortir de la fonction `eternel-retour`.

Exercice 3.10 : Soit la fonction (due à Alan Bawden) suivante :

```
(define (make-box value)
  (let ((box
         (call/cc
          (lambda (exit)
            (letrec
                ((behavior
                  (call/cc
                   (lambda (store)
                     (exit (lambda (msg . new)
                             (call/cc
                              (lambda (caller)
                                (case msg
                                  ((get) (store (cons (car behavior)
                                                      caller )))
                                  ((set)
                                   (store
                                    (cons (car new)
                                          caller ) ) ) ) ) ) ) ) ) ) ))
                 ((cdr behavior) (car behavior)) ) ) ) ) )
    (box 'set value)
    box ) )
```

Que valent les expressions ci-dessous si l'on suppose que `box1` a pour valeur le résultat de `(make-box 33)` ?

```
(box1 'get)
(begin (box1 'set 44) (box1 'get))
```

Exercice 3.11 : La fonction `evaluate` est bien la seule à ne pas être générique. Si l'on créait la classe des programmes avec autant de sous-classes qu'il y a de formes syntaxiques différentes, le lecteur pourrait alors retourner non plus une S-expression mais un objet correspondant au programme. La fonction `evaluate` serait alors générique et l'on pourrait facilement et incrémentiellement ajouter de nouvelles formes spéciales. Reconcevez donc l'interprète pour que ce soit possible.

Exercice 3.12 : Définissez `throw` comme une fonction plutôt que comme une forme spéciale.

Exercice 3.13 : Comparez les vitesses d'exécution de code normal et de code traduit en CPS.

Exercice 3.14 : Programmez `call/cc` au moyen de `the-current-continuation`. On supposera que cette dernière est définie par :

```
(define (the-current-continuation)
   (call/cc (lambda (k) k)) )
```

3.10 Lectures recommandées

Un bon exemple d'usage non trivial de continuations figure en [Wan80a]. On pourra aussi lire [HFW84] sur la simulation de structures échevelées de contrôle. La notion de réflexion en matière de formes de contrôle est remarquablement située dans sa perspective historique en [dR87].

4

Affectation et effets de bord

Dans le crescendo ravélien des effets, facettes et autres variations que vous subissez depuis quelques chapitres, deux lacunes sont apparentes, ce sont l'affectation et les effets de bord. Certains langages les abhorrent en raison de leurs néfastes caractéristiques. Les dialectes de Lisp les procurant, leur étude s'impose donc ici. Ce chapitre examine en détail le mécanisme d'affectation ainsi que les autres effets de bord perpétrables. Ces discussions amèneront notamment à quelques échappées sur l'égalité et la sémantique des citations.

Issue des langages algorithmiques classiques, l'affectation permet grosso modo de modifier la valeur associée à une variable. Elle induit une modification de l'état du programme qui doit enregistrer d'une manière ou d'une autre que telle ou telle variable vaut désormais une autre valeur que précédemment. Pour qui a peu ou prou tâté des langages impératifs, le sens imputable à l'affectation paraît simple et pourtant, ce chapitre montrera que la présence de fermetures ainsi que l'héritage du λ-calcul compliquent les notions de liaison et de variable.

Le problème majeur de la définition de l'affectation (et des effets de bord) est de choisir une formalisation indépendante des traits mêmes que l'on souhaite décrire. En conséquence, ni affectation ni effet de bord ne doivent apparaître dans la description. Non que nous en eussions usé immodérément auparavant ; les seuls effets de bord qui apparaissaient dans les interprètes précédemment donnés étaient localisés dans la fonction `update!` (lorsque, justement, il s'agissait de décrire l'affectation) et dans la définition des outils chirurgicaux, tel `set-car!` (du Lisp défini) qui n'était qu'une enveloppe autour de `set-car!` (du Lisp de définition).

4.1 Affectation

L'affectation permet de modifier la valeur d'une variable ; on peut, par exemple, programmer la recherche du minimum et du maximum d'un arbre binaire d'entiers naturels en maintenant deux variables contenant la plus grande et la plus petite des valeurs découvertes dans la partie de l'arbre déjà explorée. On écrira :

```
(define (min-max tree)
  (define (first-number tree)
    (if (pair? tree)
```

```
        (first-number (car tree))
        tree ) )
  (let* ((min (first-number tree))
         (max min) )
    (define (scan! tree)
      (cond ((pair? tree)
             (scan! (car tree))
             (scan! (cdr tree)) )
            (else (if (> tree max) (set! max tree)
                      (if (< tree min) (set! min tree)) )) ) )
    (scan! tree)
    (list min max) ) )
```

La fonction `min-max` est facile à comprendre et a l'avantage de ne consommer que deux paires pointées pour retourner le résultat souhaité. Cet algorithme est tout à fait semblable à ce que l'on pourrait écrire en Pascal et est non moins représentatif de l'usage classique que l'on peut faire des variables. On notera aussi que les effets de bord perpétrés sur ces variables locales sont totalement invisibles de l'extérieur de la fonction `min-max` et donc ne peuvent nuire aux qualités générales du programme enrobant : c'est un effet de bord sain, clair et efficace comparé à des versions purement fonctionnelles de ce même programme **[☞ Ex. 4.1]**

L'affectation sur une variable non partagée ne pose guère de problèmes. Dans la fonction qui suit et qui énumère les entiers naturels à partir de zéro, il est évident que demander la valeur de la variable `n`, immédiatement après l'avoir affectée, retourne la nouvelle valeur qu'elle a prise.

```
(define enumerate
  (let ((n -1))
    (lambda () (set! n (+ n 1))
               n ) ) )
```

Tout appel à `enumerate` retourne un nombre. La fonction `enumerate` a un état interne représenté par le nombre `n` qui est modifié au fur et à mesure qu'`enumerate` est appelée. Altérer une variable close est un problème en soi. Le λ-calcul est en effet muet sur la sémantique de cet effet qui n'existe que parce que l'on désire s'offrir l'affectation au sein du langage. L'application d'une fonction en mathématiques, comme en λ-calcul, s'effectue par substitution de ses variables par les valeurs ou expressions qu'elles prennent lors de l'application. L'affectation force l'abandon de cette sémantique et fait perdre la transparence référentielle pour les programmes qui l'utilisent. Des théories plus puissantes raffinant la notion de transparence référentielle sont donc nécessaires. Si, par exemple, on remplaçait toutes les occurrences de la variable `n` par sa valeur initiale dans la fonction `enumerate`, alors il ne serait plus question d'engendrer la suite des naturels et `enumerate` retournerait sempiternellement la valeur `-1`. L'affectation force donc l'abandon de la substitution instantanée des variables par leur valeur comme modèle de calcul. Cette substitution est maintenant différée dans le temps et n'est effectuée que quand on désire la valeur précise d'une variable. Il y a alors moyen d'intercaler des affectations pour modifier les substitutions non encore effectuées.

Un nouvel exemple fera mieux sentir le problème. Soit le programme :

```
(let ((name "Nemo"))
  (set! winner (lambda () name))
```

```
(set! set-winner! (lambda (new-name) (set! name new-name)
                                     name ))
(set-winner! "Me")
(winner) )
```

) L'appel `(winner)` va-t-il maintenant retourner `"Nemo"` ou `"Me"` ? En d'autres termes, les modifications qu'apporte la fonction `set-winner!` sont-elles perçues de `winner` ? Le λ-calcul est, une fois de plus, muet sur ce point et pour cause puisque la notion d'affectation lui est étrangère. Nous avions toutefois dit que la création d'une fonction capturait son environnement de définition, c'est-à-dire que les fonctions `winner` et `set-winner!` mémorisent que la variable `name` a, à l'instant de leur création, la valeur `"Nemo"`. Il semble évident que la forme `(set-winner! "Me")` va retourner `"Me"` puisque le texte de la fonction déclare que l'on affecte la variable `name` puis que l'on retourne sa valeur. Le problème est de déterminer si `(winner)` qui voit la même variable `name` va maintenant voir cette nouvelle valeur. Une vision littérale du fait qu'une fermeture est définie dans l'environnement où les variables libres de son corps ont pour valeur les valeurs qu'elles avaient lors de sa création militerait pour l'isolement et donc on ne pourrait distinguer le programme précédent du suivant :

```
(let ((name "Nemo"))
  (set! winner (lambda () name))
  (winner) )
```

Il a été proposé [Sam79] que l'affectation ne soit visible que de l'affectant. Dans ce cas, la fonction `set-winner!` retournerait la valeur modifiée tandis que `winner` continuerait de ne toujours renvoyer que `"Nemo"`. Dans ce monde, la notion de liaison n'existe pas ; il y a bien une connexion entre variable et valeur mais celle-ci est directe. La référence à une variable consulte donc l'environnement courant et retourne la valeur associée. Une affectation crée un nouvel environnement équivalent à l'ancien sauf que la variable affectée vaut maintenant la valeur affectée et fournit cet environnement comme environnement courant à la continuation. Cette vision rappelle la classique implantation des fermetures dans les anciens Lisps dynamiques. Une fermeture était créée explicitement par la forme spéciale `closure` qui prenait en premier terme la liste des variables à clore et en second terme une fonction. Par exemple, la forme `(closure (x) (lambda (y) (+ x y)))` conduisait à `(lambda (y) (let ((x 'valeur-de-x)) (+ x y)))`. Il y a bien capture de la valeur associée à la variable `x`, ce qui semble tout à fait conforme à la philosophie héritée du λ-calcul. Ce modèle supporte cependant mal l'affectation et rend délicate la notion de variable partagée. Des solutions ont aussi été proposées comme l'existence d'une forme `closure` [SJ87] analysant le corps de la fonction à clore pour en extraire automatiquement les variables libres qui sont à capturer. On peut aussi modifier les affectations qui se trouvent dans le corps de la fonction à clore de manière à régler le difficile problème de la mise à jour des variables closes [BCSJ86, SJ93].

Scheme tranche différemment le problème en introduisant la notion de *liaison* et il se trouve que cette notion permet un certain nombre d'effets programmatiques intéressants. La forme `let` introduit une nouvelle liaison entre la variable `name` et la valeur `"Nemo"`. Les fonctions qui sont créées dans le corps de `let` capturent non pas la valeur de la variable `name` mais sa liaison. La référence à la variable `name` est interprétée alors comme la recherche de la liaison associée à la variable `name` puis l'extraction de la valeur associée à la liaison précédemment découverte. L'affectation

procède de la même technique : l'affectation recherche la liaison associée à la variable `name` puis altère la valeur associée au sein de cette liaison. La liaison est donc une entité de seconde classe : une variable référence une liaison ; une liaison désigne une valeur. Affecter une variable ne change pas la liaison à laquelle elle est associée mais modifie le contenu de cette liaison pour désigner une nouvelle valeur.

4.1.1 Les boîtes

Pour donner un support à la notion de liaison, on peut reprendre l'image de la A-liste qui nous a servi à représenter, au sein des précédents interprètes, l'environnement des variables. L'épine dorsale de la A-liste ne sert qu'à regrouper l'ensemble des couples variable-valeur. Un couple variable-valeur est représenté par une paire pointée. C'est cette paire pointée que l'on recherche lorsque la variable est référencée soit en lecture soit en écriture. Cette paire pointée (ou plus exactement son champ `cdr`) est modifiée par affectation. On peut donc identifier, pour cette représentation, la liaison à cette paire pointée. D'autres codages sont possibles pour les environnements et, notamment, les liaisons peuvent être représentées par des *boîtes*. Cette transformation est importante car elle permet de s'affranchir totalement des affectations au profit des seuls effets de bord.

Une boîte est créée par la fonction `make-box` qui lui confère sa valeur initiale ; elle est lue et écrite par les fonctions `box-ref` et `box-set!`. Une première réalisation en style dit « par messages » (pour *message passing style*) est :

```
(define (make-box value)
  (lambda (msg)
    (case msg
      ((get) value)
      ((set!) (lambda (new-value) (set! value new-value))) ) ) )
(define (box-ref box)
  (box 'get) )
(define (box-set! box new-value)
  ((box 'set!) new-value) )
```

Une réalisation sans fermeture userait directement de paires pointées :

```
(define (other-make-box value)
  (cons 'box value) )
(define (other-box-ref box)
  (cdr box) )
(define (other-box-set! box new-value)
  (set-cdr! box new-value) )
```

Enfin on pourrait tout simplement employer `define-class` et seulement écrire :

```
(define-class box Object (content))
```

Dans ces deux réalisations, on notera l'indétermination de la valeur que retourne `box-set!`. Toute variable qui subit une (ou plusieurs) affectation(s) peut être transcrite en une boîte qui sera convenablement mise en œuvre. Il est simple de déterminer si une variable est écrite en regardant dans le corps de la forme qui la lie si la variable fait l'objet d'une affectation. C'est un des attraits des langages lexicaux que tous les sites d'emploi d'une variable locale soient visibles. La transformation sera spécifiée par des règles de réécritures où π sera transformée en $\overline{\pi}^{v}$ et où l'on nomme v la variable à transformer en boîte :

$$
\begin{aligned}
\overline{x}^{v} &= \text{if } x = v \text{ then } \texttt{(box-ref } v\texttt{)} \text{ else } x \\
\overline{\texttt{(quote } \epsilon\texttt{)}}^{v} &= \texttt{(quote } \epsilon\texttt{)} \\
\overline{\texttt{(if } \pi_c\ \pi_t\ \pi_f\texttt{)}}^{v} &= \texttt{(if } \overline{\pi_c}^{v}\ \overline{\pi_t}^{v}\ \overline{\pi_f}^{v}\texttt{)} \\
\overline{\texttt{(begin } \pi_1\ \ldots \pi_n\texttt{)}}^{v} &= \texttt{(begin } \overline{\pi_1}^{v}\ \ldots \overline{\pi_n}^{v}\texttt{)} \\
\overline{\texttt{(set! } x\ \pi\texttt{)}}^{v} &= \text{if } x = v \text{ then } \texttt{(box-set! } v\ \overline{\pi}^{v}\texttt{)} \text{ else } \texttt{(set! } x\ \overline{\pi}^{v}\texttt{)} \\
\overline{\texttt{(lambda (}\ldots x \ldots\texttt{) } \pi\texttt{)}}^{v} &= \text{if } v \in \{\ldots x \ldots\} \text{ then } \texttt{(lambda (}\ldots x \ldots\texttt{) } \pi\texttt{)} \\
&\quad \text{else } \texttt{(lambda (}\ldots x \ldots\texttt{) } \overline{\pi}^{v}\texttt{)} \\
\overline{\texttt{(}\pi_0\ \pi_1\ \ldots \pi_n\texttt{)}}^{v} &= \texttt{(} \overline{\pi_0}^{v}\ \overline{\pi_1}^{v}\ \ldots \overline{\pi_n}^{v}\texttt{)}
\end{aligned}
$$

Comme toujours on doit prendre garde à correctement traiter les variables locales qui porteraient le nom même de la variable que l'on souhaite réécrire en boîte. En réitérant le processus sur toutes les variables qui font l'objet d'une affectation, on supprime totalement toute affectation au profit de boîtes, dont le contenu peut être révisé par effet de bord. On ajoutera donc la règle suivante :

$$
\begin{aligned}
&\texttt{(lambda (}\ldots x \ldots\texttt{) } \pi\texttt{)} \quad \wedge \quad \texttt{(set! } x\ \ldots\texttt{)} \in \pi \\
&\rightarrow \quad \texttt{(lambda (}\ldots x \ldots\texttt{) (let ((}x\ \texttt{(make-box } x\texttt{)))}\ \overline{\pi}^{x}\texttt{)}
\end{aligned}
$$

On peut donc reprendre l'exemple précédent et le retranscrire en terme de boîte ce qui conduit à :

```
(let ((name (make-box "Nemo")))
  (set! winner (lambda () (box-ref name)))
  (set! set-winner! (lambda (new-name) (box-set! name new-name)
                                       (box-ref name) ))
  (set-winner! "Me")
  (winner) )
```

L'emploi de boîtes en lieu et place de variables modifiables, c'est-à-dire affectables est classique et correspond aux « références » présentes dans les dialectes de ML. Les boîtes offrent l'avantage de supprimer (apparemment) les affectations et les problèmes qui leur sont liés, elles permettent aussi de ne pas introduire de cas particuliers lors de la recherche de la valeur d'une variable puisqu'aucune variable n'est maintenant modifiable. Étant toutefois de purs effets de bord, elles font perdre la transparence référentielle mais se prêtent bien au typage. Néanmoins, l'introduction de boîtes amène deux problèmes : le premier est que la liaison dont la représentation est maintenant la boîte est désormais un objet de première classe qui peut être manipulé en tant que tel. En d'autres termes `box-ref` et `box-set!` ne sont pas les seules opérations qui se peuvent appliquer aux boîtes. Que deux personnes connaissent la même boîte procure un effet d'alias qui peut notamment être mis en œuvre pour implanter des modules.

Le second problème est que le devenir d'une boîte est généralement indéterminé puisque, comme une paire pointée que n'importe qui peut soumettre à un `set-car!` désastreux, les sites d'emploi des boîtes sont généralement inconnus. En revanche, l'affectation lexicale a ceci pour elle que tous les sites où une liaison peut être altérée sont statiquement connus. Cette connaissance peut être mise à profit lors de la compilation lorsque par exemple une variable affectée n'est ni close ni partagée comme c'est le cas, par exemple, pour les variables `min` et `max` dans la fonction `min-max`.

En définitive, à chaque variable affectable est associé un emplacement mémoire (caractérisé par son adresse ou *location*) contenant la valeur de la liaison. C'est cet emplacement en mémoire qui sera modifié lorsque la variable sera affectée.

4.1.2 Affectation sur variable libre

Un autre problème lié à l'affectation est le sens qu'il faut accorder à une affectation sur une variable libre comme par exemple :

```
(let ((passwd "timhukiTrolrk"))
  (set! can-access? (lambda (pw) (string=? passwd (crypt pw)))) )
```

La variable `can-access?` est libre dans le corps du `let` ; elle est de plus affectée. Lorsque l'on suit les règles de Scheme, la variable `can-access?` ne peut être que globale puisque non revendiquée localement. Que l'environnement soit global ne signifie pas nécessairement qu'il contient la variable `can-access?`. Comme nous en avions déjà débattu, **[☞ p. 51]** plusieurs solutions sont envisageables.

Que peut-on faire d'une variable globale outre l'affecter ? Comme toute variable on peut la référencer, l'enclore, obtenir sa valeur et en général la définir avant toute autre opération. L'environnement global est lui-même un espace de noms dont nous allons passer en revue les différentes réalisations.

Environnement global universel

L'environnement global peut être défini comme le lieu où toutes les variables préexistent. De fait, l'implantation se débrouille, chaque fois qu'un nouveau nom de variable apparaît de faire en sorte que la variable ainsi nommée soit présente dans l'environnement global, comme si elle y avait toujours été. Dans ce monde et pour tout nom, il existe une, exactement une et une seule variable portant ce nom. La notion de définition d'une variable globale n'a pas de sens puisque toutes déjà préexistent. Modifier une variable ne pose pas de problème puisque son existence ne peut être mise en doute. On peut donc réduire l'opérateur `define` à n'être que `set!` et corollairement plusieurs définitions de la même variable sont possibles.

Un seul problème se pose lorsque l'on désire obtenir la valeur d'une variable qui n'a pas encore été affectée. La variable existe mais n'a pas encore de valeur. Ce type d'erreur est souvent affublée du message *unbound variable* alors que *stricto sensu* la variable possède une liaison, c'est-à-dire une boîte associée.

On peut résumer les propriétés de cet environnement global par le tableau suivant :

Référence	*x*
Valeur	*x*
Modification	`(set! x ...)`
Extension	non
Définition	non, `define`≡`set!`

Cette définition d'environnement est intéressante à plus d'un titre. Elle ne nécessite que peu de concepts puisque tout préexiste ; elle autorise d'écrire simplement des fonctions mutuellement récursives et, en cas d'erreur, permet de redéfinir des fonctions globales. Cette dernière caractéristique fait que toute référence à une variable globale doit être traitée avec précaution puisque *(i)* elle peut être non initialisée (mais une fois qu'elle l'est, elle le demeure) ; *(ii)* elle peut changer de valeur, ce qui interdit toute hypothèse se fondant sur sa valeur courante. En particulier, on ne devrait même pas intégrer (*inline*) les fonctions primitives `car` ou `cons`, mais on peut recompiler automatiquement tout ce qui dépend d'une hypothèse qui viendrait à être bafouée.

Pour illustrer cet environnement, considérez le fragment suivant qui illustre ces diverses propriétés :

```
g                              ; error:  g uninitialized
(define (P m) (* m g))
(define g 10)
(define g 9.81)                ; ≡  (set! g 9.81)
(P 10)                 → 98.1
(set! e 2.78)                  ; definition of  e
```

Pour conclure, l'environnement global peut être vu comme une gigantesque forme `let` définissant toutes les variables, quelque chose comme :

```
(let (... a aa  ... ab  ... ac ... )
  ... )
```

Environnement global figé

On peut maintenant imaginer que, pour tout nom, il y a au plus une variable globale de ce nom. Ceci est le cas d'une application compilée et autonome sans création dynamique de code (pas d'appel à `eval`) ; c'était aussi le cas des précédents interprètes qui n'autorisaient pas la création de nouvelles variables globales. Elles devaient être créées explicitement par la forme (du langage d'implantation) `definitial`.

Dans un tel environnement, une variable globale n'existe qu'après avoir été créée par `define`. On ne peut obtenir ou modifier sa valeur que si elle a auparavant été définie. Comme cependant il ne peut y avoir au plus qu'une seule variable de ce nom, il est permis de référencer la variable avant qu'elle ne soit définie : c'est ce qui permet notamment la récursion mutuelle. Il n'est par contre point permis de définir plusieurs fois une variable globale. Nous pouvons résumer ces propriétés par notre, maintenant classique, tableau :

Référence	x
Valeur	x mais x doit exister
Modification	`(set! x ...)` mais x doit exister
Extension	`define` (une seule fois)
Définition	non

Pour illustrer cet environnement, considérez encore le fragment précédent et les résultats auxquels il conduit :

```
g                              ; error:  g uninitialized
(define (P m) (* m g))         ; forward reference to  g
(define g 10)
(define g 9.81)                ; error: redefinition of  g
(set! g 9.81)                  ; modification of  g
(P 10)                 → 98.1
(set! e 2.78)                  ; error: no variable  e
```

Cet environnement permet de donner une idée de la notion de programme. Un programme est défini comme une suite d'expressions $\pi_1 \ldots \pi_n$ que l'on peut regrouper en une seule expression ainsi construite : on enserre les formes $\pi_1 \ldots \pi_n$ dans une unique forme `let` introduisant comme variables locales non initialisées toutes les variables libres présentes dans $\pi_1 \ldots \pi_n$ et l'on modifie toutes les formes `define` par des formes équivalentes `set!`. Pour fixer plus complètement les idées, une petite application pourrait être écrite en Scheme comme :

```
(define (crypt pw) ...)
(let ((passwd "timhukiTrolrk"))
  (set! can-access? (lambda (pw) (string=? passwd (crypt pw)))) )
(define (gatekeeper)
  (until (can-access? (read)) (gatekeeper)) )
```

Cette petite application exige un mot de passe de l'utilisateur et ne le laisse franchir cette épreuve que s'il lui soumet le bon mot. Ce Cerbère informatique est donc équivalent à :

```
(let (crypt make-can-access? can-access? gatekeeper)
(set! crypt (lambda (pw) ...))
(set! make-can-access? (lambda (passwd)
                         (lambda (pw)
                            (string=? passwd (crypt pw)) ) ))
(set! can-access? (make-can-access? "timhukiTrolrk"))
(set! gatekeeper
      (lambda () (until (can-access? (read)) (gatekeeper))) )
(gatekeeper) )
```

```
car≡ car
cons≡ cons
...
```

Les définitions globales sont transformées en des définitions locales affectant les variables libres de l'application, regroupées, non initialisées, dans le `let` englobant [1]. Bien sûr les fonctions usuelles comme `read`, `string=?`, `string-reverse` ou `string-append` sont considérées comme visibles. Dans ce monde, l'environnement global est fini et restreint aux variables prédéfinies (comme `car` ou `cons`) et aux variables libres du programme. Il n'est toutefois pas possible d'affecter une variable libre non présente dans l'environnement global puisque ce dernier a été conçu pour éviter cette situation. La seule façon de provoquer cette erreur serait de disposer d'une forme `eval` permettant l'évaluation dynamique de code.

Environnement global automatiquement extensible

La présence d'une boucle fondamentale d'interaction demande d'augmenter dynamiquement l'environnement global. On vient de voir que la forme `define` pouvait être mise à contribution pour augmenter l'ensemble des variables globales. On peut aussi penser qu'une affectation peut suffire pour la même tâche, c'est-à-dire qu'une affectation sur une variable libre équivaut à une définition de cette variable dans l'environnement global si elle n'y figurait pas auparavant. On pourra ainsi écrire directement :

```
(let ((name "Nemo"))
  (set! winner (lambda () name))
  (set! set-winner! (lambda (new-name) (set! name new-name)
                                       name )) )
```

Avec la précédente variante, il aurait fallu préfixer cette expression par deux définitions absurdes comme :

```
(define winner      'sans-queue)
(define set-winner! 'ni-tete)
```

1. On pourra remarquer que cette définition de programme donne un sens à de multiples définitions d'une même variable ; il faudrait donc ajouter quelques contraintes syntaxiques.

Ces définitions auraient créés les deux variables, les auraient initialisées n'importe comment puis les auraient immédiatement modifiées pour qu'elles prennent leur véritable valeur. Cette technique est gênante car elle rend explicitement modifiables les variables définies, puisqu'elles ont fait l'objet d'au moins une affectation. C'est pourquoi, il y a fort longtemps, en Scheme, [SS78b], existait la forme `static` qui aurait permis d'imaginer écrire [2] :

```
(let ((name "Nemo"))
  (define (static winner) (lambda () name))
  (define (static set-winner!)
    (lambda (new-name) (set! name new-name)
                       name ) ) )
```

Les deux variables globales auraient ainsi pu être codéfinies ensemble dans un contexte lexical local sans user d'affectation.

Si l'affectation permet de créer des variables globales, on risque de polluer l'environnement global. On peut aussi penser qu'il serait plus astucieux de ne créer la variable que localement et, par exemple, au niveau du dernier `let` lexicalement ou du dernier `prog` [Nor72] dynamiquement englobant [3]. Ces dernières idées brisent malheureusement la transparence référentielle et interdisent les codéfinitions précédentes.

Environnement global hyperstatique

Une dernière sorte d'environnement global existe où plusieurs variables globales peuvent être associées à un nom mais où seules les formes situées après une définition peuvent la voir. Si nous reprenons notre exemple préféré :

```
g                              ; error: no variable  g
(define (P m) (* m g))         ; error: no variable  g
(define g 10)
(define (P m) (* m g))
(P 10)                  → 100
(define g 9.81)
(P 10)                  → 100  ; P sees the old  g
(define (P m) (* m g))
(P 10)                  → 98.1
(set! e 2.78)                  ; error: no variable  e
```

Ici, fidèle à l'esprit qui veut que l'on close les fonctions dans l'environnement où elles sont créées, la première définition de `P` est statiquement erronée puisqu'elle référence `g` qui n'existe pas à ce moment. Une autre solution serait de permettre de définir `P` mais de rendre erroné tout appel à `P` puisque `g` n'avait pas de valeur à l'instant où `P` a été créée.

La seconde définition de `P` enclôt que `g` vaut 10 ce qui durera autant que la fonction `P`. Si d'aventure l'on désire une meilleure approximation de `g`, il faudra redéfinir la fonction `P` pour prendre en compte cette nouvelle valeur. L'environnement global est géré de façon complètement lexicale, on le qualifie d'*hyperstatique* et c'est ce mode qu'a choisi ML. On peut le synthétiser par notre tableau habituel.

---

2. Il faudrait cependant changer la syntaxe des formes `define` internes pour qu'elle reconnaissent la présence locale de définitions globales.

3. C'est aussi ce que fait TEX : une définition (par `\def`) disparaît lorsque l'on sort du groupe courant (par `\endgroup`).

Référence	x mais x doit exister
Valeur	x mais x doit exister
Modification	`(set! x ...)` mais x doit exister
Extension	`define`
Définition	non

Ce mode pose cependant le problème des définitions récursives. Il faut pouvoir à la fois définir des fonctions simplement récursives ou des groupes de fonctions mutuellement récursives. ML a un mot-clé, `rec`, pour indiquer la première chose et un mot-clé, `and`, pour indiquer la codéfinition. Ils correspondent aux `letrec` et `let` de Scheme qui, eux, autorisent déjà des définitions multiples. Plus picturalement parlant, l'exemple précédent correspond à une série de `let` ou `letrec` gigognes :

```
g                                          ; error: no variable  g
(let ((P (lambda (m) (* m g))))            ; error: no variable  g
  (let ((g 10))
    (let ((P (lambda (m) (* m g))))
      (P 10)
      (let ((g 9.81))
        ... ) ) ) )
```

Et voici un autre exemple en ML de fonctions mutuellement récursives :

```
let rec odd n = if n = 0 then false else even (n - 1)
    and even n = if n = 0 then true else odd (n - 1)
```

Cela correspondrait bien évidemment à :

```
(letrec ((odd? (lambda (n) (if (= n 0) #f (even? (- n 1)))))
         (even? (lambda (n) (if (= n 0) #t (odd? (- n 1))))) )
  ... )
```

Les environnements globaux hyperstatiques ont le grand avantage de pouvoir détecter statiquement les variables indéfinies (et, si l'immuabilité de la liaison était connue, de pouvoir compiler efficacement ce qui suivra en profitant de la connaissance de la valeur) mais ont le désavantage de requérir, lors d'une erreur, la redéfinition de tout ce qui suit la définition erronée.

4.1.3 Affectation sur variable prédéfinie

Parmi toutes les variables libres qui apparaissent dans un programme, se trouvent des variables prédéfinies comme `car` ou `read`. Les affecter est donc un droit imprescriptible, mais quel est alors le sens de ces affectations ? La vitesse des implantations dépend souvent d'hypothèses qui ne sont pas toujours explicites. En Lisp, on parle souvent de fonctions intégrables, c'est-à-dire compilées en ligne (*inlined function*) ; les accesseurs comme `car` ou `cdr` sont généralement intégrables. Les redéfinir pose problème puisque, dès qu'une affectation portant sur `car` est perçue, tous les appels à `car` doivent utiliser la valeur courante de `car` plutôt que la valeur primitive. Dans une boucle d'interaction hyperstatique, le problème n'en est pas un puisque seules les futures expressions verront cette fonction `car` modifiée. Si, par contre, la boucle d'interaction est dynamique, il faudrait, pour bien faire, recompiler tous les appels en ligne à `car`. Cela amènerait logiquement à redéfinir la fonction `cadr` qui est probablement définie comme la composée de `car` et `cdr`. Ce travail n'est jamais fait car il induit trop de dérangement et ne correspond probablement pas aux intentions profondes des

utilisateurs. Il est d'ailleurs interdit, en Scheme, que la modification d'une liaison globale modifie les valeurs ou les comportements des autres fonctions prédéfinies. Ainsi modifier `car` est illicite mais définir une nouvelle variable `car` est permis et ne saurait changer `cadr`. Comme `map` en Scheme (`mapcar` en Lisp) apparaît dans le standard définissant Scheme, une modification de `car` ne saurait la perturber. Comme `mapc` n'y apparaît pas, modifier `car` est donc susceptible de l'altérer. Modifier une variable globale est souvent un moyen de tracer ou analyser les appels à la fonction qui est la valeur de cette variable globale. C'est aussi un moyen pour corriger un comportement erroné voire aberrant. On ne saurait trop conseiller à notre impétueuse jeunesse de se tenir à l'écart de telles tentations : aux variables prédéfinies, point ne touchera !

En résumé, un environnement global hyperstatique a un comportement logique mais n'est toutefois pas aussi pratique qu'un environnement global dynamique pour la mise au point.

4.2 Effets de bord

On a vu que le calcul en Lisp était représenté par un triplet : expression, environnement et continuation. Pour aussi séduisant que soit ce triplet, il ne permet d'exprimer le sens ni de l'affectation ni de la modification physique de données en mémoire. Les *effets de bord* (pour *side effects*, littéralement « effets secondaires ») correspondent à des altérations de l'état du monde et, en particulier, à des changements dans l'état de la mémoire. Les plus connues de ces altérations proviennent des modificateurs physiques de données en mémoire tels que `set-car!` et des fonctions d'entrées-sorties. Lire (ou écrire) dans un flux séquentiel fait progresser une marque désignant le dernier endroit lu ou écrit. Écrire sur un écran le souille irrémédiablement ; c'est encore plus indélébile sur du papier. Requérir de l'utilisateur qu'il tape sur son clavier est encore plus irréversible. Les effets de bord sont malheureusement omniprésents en informatique, qu'ils correspondent aux jeux d'instructions des ordinateurs opérant sur des registres ou aux systèmes de fichiers conservant, entre deux sessions et, plus si possible les merveilleux textes et programmes que vous concoctez. Bien sûr, il est possible d'imaginer vivre dans un monde idyllique sans effet de bord, mais ce serait là soufrir d'une forme aiguë d'autisme informatique puisque l'on ne pourrait alors plus communiquer les résultats de ses calculs. Ce rêve n'est toutefois guère gênant car il n'existe pas d'ordinateur procurant un tel monde et y pénétrer serait un effet de bord qui l'annihilerait.

On a pu voir précédemment que les affectations pouvaient être simulées par des boîtes. Peut-on inversement simuler des paires pointées sans paires pointées ? La réponse est oui, mais en usant d'affectation ! Une paire pointée peut-être simulée par une fonction répondant aux messages `car`, `cdr`, `set-car!` et `set-cdr!` [4] :

```
(define (kons a d)
  (lambda (msg)
    (case msg
      ((car) a)
      ((cdr) d)
```

4. Pour ne point les confondre avec les primitives usuelles, nous préfixerons leur nom par la lettre `k`.

```
      ((set-car!) (lambda (new) (set! a new)))
      ((set-cdr!) (lambda (new) (set! d new))) ) ) )
(define (kar pair)
  (pair 'car) )
(define (set-kdr! pair value)
  ((pair 'set-cdr!) value) )
```

Une fois encore la simulation n'est pas parfaite [Fel90] car l'on ne peut plus reconnaître les paires pointées des fonctions normales. On ne peut avec cette programmation écrire le prédicat `pair?`. Hormis cette légère diminution de moyens qui est en fait équivalente à la capacité de créer de nouveaux types de données, on voit bien que les effets de bord et l'affectation entretiennent des rapports troublants et que l'un par l'autre est simulable. Il faut donc les bannir tous ou accepter leur présence et leurs plus ou moins incontrôlables effets. Choisir entre l'affectation ou les modifications physiques dépend du contexte et des propriétés que l'on cherche à préserver.

4.2.1 Égalité

L'inconvénient des modificateurs physiques est qu'ils induisent une altération de l'égalité. Quand peut-on dire que deux objets sont égaux ? La notion d'égalité, au sens de Leibniz, définit que deux objets sont égaux si l'on peut les substituer sans dommage l'un par l'autre ou, en termes plus programmatiques, si l'on ne dispose d'aucun moyen pour les différencier. L'égalité tautologique signifie qu'un objet est égal à lui-même : on peut toujours le remplacer par lui-même, ce qui bien sûr ne change rien. C'est une bien faible notion de l'égalité si on la prend au pied de la lettre mais c'est celle qui est applicable aux nombres entiers. Là où cela se complique est lorsque l'on considère deux objets dont on ignore les rapports (parce qu'ils sont, par exemple, issus de deux calculs différents comme `(* 2 2)` et `(+ 2 2)`) et que l'on s'interroge sur leur substituabilité ou leur ressemblance.

En présence de modificateurs physiques, deux objets sont différents si une altération de l'un ne provoque aucun changement de l'autre. Notez que c'est ici une définition de la différence qui est proposée et non celle de l'égalité. Il s'avère donc que l'on peut classer les objets en deux classes : ceux qui sont muables et ceux qui sont immuables.

Dieu nous ayant fait cadeau des entiers (*dixit* Kronecker), il est logique de les considérer comme immuables : 3 est égal à 3 dans tous les contextes. Il est simple pour un objet non composite, c'est-à-dire sans constituant, d'être immuable. À part les entiers, les caractères, les booléens et la liste vide, il n'est pas d'autres objets immuables et non composites.

Pour les objets composites comme les listes, les vecteurs, les chaînes de caractères, il paraît logique de les considérer comme égaux s'ils ont les mêmes constituants : c'est ce que l'on trouve généralement sous le nom de `equal?` avec différentes variantes récursives ou pas [5]. Malheureusement, en présence de modificateurs physiques, l'égalité des constituants à un instant donné ne permet guère d'assurer une persistance de cette égalité dans le futur. On introduit alors un nouveau prédicat d'égalité physique : `eq?` :

5. Voir par exemple les fonctions `equalp`, `tree-equal` en COMMON LISP.

deux objets sont `eq?` si en fait ils sont le même [6]. Ce prédicat est souvent implanté de façon extrêmement efficace par comparaison de pointeurs et permet donc de tester si deux pointeurs désignent le même emplacement en mémoire.

En définitive, on dispose de deux prédicats `eq?` et `equal?`, testant l'un l'identité physique, l'autre l'équivalence structurelle. Toutefois ces deux prédicats extrêmes négligent la muabilité des objets. Deux objets immuables sont égaux s'ils ont des constituants égaux. Acquérir la certitude que deux objets immuables sont égaux à un instant donné permet d'assurer qu'ils le seront toujours car ils ne varient pas. Inversement deux objets muables ne peuvent être sempiternellement égaux que s'ils sont un même et unique objet. En fait, un seul concept semble émerger, celui de substituabilité : deux objets sont égaux si aucun programme ne peut les distinguer.

Le prédicat `eq?` ne compare que des entités comparables du point de vue de l'implantation : des adresses en mémoire [7] ou des constantes immédiates. Il se trouve que les techniques de représentation classiques des objets dans les dialectes que nous considérons n'assurent pas que deux objets égaux aient des représentations comparables pour `eq?` [8]. Il s'ensuit que `eq?` n'implante même pas exactement la tautologie. Un nouveau prédicat, nommé `eqv?` en Scheme et `eql` en COMMON LISP, améliore `eq?`. En termes grossiers, il se comporte comme `eq?` mais un peu plus lentement et assure que deux objets immuables égaux seront reconnus comme tels. Le prédicat `eqv?` assure notamment que deux nombres égaux, même gigantesques et représentés par ce que l'on nomme couramment des *bignum*s, seront comparés avec succès indépendamment de leur représentation en mémoire.

Si l'on se remémore les interprètes que nous avons donnés, on peut remarquer le faible nombre de données muables qui s'y trouvaient. À l'exception des liaisons, tout était immuable. De fait les structures muables ne sont pas légion et la majorité des paires pointées allouées ne sont jamais soumises aux modificateurs physiques que sont `set-car!` et `set-cdr!`. En ML mais aussi en COMMON LISP, il est possible d'indiquer la muabilité des champs d'un objet. On pourrait donc créer la paire pointée constante :

COMMON LISP
```
(defstruct immutable-pair
   (car '() :read-only t)
   (cdr '() :read-only t) )
```

Physiquement les paires pointées constantes pourraient être allouées dans une zone à part et les constantes citées dans les programmes pourraient utiliser ce type de paire pointée.

Henry Baker [Bak93] a proposé d'unifier tous les prédicats d'égalité en un seul, `egal`, défini comme suit : si les objets à comparer sont muables, `egal` se comporte comme `eq?` sinon comme `equal?`. Ce nouveau prédicat assure la substituabilité de deux objets dès lors qu'ils sont reconnus égaux par `egal`. On voit tout l'intérêt de ce prédicat dans un monde parallèle lorsqu'il s'agira de migrer des données [QD93, Que94].

Les donnés cycliques sont une autre source de problèmes. Les comparer est pos-

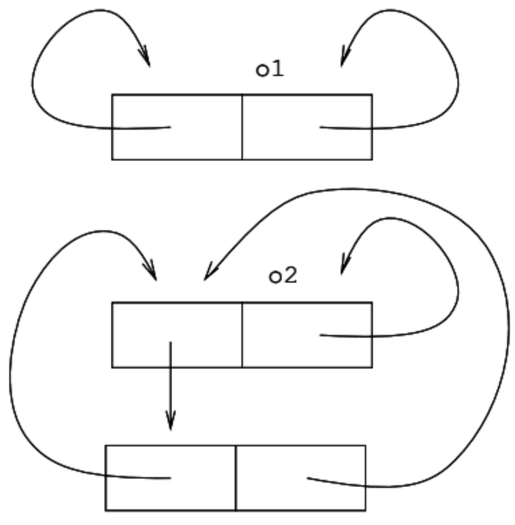
---

6. La bizarrerie grammaticale exprime bien qu'il n'y a pas deux objets mais un seul.

7. Sur un réseau d'ordinateurs pour un Scheme distribué, `eq?` doit pouvoir comparer des objets situés sur des sites différents, ce qui rend `eq?` non nécessairement immédiat.

8. Par exemple, `(eq? 33 33)` n'est pas garanti retourner `#t` en Scheme à la différence de `eqv?`.

sible mais coûteux. Ignorer que les données à comparer peuvent être cycliques peut conduire à faire boucler `equal?`. Et si cela ne suffisait pas, comparer ces structures ne semble pas revêtir un caractère d'évidence. Considerez :




```
(define o1 (let ((pair (cons 1 2)))
             (set-car! pair pair)
             (set-cdr! pair pair)
             pair ))
(define o2 (let ((pair (cons (cons 1 2) 3)))
             (set-car! (car pair) pair)
             (set-cdr! (car pair) pair)
             (set-cdr! pair pair)
             pair ))
```

Supposons tout d'abord avoir à évaluer `(equal? o1 o1)`, si l'implantation d'`equal?` commence par tester si les objets sont `eq?` avant de se lancer dans la comparaison structurelle de leurs champs respectifs, alors la réponse est immédiate et positive. Dans le cas contraire, `equal?` va irrémédiablement boucler[9]. Si `(equal? o1 o1)` est déjà suspect, que penser de `(equal? o1 o2)` ? La réponse tient encore une fois dans la connaissance de l'immuabilité des paires pointées qui composent `o1` et `o2`. Si personne ne peut les modifier alors, en absence de `eq?`, on ne peut écrire de programmes les distinguant. On retiendra qu'il vaut mieux s'abstenir de comparer des structures cycliques par `equal?`.

Les prédicats précédents couvraient bien les divers types de données usuellement présents à l'exception des symboles et des fonctions. Les symboles sont des structures de données complexes car souvent utilisées par les implantations pour y stocker certaines informations concernant les variables globales éponymes. Fondamentalement, un symbole est une structure de données que l'on peut retrouver par son nom et qui est garantie unique pour un nom donné.

Les symboles contiennent souvent une liste de propriétés. C'est un ajout dangereux car cette dernière est uniquement entretenue et manipulée par des effets de bord globalement perçus. La liste de propriétés est de plus souvent onéreuse tant à l'archivage (2 paires pointées par propriété) qu'à l'utilisation (recherche linéaire). On lui préférera donc les tables à adressage dispersé ou *tables de hachage*[10].

4.2.2 Égalité de fonctions

Pour les fonctions, le cas peut sembler presque désespéré. On peut définir l'égalité de deux fonctions en posant que deux fonctions sont égales si, à égalité d'arguments, correspond l'égalité des résultats.

$$f = g \Leftrightarrow \forall x, f(x) = g(x)$$

C'est, infortunément, un problème indécidable que de comparer des fonctions. On peut donc refuser de comparer des fonctions ou bien adopter une attitude un peu plus souple mais en restreignant le problème. De larges classes de fonctions sont comparables et, dans de nombreux cas, on peut facilement découvrir que deux fonctions ne peuvent être égales, par exemple, quand elles n'ont pas le même nombre d'arguments.

---

9. En écrivant cela, je viens de vérifier sur quatre implantations de Scheme et en ai trouvé deux qui bouclaient. Je ne donnerai pas de noms car elles en ont parfaitement le droit [CR91b] !

10. Leur invention est, à mon sens, l'une des plus immenses découvertes jamais faites en informatique.

On peut donc décider de procurer une approximation du prédicat d'égalité sur les fonctions en admettant que parfois sa réponse soit imprécise voire erronée. On ne peut utiliser un tel prédicat que si l'on sait exactement comment il est imprécis. Scheme définit `eqv?` sur des fonctions de la façon suivante : si l'on doit comparer les fonctions `f` et `g`, alors s'il existe une application de `f` et `g` à de mêmes (lorsque comparés par `eqv?`) arguments telle que les résultats soient différents alors les fonctions `f` et `g` ne sont pas `eqv?`. On notera qu'ici encore nous sommes face à une apologie de la différence plutôt qu'à une définition de l'égalité.

Donnons quelques exemples. La comparaison `(eqv? car cdr)` doit retourner faux car, manifestement et sur l'exemple `(a . b)`, les résultats sont différents.

Il peut aussi paraître évident que `(eqv? car car)` retourne vrai puisque l'égalité ne saurait être moins définie que la tautologie. C'est faux dans certains dialectes pour qui cette forme était équivalente à `(eqv? (lambda (x) (car x)) (lambda (x) (car x)))` du fait de l'intégrabilité de la fonction `car`. De fait, il n'est pas précisé dans le R^{4}RS ce que doit retourner `eqv?` lorsqu'employée à comparer des fonctions prédéfinies comme `car`.

Sans invoquer la tautologie, comment comparer `cons` et `cons` ? La fonction `cons` est un allocateur de données muables et donc, même si les arguments sont semblables, les résultats ne sont pas égaux car alloués en des emplacements différents. On est donc en droit d'exiger que `cons` ne soit pas égal à `cons` et donc `(eqv? cons cons)` devrait retourner faux puisqu'il est faux que `(eqv? (cons 1 2) (cons 1 2))` ! D'ailleurs est-il véritablement vrai que `car` soit égal à `car` comme au paragraphe précédent ? Dans un langage sans type, `(car 'foo)` est erroné, signale une exception puis est laissé aux bons soins du ratrappeur d'erreur local dont le comportement est donc inconnu et peut varier. Ainsi n'est-on point assuré que cet appel rende des résultats toujours favorablement comparables. Cela est aussi vrai de toute fonction partielle employée en dehors de son domaine de définition.

Comment rompre cette situation ? Certains dialectes de ML, du fait de leur typage qui leur permet de détecter tous les endroits où sont effectués des comparaisons entre fonctions, les interdisent purement et simplement puisque l'on ne peut et ne doit comparer des fonctions. Scheme, dans sa sémantique, fait de `lambda` un constructeur de fermetures. Chaque forme `lambda` alloue donc une nouvelle fermeture quelque part en mémoire. Cette fermeture a une certaine adresse, la comparaison de fonctions peut donc s'effectuer par comparaison d'adresses et le seul cas où deux fonctions sont les mêmes (par `eqv?`) est lorsqu'elles ne sont qu'une seule et même fermeture.

Ce comportement empêche certaines améliorations communes au λ-calcul. Soit la fonction, astucieusement baptisée `make-named-box`, elle prend un unique argument : le message, l'analyse et répond de façon appropriée. C'est une des réalisations possibles des objets et c'est celle que nous adopterons pour le codage de l'interprète de ce chapitre.

```
(define (make-named-box name value)
  (lambda (msg)
    (case msg
      ((type) (lambda () 'named-box))
      ((name) (lambda () name))
      ((ref)  (lambda () value))
      ((set!) (lambda (new-value) (set! value new-value))) ) ) )
```

Cette fonction crée une boîte non anonyme puisque dotée d'un nom. La fermeture obtenue grâce au message `type` ne dépend aucunement des variables locales. On pourrait donc réécrire la fonction toute entière en :

```
(define other-make-named-box
  (let ((type-closure (lambda () 'named-box)))
    (lambda (name value)
     (let ((name-closure  (lambda () name))
           (value-closure (lambda () value))
           (set-closure   (lambda (new-value)
                            (set! value new-value) )) )
       (lambda (msg)
         (case msg
           ((type) type-closure)
           ((name) name-closure)
           ((ref)  value-closure)
           ((set!) set-closure) ) ) ) ) ) )
```

La différence avec la précédente version est que, par exemple, la fermeture `(lambda () 'named-box)` n'est allouée qu'une unique fois tandis qu'auparavant elle était allouée chaque fois que nécessaire. Mais alors que vaut la comparaison suivante ?

```
(let ((nb (make-named-box 'foo 33)))
  (comparer? (nb 'type) (nb 'type)) )
```

La question est mal posée car elle n'a de sens que si l'on précise le prédicat de comparaison. Si l'on utilise `eq?` alors le nombre exact d'allocations de `(lambda () 'named-box)` jouera un rôle tandis que si l'on use de `egal`, la question est vidée de son sens et la valeur finale est toujours vraie. On le voit donc, la présence d'un comparateur physique dans un langage rend discernables des objets éventuellement égaux.

Lorsque `lambda` est un allocateur, on associe une adresse en mémoire à toutes les fermetures qui sont créées. Deux fermetures ayant la même adresse ne sont qu'un même objet et donc égales. Ainsi donc, en Scheme :

```
(let ((f (lambda (x y) (cons x y))))
  (eqv? f f) )    → #t
```

Comparant le même objet en mémoire, le prédicat `eqv?` renvoie vrai même s'il est faux que `(eqv? (f 1 2) (f 1 2))`.

On le constate, comparer des fonctions est un chemin semé d'embûches pour lequel plusieurs points de vue peuvent être adoptés suivant les propriétés que l'on souhaite préserver. Conserver l'aspect mathématique est entâché d'infaisabilité, suivre l'implantation occulte un large pan d'emploi. Quoique ce livre contienne un exemple d'une telle comparaison dans l'implantation de MEROONET, **[☞ p. 424]** on retiendra qu'il vaut mieux éviter, autant que faire se peut, de comparer des fonctions.

4.3 Implantation

Pour changer, l'interprète que nous allons expliciter n'emploiera que des fermetures pour ses structures de données : tout sera codé par des formes `lambda` et par envoi de messages. Tous les objets seront donc des fermetures basées sur des fonctions de code similaire à `(lambda (msg) (case msg ...))`, comme on en a vu quelques exemples précédemment. Quelques messages seront standard comme `boolify` qui

permet d'associer à toute valeur sa valeur de vérité équivalente (c'est-à-dire `#t` ou `#f` pour les besoins de l'alternative) ou `type`, le bien nommé, qui retournera son type.

Le principal problème est de trouver une méthode pour décrire les effets de bord. Si l'on y songe, une variable référence une liaison qui elle-même est associée à une valeur. Plus prosaïquement, une variable pointe sur une boîte (une adresse) qui contient une valeur. La mémoire n'est qu'une fonction associant à des adresses des valeurs contenues dans ces adresses. L'environnement est une autre fonction associant des adresses à des variables. Cela paraît tout à fait naturel dès lors que l'on se soucie de simuler l'implantation.

La mémoire doit être partout visible. On pourrait en faire une variable globale mais ce n'est pas très beau et on a vu tous les problèmes que posait l'environnement global. Une autre solution est de faire en sorte que toutes les fonctions voient la mémoire. Il suffit donc que toutes les fonctions reçoivent en argument la mémoire et la transmettent aux autres, éventuellement modifiée. C'est cette démarche que nous adopterons et qui nous pousse donc à considérer maintenant qu'un calcul est un quadruplet : *expression, environnement, continuation* et *mémoire*. Ces quatre composantes seront nommées respectivement `e`, `r`, `k` et `s`. Afin d'assurer la circulation de la mémoire, les continuations recevront non seulement la valeur à transmettre mais aussi l'état mémoire résultant.

Comme d'habitude la fonction `evaluate` analysera syntaxiquement son argument et appellera la fonction appropriée. Nous conserverons les conventions de nommage des variables du chapitre précédent auxquelles nous ajouterons :

`e, et, ec, ef ...`	expression, forme
`r ...`	environnement
`k, kk ...`	continuation
`v, void ...`	valeur (entier, paire, fermeture etc.)
`f ...`	fonction
`n ...`	identificateur
`s, ss, sss ...`	mémoire
`a, aa`	adresse (boîte)

Comme le nombre de variables augmente, nous nous efforcerons de toujours les énoncer dans le même ordre : `e`, `r`, `s` puis `k`. Voici donc l'interprète :

```
(define (evaluate e r s k)
  (if (atom? e)
      (if (symbol? e) (evaluate-variable e r s k)
          (evaluate-quote e r s k) )
      (case (car e)
        ((quote)  (evaluate-quote (cadr e) r s k))
        ((if)     (evaluate-if (cadr e) (caddr e) (cadddr e) r s k))
        ((begin)  (evaluate-begin (cdr e) r s k))
        ((set!)   (evaluate-set! (cadr e) (caddr e) r s k))
        ((lambda) (evaluate-lambda (cadr e) (cddr e) r s k))
        (else     (evaluate-application (car e) (cdr e) r s k)) ) ) )
```

4.3.1 Alternative

L'alternative introduit une continuation auxiliaire attendant la valeur de la condition.

```
(define (evaluate-if ec et ef r s k)
  (evaluate ec r s
    (lambda (v ss)
      (evaluate ((v 'boolify) et ef) r ss k) ) ) )
```

La continuation auxiliaire reçoit non seulement la valeur `v` booléenne à prendre en considération mais aussi l'état mémoire `ss` résultant de l'évaluation de la condition. Il est en effet licite d'opérer des effets de bord dans la condition comme le montre l'expression suivante :

```
(if (begin (set-car! pair 'foo)
           (cdr pair) )
    (car pair) 2 )
```

La modification commise dans la condition doit être perçue des deux branches de l'alternative. Il serait tout à fait différent de définir l'alternative comme :

```
(define (evaluate-amnesic-if ec et ef r s k)
  (evaluate ec r s
    (lambda (v ss)
      (evaluate ((v 'boolify) et ef) r s   ; s ≠ ss!
                k ) ) ) )
```

Dans ce dernier cas, il y aurait restauration de l'état mémoire présent au début de l'évaluation de la condition. Ce trait pourrait être présent dans un langage procurant un retour arrière à la Prolog. **[☞ Ex. 4.4]**

Pour tenir compte que tout objet est codé par une fonction, Vrai et Faux ne doivent point être les `#t` et `#f` du langage d'implantation. Comme tout objet peut aussi être considéré comme une valeur de vérité, on supposera que tout objet peut recevoir le message `boolify` et retournera alors l'un des λ-calculesque combinateurs `(lambda (x y) x)` et `(lambda (x y) y)`.

4.3.2 Séquence

Bien que non essentielle, la séquence montre bien comment est traité le séquencement dans le langage. Elle met aussi en œuvre une continuation intermédiaire. On supposera pour ne pas s'encombrer que les séquences comprennent au moins un terme.

```
(define (evaluate-begin e* r s k)
  (if (pair? (cdr e*))
      (evaluate (car e*) r s
        (lambda (void ss)
          (evaluate-begin (cdr e*) r ss k) ) )
      (evaluate (car e*) r s k) ) )
```

On peut noter, sous une forme particulièrement dépouillée, l'impavide mépris dans lequel est tenue la valeur de `void` et l'appel terminal à `evaluate` lorsqu'une seule forme subsiste dans la séquence.

4.3.3 Environnement

L'environnement doit être codé comme une fonction et transformer des variables en des adresses. Similairement la mémoire est aussi représentée comme une fonction mais qui transforme des adresses en des valeurs. Initialement, l'environnement ne contient rien :

```
(define (r.init id)
  (wrong "No binding for" id) )
```

Comment modifier un environnement ou une mémoire sans modificateur physique ni affectation ? Modifier un état mémoire, c'est changer la valeur associée à une adresse. On peut donc définir la fonction `update` apte à modifier une mémoire pour peu qu'on lui propose une adresse et une valeur :

```
(define (update s a v)
  (lambda (aa)
    (if (eqv? a aa) v (s aa)) ) )
```

Le sens est apparent pour peu que l'on se souvienne qu'un état mémoire est représenté par une fonction transformant des adresses en valeurs. La forme `(update s a v)` retourne un nouvel état mémoire fidèlement semblable à `s` sauf qu'en `a` la valeur associée est maintenant `v` ; pour les autres, consulter `s`. On procurera une généralisation de `update` qui prendra plusieurs adresses et plusieurs valeurs à la fois.

```
(define (update* s a* v*)
  ;; (assume (= (length a*) (length v*)))
  (if (pair? a*)
      (update* (update s (car a*) (car v*)) (cdr a*) (cdr v*))
      s ) )
```

La fonction `update` peut aussi être appliquée à l'extension des environnements, ce serait en ML une fonction polymorphe. Au vu de la comparaison qu'effectue `update`, on s'arrangera pour coder les adresses par des objets non indûment comparables : des entiers. Pour comparer les variables, on comparera leur nom. En Scheme, la fonction `eqv?` conviendra donc dans ces deux cas.

4.3.4 Référence à une variable

La valeur d'une variable s'exprime donc simplement :

```
(define (evaluate-variable n r s k)
  (k (s (r n)) s) )
```

La forme `(r n)` retourne l'adresse où se trouve la valeur de la variable, valeur de `n`. Le contenu de cette adresse est cherché en mémoire et rendu à la continuation. Comme la recherche de la valeur d'une variable est un processus non destructif, la mémoire n'est pas modifiée et est transmise telle quelle à la continuation.

4.3.5 Affectation

L'affectation requiert une continuation intermédiaire.

```
(define (evaluate-set! n e r s k)
  (evaluate e r s
    (lambda (v ss)
      (k v (update ss (r n) v)) ) ) )
```

Après avoir évalué le second terme, sa valeur et la mémoire résultante sont fournies à la continuation qui met à jour le contenu de l'adresse associée à la variable, c'est-à-dire fabrique un nouvel état mémoire qui est rendu à la continuation originelle de l'affectation. Par ailleurs, cette affectation retourne la valeur affectée mais pourrait aussi bien retourner l'ancienne valeur de la variable sachant que cette dernière spécification est ambiguë. **[☞ Ex. 4.5]**.

On peut admirer la fonctionnalisation de la mémoire qui permet de représenter les modifications qu'elle subit sans aucun effet de bord. En termes plus lispiens, cette représentation de la mémoire l'assimile à la liste de toutes les modifications qu'elle a connues. Comparée à une vraie mémoire, cette représentation est manifestement superfétatoire mais autorise la manipulation de diverses mémoires en même temps.

4.3.6 Application fonctionnelle

L'application fonctionnelle consiste en l'évaluation de tous les termes, on choisira ici l'ordre gauche-droite.

```
(define (evaluate-application e e* r s k)
  (define (evaluate-arguments e* r s k)
    (if (pair? e*)
        (evaluate (car e*) r s
          (lambda (v ss)
            (evaluate-arguments (cdr e*) r ss
              (lambda (v* sss)
                (k (cons v v*) sss) ) ) ) )
        (k '() s) ) )
  (evaluate e r s
    (lambda (f ss)
      (evaluate-arguments e* r ss
        (lambda (v* sss)
          (if (eq? (f 'type) 'function)
              ((f 'behavior) v* sss k)
              (wrong "Not a function" (car v*)) ) ) ) ) ) )
```

La fonction, usuellement nommée `evlis`, a ici été rendue locale sous le nom `evaluate-arguments`. Elle évalue ses arguments dans l'ordre et rassemble leur valeur en une liste. La fonction, valeur du premier terme de l'application fonctionnelle, est alors appliquée à ses arguments accompagnés de la mémoire et de la continuation d'appel. Là encore, on peut apprécier la concision de ce programme.

La fonction étant représentée par une fermeture qui répond au moins au message `boolify`, un nouveau message `behavior` extraiera son comportement fonctionnel, c'est-à-dire la façon dont elle se calcule.

4.3.7 Abstraction

Par souci de simplification, on supposera pour l'heure, que la forme spéciale `lambda` ne permet de créer que des fonctions d'arité fixe. Deux effets se conjuguent ici, une allocation en mémoire et la construction d'une valeur de première classe : une fermeture. La création d'une fonction entraine la fabrication d'un objet qui, bien entendu, se trouve quelque part alloué en mémoire. Il est donc nécessaire que la mémoire soit affectée lorsqu'une fonction est créée : on fournira donc une adresse à l'utilitaire

`create-function` ainsi que le comportement de la fonction à créer. C'est ce comportement qu'extraiera d'une fonction le message `behavior` comme précédemment lors de l'application fonctionnelle.

```
(define (evaluate-lambda n* e* r s k)
  (allocate 1 s
    (lambda (a* ss)
      (k (create-function
          (car a*)
          (lambda (v* s k)
            (if (= (length n*) (length v*))
                (allocate (length n*) s
                 (lambda (a* ss)
                   (evaluate-begin e*
                                   (update* r n* a*)
                                   (update* ss a* v*)
                                  k ) ) )
                (wrong "Incorrect arity") ) ) )
          ss ) ) ) )
```

Lorsqu'une fonction, construite par la forme spéciale `lambda`, est appelée sur ses arguments, son comportement stipule qu'elle alloue dans la mémoire courante à l'instant d'appel autant de nouvelles adresses qu'elle a de variables à lier puis elle initialise ces adresses avec les valeurs associées et poursuit ses calculs. Si les adresses étaient allouées, par exemple, dans la mémoire courante à l'instant de création des fonctions, nous aurions un comportement (un peu à la Fortran) qui interdirait la récursion.

```
(define (evaluate-ftn-lambda n* e* r s k)
  (allocate (+ 1 (length n*)) s
    (lambda (a* ss)
      (k (create-function
          (car a*)
          (lambda (v* s k)
            (if (= (length n*) (length v*))
                (evaluate-begin e*
                                (update* r n* (cdr a*))
                                (update* s (cdr a*) v*)
                               k )
                (wrong "Incorrect arity") ) ) )
          ss ) ) ) )
```

En effet, tout appel récursif à la fonction utiliserait les mêmes adresses pour y stocker les valeurs variables, rendant ainsi les seules dernières valeurs de celles-ci accessibles. L'intérêt de cette variante est qu'il n'y a pas de création dynamique de fonctions en Fortran et que donc ces adresses peuvent être allouées lors de la compilation augmentant ainsi la vitesse de l'appel fonctionnel mais interdisant ce qui fait la force des langages fonctionnels : la récursion.

4.3.8 Mémoire

La mémoire est représentée par une fonction prenant des adresses et retournant des valeurs. Il doit aussi être possible d'y allouer de nouvelles adresses garanties libres, ce qui est l'apanage de la fonction `allocate`. Elle prend une mémoire et le nombre d'adresses qu'elle doit y réserver ; elle prend aussi un troisième argument, une fonction à qui elle donnera la liste des adresses qu'elle a sélectionnées dans la mémoire ainsi

que la nouvelle mémoire où ces adresses sont initialisées à l'indéfini. La fonction `allocate` condense en elle toute la gestion de la mémoire et notamment le glaneur de cellules s'il y en avait un. Fort heureusement, on peut en donner une caractérisation simple, si l'on suppose que la mémoire est infiniment grande, c'est-à-dire qu'il est toujours possible d'y allouer de nouveaux objets :

```
(define (allocate n s q)
  (if (> n 0)
      (let ((a (new-location s)))
        (allocate (- n 1)
                  (expand-store a s)
                  (lambda (a* ss)
                    (q (cons a a*) ss) ) ) )
      (q '() s) ) )
```

La forme `new-location` cherche une adresse libre dans une mémoire. C'est une vraie fonction en ce sens que la forme `(eqv? (new-location s) (new-location s))` retourne toujours Vrai. À toute mémoire est associée la plus haute adresse utilisée et c'est celle-ci qui sera utilisée lorsque l'on recherche une adresse libre par `new-location`. Pour marquer qu'une telle adresse est réservée, on étendra la mémoire au moyen de la fonction `expand-store`.

```
(define (expand-store high-location s)
  (update s 0 high-location) )
(define (new-location s)
  (+ 1 (s 0)) )
```

La mémoire initiale ne contient rien mais définit la première adresse libre. La mémoire est donc une fermeture répondant à un certain message lorsqu'il s'agit de déterminer une adresse libre ou à des messages codés comme des entiers (des adresses) lorsque l'on souhaite connaître le contenu d'une mémoire. Nous unifierons ces deux types de message en supposant que l'adresse 0 contient la dernière adresse utilisée.

```
(define s.init
  (expand-store 0 (lambda (a) (wrong "No such address" a))) )
```

4.3.9 Représentation des valeurs

Nous avons décidé de représenter toutes les valeurs que manipule l'interprète par des fonctions avec envoi de messages. Ces valeurs sont la liste vide, les booléens, les symboles, les nombres, les paires et les fonctions. Nous allons successivement examiner ces divers types de données.

Toutes les valeurs seront donc créées sur un squelette de fonction qui répondra au moins aux messages *(i)* `type` pour s'enquérir de son type ; *(ii)* `boolify` pour être converti en une valeur de vérité. D'autres messages existeront pour des types de données spécifiques. Le squelette qui servira d'ossature à toutes ces valeurs sera donc :

```
(lambda (msg)
  (case msg
    ((type)    ...)
    ((boolify) ...)
    ... ) )
```

La liste vide n'existe qu'en un unique exemplaire et, conformément au R^{4}RS, est un représentant légal de Vrai. Aussi Vrai sont les paires crées par `create-pair`.

```
(define the-empty-list
  (lambda (msg)
    (case msg
      ((type)    'null)
      ((boolify) (lambda (x y) x)) ) ) )
(define (create-pair a d)
  (lambda (msg)
    (case msg
      ((type)    'pair)
      ((boolify) (lambda (x y) x))
      ((set-car) (lambda (s v) (update s a v)))
      ((set-cdr) (lambda (s v) (update s d v)))
      ((car)     a)
      ((cdr)     d) ) ) )
```

Les deux valeurs booléennes sont créées par `create-boolean`

```
(define (create-boolean value)
  (let ((combinator (if value (lambda (x y) x) (lambda (x y) y))))
    (lambda (msg)
      (case msg
        ((type)    'boolean)
        ((boolify) combinator) ) ) ) )
```

Les symboles répondront au message spécifique `name` qui extraiera leur nom, représenté comme un symbole du Scheme de définition.

```
(define (create-symbol v)
  (lambda (msg)
    (case msg
      ((type)    'symbol)
      ((name)    v)
      ((boolify) (lambda (x y) x)) ) ) )
```

Les nombres auront `value` comme message spécifique.

```
(define (create-number v)
  (lambda (msg)
    (case msg
      ((type)    'number)
      ((value)   v)
      ((boolify) (lambda (x y) x)) ) ) )
```

Les fonctions répondront au message `behavior` qui extrait leur comportement et au message `tag` qui indique l'adresse à laquelle elles ont été allouées.

```
(define (create-function tag behavior)
  (lambda (msg)
    (case msg
      ((type)     'function)
      ((boolify)  (lambda (x y) x))
      ((tag)      tag)
      ((behavior) behavior) ) ) )
```

Le dernier cas qui nous occupe est celui des paires pointées. Une telle paire désigne deux valeurs qui sont toutes deux modifiables via les modificateurs physiques. Nous allons donc représenter une paire pointée par un couple de deux adresses : l'une contiendra le `car` de la paire tandis que l'autre contiendra son `cdr`. Cette organisation peut sembler curieuse car, usuellement, une paire pointée est représentée par une boîte de deux valeurs contiguës [11] : une seule adresse est donc nécessaire pour désigner la

11. De très sérieuses études [Cla79, CG77] ont montré qu'il valait mieux que le `cdr`, plutôt que le `car`,

paire. Ce codage ne rend pas hommage à la technique d'implantation qui consiste à mémoriser les `car` et `cdr` dans deux tableaux différents et au même index : une paire pointée est donc représentée par cet index. Sous ce codage, deux adresses différentes sont donc bien associées à toute paire pointée.

Pour simplifier l'allocation des listes, nous procurons les deux fonctions suivantes qui permettront de prendre une liste et une mémoire et d'y allouer cette liste. Comme l'allocation modifie la mémoire, un troisième argument (une continuation) sera finalement appelée avec la mémoire résultante et la liste allouée.

```
(define (allocate-list v* s q)
  (define (consify v* q)
    (if (pair? v*)
        (consify (cdr v*) (lambda (v ss)
                            (allocate-pair (car v*) v ss q) ))
        (q the-empty-list s) ) )
  (consify v* q) )
(define (allocate-pair a d s q)
  (allocate 2 s
   (lambda (a* ss)
     (q (create-pair (car a*) (cadr a*))
        (update (update ss (car a*) a) (cadr a*) d) ) ) ) )
```

4.3.10 Comparaison avec la programmation par objets

Le sequelette de fermeture choisi pour représenter les valeurs de cet interprète leur permet de répondre à plusieurs messages et, en ce sens, les apparente aux objets que nous avions employés au chapitre précédent. Il y a cependant quelques différences entre ces objets codés par des fermetures et ceux de MEROONET. Les objets actuels contiennent leurs méthodes en eux et il n'est pas possible de leur en ajouter de nouvelles. Les notions de classe et de sous-classe sont des concepts virtuels puisque non représentés par de quelconques valeurs. En contrepartie, les fonctions génériques permettent d'ajouter des comportements aux objets, de l'extérieur et sans même requérir leur coopération ; elles permettent non seulement les méthodes mais les multi-méthodes. Enfin la notion de sous-classe permet de partager la structure des objets sans répétition inutile de leurs caractéristiques communes. Ces diverses qualités font que l'on ne peut considérer les objets comme les *fermetures du pauvre* comme cela a parfois été suggéré dans les milieux schemiens. Les objets du chapitre précédent apportent plus de qualités que ceux de ce chapitre. On lira toutefois avec intérêt une défense et illustration de ces derniers en [AR88].

4.3.11 L'environnement initial

Comme d'habitude, nous introduirons deux syntaxes pour introduire dans l'environnement initial (et ici dans la mémoire initiale) les variables prédéfinies. À toute variable globale est donc associée une adresse qui contient sa valeur. La première syntaxe `definitial` allouera donc une nouvelle adresse et la renseignera avec la valeur idoine.

---

soit directement accessible, sans déplacement supplémentaire. Une autre raison est que les paires, servant à implanter des listes, héritent donc de la classe des objets chaînés qui ne définit que le seul champ `cdr`.

```
(define s.global s.init)
(define r.global r.init)
(define-syntax definitial
  (syntax-rules ()
   ((definitial name value)
    (allocate 1 s.global
     (lambda (a* ss)
       (set! r.global (update r.global 'name (car a*)))
       (set! s.global (update ss (car a*) value)) ) ) ) ) )
```

La syntaxe `defprimitive` sera construite au-dessus de `definitial` et définira une fonction dont l'arité sera vérifiée.

```
(define-syntax defprimitive
  (syntax-rules ()
   ((defprimitive name value arity)
    (definitial name
      (allocate 1 s.global
       (lambda (a* ss)
         (set! s.global (expand-store (car a*) ss))
         (create-function
          (car a*)
          (lambda (v* s k)
            (if (= arity (length v*))
                (value v* s k)
                (wrong "Incorrect arity" 'name) ) ) ) ) ) ) ) ) )
```

Comme à l'accoutumée, nous enrichirons l'environnement initial avec les variables booléennes `t`, `f` et la liste vide `nil` :

```
(definitial t (create-boolean #t))
(definitial f (create-boolean #f))
(definitial nil the-empty-list)
```

Donnons deux exemples de fonction prédéfinie, l'une arithmétique et l'autre prédicative. On y notera que les arguments sont extraits de leur représentation puis que le résultat final est réempaqueté comme il se doit.

```
(defprimitive <=
  (lambda (v* s k)
    (if (and (eq? ((car v*) 'type) 'number)
             (eq? ((cadr v*) 'type) 'number) )
        (k (create-boolean (<= ((car v*) 'value)
                               ((cadr v*) 'value) ))
           s )
        (wrong "<= require numbers") ) )
  2 )
(defprimitive *
  (lambda (v* s k)
    (if (and (eq? ((car v*) 'type) 'number)
             (eq? ((cadr v*) 'type) 'number) )
        (k (create-number (* ((car v*) 'value)
                             ((cadr v*) 'value) ))
           s )
        (wrong "* require numbers") ) )
  2 )
```

4.3.12 Paires pointées

Pour la première fois parmi tous les interprètes que nous avons exposés les paires pointées du Lisp défini ne seront pas représentées par des paires pointées du Scheme de définition.

Grâce à la fonction `allocate-pair`, la construction d'une paire pointée est assez simple.

```
(defprimitive cons
  (lambda (v* s k)
    (allocate-pair (car v*) (cadr v*) s k) )
  2 )
```

Lire les champs ou les modifier est tout aussi simple :

```
(defprimitive car
  (lambda (v* s k)
    (if (eq? ((car v*) 'type) 'pair)
        (k (s ((car v*) 'car)) s)
        (wrong "Not a pair" (car v*)) ) )
  1 )
(defprimitive set-cdr!
  (lambda (v* s k)
    (if (eq? ((car v*) 'type) 'pair)
        (let ((pair (car v*)))
          (k pair ((pair 'set-cdr) s (cadr v*))) )
        (wrong "Not a pair" (car v*)) ) )
  2 )
```

Le codage de toutes les valeurs manipulables permet que soit inspecté leur type. Comme tous les objets savent répondre au message `type`, il est simple d'écrire le prédicat `pair?`. Notez l'emploi de `create-boolean` pour convertir un booléen du Lisp de définition en un booléen du Scheme défini.

```
(defprimitive pair?
  (lambda (v* s k)
    (k (create-boolean (eq? ((car v*) 'type) 'pair)) s) )
  1 )
```

4.3.13 Comparaison

Un des buts de cet interprète était de pouvoir spécifier le prédicat de comparaison physique `eqv?`. Celui-ci compare donc deux objets physiquement ainsi que les nombres. Le prédicat compare donc les types des deux objets puis, s'il y a accord, compare spécifiquement les deux objets. Les symboles doivent avoir même nom ; les booléens ou les nombres doivent être les mêmes ; les paires pointées ou les fonctions doivent être alloués aux même(s) adresse(s).

```
(defprimitive eqv?
  (lambda (v* s k)
    (k (create-boolean
        (if (eq? ((car v*) 'type) ((cadr v*) 'type))
            (case ((car v*) 'type)
              ((null) #t)
              ((boolean)
               (((car v*) 'boolify)
                (((cadr v*) 'boolify) #t #f)
```

```
   (((cadr v*) 'boolify) #f #t) ) )
((symbol)
 (eq? ((car v*) 'name) ((cadr v*) 'name)) )
((number)
 (= ((car v*) 'value) ((cadr v*) 'value)) )
((pair)
 (and (= ((car v*) 'car) ((cadr v*) 'car))
      (= ((car v*) 'cdr) ((cadr v*) 'cdr)) ) )
((function)
 (= ((car v*) 'tag) ((cadr v*) 'tag)) )
(else #f) )
            #f ) )
       s ) )
  2 )
```

En fait la seule raison d'avoir conféré une adresse aux fonctions, outre que cela évoque (à juste titre) que les fonctions (ou plus précisément les fermetures) sont des structures de données allouées en mémoire, est que l'on peut comparer des fonctions par `eqv?`. Les fonctions sont projetées sur les adresses qui sont des entités aisément comparables puisque ce sont des entiers. Notez aussi que la comparaison des paires pointées ne nécessite pas d'inspecter leur contenu mais seulement les adresses qui leur sont associées. C'est donc une fonction à coût constant comparé à `equal?`. La définition de `eqv?` peut sembler générique puisque nous l'avons définie comme un rassemblement de méthodes sur des classes disjointes ; en fait, une représentation de données astucieuses permet de s'affranchir de ces tests de type et de comparer directement les adresses d'implantation des objets : on peut souvent la réduire à une simple instruction, excepté peut-être sur les grands nombres.

4.3.14 Lancer l'interprète

Donnons tout de suite l'interprète. On y notera comment la boucle fondamentale d'interaction se réinvoque dans la continuation qu'elle donne à `evaluate` tout en assurant de repartir de l'état mémoire obtenu lors de la dernière interaction.

```
(define (chapter4-interpreter)
  (define (toplevel s)
    (evaluate (read)
              r.global
              s
              (lambda (v ss)
                (display (transcode-back v ss))
                (toplevel ss) ) ) )
  (toplevel s.global) )
```

Le problème essentiel de cet interprète est que, pour la première fois dans ce livre, les données du Scheme interprété sont très différentes des données du Scheme sous-jacent. C'est particulièrement vrai pour les paires pointées, c'est-à-dire les listes, qui servaient de façon interne à, par exemple, rassembler les arguments soumis à une fonction. Ne plus avoir l'équivalence des paires pointées entre les deux niveaux de langage nous force à recourir à un décodage de la valeur finale obtenue grâce à la fonction `transcode-back`. Celle-ci parcourt donc la valeur dans un certain état mémoire et la transforme en une valeur du Scheme de définition que l'on pourra alors imprimer. On aurait tout aussi bien choisi d'imprimer directement la valeur sans passer par la

fonction `display` du Scheme sous-jacent.

```
(define (transcode-back v s)
  (case (v 'type)
    ((null)     '())
    ((boolean)  ((v 'boolify) #t #f))
    ((symbol)   (v 'name))
    ((string)   (v 'chars))
    ((number)   (v 'value))
    ((pair)     (cons (transcode-back (s (v 'car)) s)
                      (transcode-back (s (v 'cdr)) s) ) )
    ((function) v) ; why not ?
    (else       (wrong "Unknown type" (v 'type))) ) )
```

4.4 Entrées/sorties et mémoire

Nous n'avons pas parlé des fonctions d'entrées/sorties. Si l'on se limite à un seul flux d'entrée et un seul de sortie, c'est-à-dire aux seules fonctions `read` et `display`, alors le problème pourrait être précisément traité en ajoutant deux arguments supplémentaires à l'interprète modélisant ces deux flux. Le flux d'entrée contiendra tout ce qui sera lu, tandis que le flux de sortie sera formé de tout ce qui sera écrit dedans. On peut même considérer que le flux de sortie est la seule réponse observable que fournit l'interprète.

Le flux de sortie peut être représenté par une liste de couples (mémoire, valeur) ceux-là mêmes qui sont tour à tour fournis à la fonction `transcode-back`. En revanche, qu'est-ce que le flux d'entrée ? La question est délicate car les valeurs qui seront lues peuvent contenir des paires pointées qui peuvent elles-mêmes être modifiées physiquement. On peut, en effet, écrire : `(set-car! (read) 'foo)` et lire `(bar hux)`. Cela signifie donc que l'expression lue doit être installée dans la mémoire qui est courante au moment où `read` est appelée. La fonction `transcode` prendra donc une valeur du Scheme sous-jacent, une mémoire et une continuation et appellera cette dernière sur la valeur transcodée et le nouvel état mémoire dans lequel elle a été installée.

```
(define (transcode v s q)
  (cond
   ((null? c)     (q the-empty-list s))
   ((boolean? c) (q (create-boolean c) s))
   ((symbol? c)   (q (create-symbol c) s))
   ((string? c)   (q (create-string c) s))
   ((number? c)   (q (create-number c) s))
   ((pair? c)
    (transcode (car c)
               s
               (lambda (a ss)
                 (transcode (cdr c)
                            ss
                            (lambda (d sss)
                              (allocate-pair a d sss q) ) ) ) ) ) ) )
```

Nous ne poursuivrons pas plus avant cette variante qui nécessite donc deux arguments de plus dans toutes les fonctions que nous avons présentées dans ce chapitre pour désigner les flux d'entrée et de sortie.

4.5 Sémantique des citations

Peut-être s'est-on aperçu de l'omission caractérisée de la citation dans l'interprète courant ? La forme `quote` avait toujours été d'une simplicité biblique dans les chapitres précédents car il y avait identité entre les valeurs du Scheme défini et définissant. Ce n'est plus le cas maintenant aussi, ne peut-on plus écrire :

```
(define (evaluate-quote v r s k)                                    WRONG
  (k v s) )
```

On commettrait ce faisant une confusion entre un élément de programme `v` qui *décrit* la valeur à citer et la valeur de cette forme. En d'autres termes, `v` n'est pas la valeur qu'il faut retourner mais la description d'icelle. Ce n'était pas apparent dans les interprètes précédents du fait de la transparence du codage. On doit donc faire usage de `transcode` pour transformer la mention `v` en une valeur du Scheme défini :

```
(define (evaluate-quote c r s k)
  (transcode c s k) )
```

Cette définition est compositionnelle car le calcul de la valeur retournée par la citation ne dépend que des seuls arguments fournis à la fonction `evaluate-quote`. Dans le même temps elle introduit un hiatus par rapport à la programmation courante en Lisp ou Scheme. Considérez seulement l'expression `(quote (a . b))`, elle est de par la définition que nous avons donnée de la citation *exactement* équivalente à `(cons 'a 'b)` [12]. La citation d'un objet composite, comme ici la paire pointée `(a . b)`, induit la fabrication d'un tel objet dans la mémoire courante. La citation n'est donc qu'un raccourci exprimant commodément la construction d'une valeur. Cela exclut pourtant que l'expression suivante retourne Vrai :

```
(let ((foo (lambda () '(f o o))))
  (eq? (foo) (foo)) )
```

Dans cet exemple, la fonction locale `foo` synthétise de nouvelles valeurs à chaque appel dont les devenirs sont indépendants et qui sont donc différentes au sens de `eq?`. Si l'on souhaite que dans le précédent calcul, l'on obtienne toujours Vrai, on peut charcuter `evaluate-quote` pour toujours retourner la même valeur une fois que celle-ci a été choisie, c'est-à-dire en faire une mémo-fonction. Une raison pour ce souhait est, qu'en l'absence d'effets de bord, c'est-à-dire en l'absence d'`eq?`, il n'est nullement nécessaire et parfaitement dispendieux de construire encore et encore une même valeur. C'est bien sûr une autre histoire en présence d'effets de bord comme la suite le montrera.

```
(define *shared-memo-quotations* '())
(define evaluate-memo-quote
  (lambda (c r s k)
    (let ((couple (assoc c *shared-memo-quotations*)))
      (if (pair? couple)
          (k (cdr couple) s)
          (transcode c s (lambda (v ss)
                           (set! *shared-memo-quotations*
                                 (cons (cons c v)
                                       *shared-memo-quotations* ) )
                           (k v ss) )) ) ) ) )
```

12. En termes plus concis : `'(a . b)`≡`` `(,'a . ,'b) ``.

Mais la mémo-fonction précédente a réintroduit une affectation, partant un effet de bord, dont nous souhaitions réduire à néant le besoin. Une autre solution est de transformer le programme initial de façon à regrouper toutes les citations dans un endroit où elles ne seront évaluées qu'une unique fois : en tête du programme. Chaque fois qu'une citation apparaîtra, elle sera remplacée par une référence à une variable qui sera, au préalable, correctement initialisée. Ainsi transformera-t-on le programme :

```
(let ((foo (lambda () '(f o o))))
  (eq? (foo) (foo)) )
```

```
(define quote35 '(f o o))
(let ((foo (lambda () quote35)))
  (eq? (foo) (foo)) )
```

Le programme ainsi transformé recouvrera la sémantique usuelle de Lisp sans nécessité de toucher à la définition de `evaluate-quote`.

Si l'on veut faire disparaître totalement les citations d'objets composites, on pourra poursuivre la transformation esquissée plus haut et rendre apparentes toutes les allocations successives construisant les valeurs citées. On obtiendra alors :

```
(define quote36 (cons 'o '()))
(define quote37 (cons 'o quote36))
(define quote38 (cons 'f quote37))
(let ((foo (lambda () quote38)))
  (eq? (foo) (foo)) )
```

Mais ce n'est pas fini puisque l'on doit aussi assurer que les symboles cités de même nom correspondent à une même valeur. On continuera donc la transformation en :

```
(define symbol40 (string->symbol "f"))
(define symbol39 (string->symbol "o"))
(define quote36 (cons symbol39 '()))
(define quote37 (cons symbol39 quote36))
(define quote38 (cons symbol40 quote37))
(let ((foo (lambda () quote38)))
  (eq? (foo) (foo)) )
```

On s'arrêtera là, car on peut considérer les chaînes comme des objets primitifs : on peut les traduire tels quels en assembleur ou en C et il serait par trop dispendieux de les traduire en leurs constituants ultimes : en caractères !

En définitive, on peut accepter la définition compositionnelle de la citation puisque l'on peut retrouver, moyennant une transformation de programme, les habitudes usuelles. On peut toutefois s'interroger sur les errements en vigueur qui reposent sur le fait que les programmes sont souvent lus avec la fonction `read` et que donc l'expression qui figure dans une forme `quote` et qui spécifie la donnée immédiate à retourner est codée avec les mêmes conventions : mêmes paires pointées, mêmes symboles, etc. La paresse naturelle pousse alors, sous interprète, à employer cette valeur même et, donc, à la retourner chaque fois que de besoin. Ce faisant, on la partage avec tous les receveurs de la citation ce qui mène aux quiproquos qui ont longtemps fait les délices des lispiens de vieille souche. Considérez, par exemple, les expressions suivantes :

```
(define vowel<=
  (let ((vowels '(#\a #\e #\i #\o #\u)))
    (lambda (c1 c2)
      (memq c2 (memq c1 vowels)) ) ) )

(set-cdr! (vowel<= #\a #\e) '())
(vowel<= #\o #\u)   → ?
```

Sous interprète, il y a fort à parier que la valeur retournée ne sera pas `#t` et provoquera une erreur. Si en outre, l'on pouvait imprimer la définition de la fonction (ou si l'on avait fait de la variable close `vowels` une variable globale), on verrait que celle-ci s'est métamorphosée en :

```
(define vowel1<=
  (let ((vowels '(#\a #\e)))
    (lambda (c1 c2)
      (memq c2 (memq c1 vowels)) ) ) )
```

On a donc fait disparaître un morceau de programme ! Et la même technique aurait pu inversement accroître la valeur de la variable `vowels` au lieu de la diminuer. Cet effet « mémo-viscéral », qui peut se produire sous interprète n'a que peu de sens si on compile le programme précédent. En effet, si la variable globale `vowel<=` est immuable, un appel à `(vowel<= #\o #\u)`, où la fonction et tous ses arguments sont connus, peut être remplacé par `#t`. C'est une occurrence du phénomène connu sous le nom de propagation de constantes (ou *constant folding*) qu'a généralisé l'évaluation partielle [JGS93].

Le compilateur pourrait aussi décider de transformer la précédente définition après analyse des cas possibles pour `c1` en :

```
(define vowel2<=
  (lambda (c1 c2)
    (case c1
     ((#\a) (memq c2 '(#\a #\e #\i #\o #\u)))
     ((#\e) (memq c2 '(#\e #\i #\o #\u)))
     ((#\i) (memq c2 '(#\i #\o #\u)))
     ((#\o) (memq c2 '(#\o #\u)))
     ((#\u) (eq? c2 #\u))
     (else  #f) ) ) )
```

Cette nouvelle forme équivalente ne garantit pas que `(eq? (cdr (vowel<= #\a #\e)) (vowel<= #\e #\i))` puisqu'alors les résultats ne proviennent pas d'une même citation. Il n'est même pas, en Lisp, garanti que la valeur de `(eq? (cdr (vowel<= #\a#\e)) (vowel<= #\e#\i))` soit toujours fausse car les compilateurs s'octroient en général le droit de fusionner les citations (en jargon *constant coalescence*) afin de les faire tenir en une moindre place. Ils pourraient donc transformer la précédente définition en :

```
(define quote82 (cons #\u '()))
(define quote81 (cons #\o quote82))
(define quote80 (cons #\i quote81))
(define quote79 (cons #\e quote80))
(define quote78 (cons #\a quote79))
(define vowel3<=
  (lambda (c1 c2)
    (case c1
     ((#\a) (memq c2 quote78))
     ((#\e) (memq c2 quote79))
     ((#\i) (memq c2 quote80))
     ((#\o) (memq c2 quote81))
     ((#\u) (eq? c2 #\u))
     (else  #f) ) ) )
```

Ce genre de transformations a une incidence sur le partage des citations, qui peut être perçue par `eq?`. Une solution simple à tous ces problèmes est d'interdire que les

valeurs des citations soient modifiées : c'est le cas en Scheme et en COMMON LISP. Modifier la valeur d'une citation a des conséquences inconnues. On peut aussi faire en sorte que les valeurs des citations soient réellement immuables et, pour cela, définir la citation comme :

```
(define (evaluate-immutable-quote c r s k)
  (immutable-transcode c s k) )
(define (immutable-transcode c s q)
  (cond
   ((null? c)     (q the-empty-list s))
   ((pair? c)
    (immutable-transcode
     (car c) s (lambda (a ss)
                 (immutable-transcode
                  (cdr c) ss (lambda (d sss)
                               (allocate-immutable-pair
                                a d sss q ) ) ) ) ) )
   ((boolean? c) (q (create-boolean c) s))
   ((symbol? c)  (q (create-symbol c) s))
   ((string? c)  (q (create-string c) s))
   ((number? c)  (q (create-number c) s)) ) )
(define (allocate-immutable-pair a d s q)
  (allocate 2 s
   (lambda (a* ss)
     (q (create-immutable-pair (car a*) (cadr a*))
        (update (update ss (car a*) a) (cadr a*) d) ) ) ) )
(define (create-immutable-pair a d)
(lambda (msg)
  (case msg
    ((type)     'pair)
    ((boolify) (lambda (x y) x))
    ((set-car) (lambda (s v) (wrong "Immutable pair")))
    ((set-cdr) (lambda (s v) (wrong "Immutable pair")))
    ((car)      a)
    ((cdr)      d) ) ) )
```

Avec cette définition, toute tentative de corruption d'une citation est détectée. Ce traitement qui consiste à différencier les paires pointées muables et immuables peut être étendu aux chaînes de caractères et l'on pourra, comme en Mesa, distinguer les *strings* modifiables des *ropes* non modifiables.

En conclusion partielle, il vaut mieux ne pas tenter de modifier les valeurs retournées par les citations ; c'est ce que recommandent à la fois Scheme et COMMON LISP. Toutefois, on n'est pas au bout des problèmes que posent les citations. On a déjà mentionné la technique de fusion de citations qui pouvait entraîner que des citations égales construisaient des valeurs physiquement égales par `eq?`. Inversement il se pourrait que l'on ait des égalités physiques dans les citations que l'on voudrait bien sûr retrouver dans leur valeur. Il existe au moins deux façons pour spécifier ces égalités physiques : les macros et les traitements à la lecture.

En COMMON LISP, des macro-caractères permettent d'effectuer des traitements programmables lors de la lecture de certain caractères. Par exemple, `#.`*`expression`* permet de « lire » la valeur de *expression* : *expression* est lue puis évaluée [13] à la volée et sa valeur est réputée être ce que l'on a lu. C'est ainsi que l'on pourra écrire :

13. On ne dira pas comment, c'est-à-dire dans quel environnement ou avec quelle continuation.

```
(define bar (quote #.(let ((list '(0 1)))                    COMMON LISP
                       (set-cdr! (cdr list) list)
                       list )))
```

Cette expression initialise la variable `bar` avec une liste circulaire de 0 et de 1 alternés. La citation créée décrit une valeur qui est physiquement égale à son `cddr`. CLtL2 [Ste90] assure qu'il en sera également ainsi à l'exécution. On peut concevoir que cela rend la transformation des citations esquissées plus haut un peu plus complexe car on doit prendre en compte les cycles pour les reconstruire.

Le cycle précédent ne peut être ainsi construit en Scheme car on n'y dispose pas de macro-caractères. On a cependant les macros qui peuvent servir aux mêmes buts. Les macros qui suivent ne peuvent être écrites avec `define-syntax`, aussi userons-nous de `define-abbreviation`, une macro de définition de macros que nous analyserons au chapitre 9. **[☞ p. 295]**

```
(define-abbreviation (cycle n)
  (let ((c (iota 0 n)))
    (set-cdr! (last-pair c) c)
    `(quote ,c) ) )
(define bar (cycle 2))
```

On a ainsi construit à l'aide d'un nombre connu lors de la macroexpansion une liste circulaire que l'on a fait figurer dans une citation. Cette citation n'aurait pas pu être écrite à la main car il n'est pas possible de faire lire à Scheme des données cycliques. C'est possible en COMMON LISP à l'aide des macro-caractères `#n=` et `#n#` [14] :

```
(define bar #1=(0 1 . #1#))
```

La sémantique actuelle de Scheme semble exclure de tels cycles car ils correspondent à des descriptions de valeurs que l'on n'aurait pu écrire à la main. Ce sont les seules citations légales.

Nous reverrons de façon plus approfondie les nombreux problèmes liés aux macros au chapitre 9. **[☞ p. 295]**

4.6 Conclusions

Suite à ce chapitre, de multiples conseils peuvent être prodigués aux schemeurs et autres lispiens. L'exacte nature des variables globales est complexe (et l'est plus encore si l'on considère les problèmes de modules). Les diverses notions d'égalité sont subtiles et loin d'être partagées. Enfin la citation n'est pas aussi simple qu'il n'y paraît et l'on s'abstiendra de citations alambiquées.

4.7 Exercices

Exercice 4.1 : Donnez une définition fonctionnelle (sans effets de bord) de la fonction `min-max` ?

---

14. On remarquera dans l'exemple qu'il faut lire un objet qui fait partie de lui-même. Il faut donc que la fonction `read` soit quelque peu intelligente pour allouer une paire pointée puis la remplir. On retrouve là les habituels paradoxes concernant les objets auto-référents comme `#1=#1#`.

Exercice 4.2 : Les paires pointées que l'on avait simulées avec des fermetures utilisaient des symboles comme messages. Si l'on retire les opérations `set-car!` et `set-cdr!`, réécrivez la simulation en n'utilisant que des λ-formes.

Exercice 4.3 : Écrivez une définition de `eq?` comparant des paires pointées à l'aide de `set-car!` ou `set-cdr!`.

Exercice 4.4 : Définissez une nouvelle forme spéciale de syntaxe `(or` α β`)` qui retourne la valeur de α si celle-ci vaut Vrai, défait sinon tous les effets de bord effectués lors de l'évaluation de α et retourne ce que retourne β.

Exercice 4.5 : L'affectation, telle que définie dans ce chapitre, retournait la valeur venant d'être affectée. Redéfinissez l'affectation pour que sa valeur soit celle de la variable avant affectation.

Exercice 4.6 : Définissez pour l'interprète de ce chapitre les fonctions `apply` et la fonction `call/cc`.

Exercice 4.7 : Modifiez l'interprète de ce chapitre afin d'introduire les fonctions n-aires, c'est-à-dire dotées d'une variable pointée.

5

Sémantique dénotationnelle

Sémantique, le grand mot est enfin lâché ! À l'issue de quelques courts rappels sur le λ-calcul, ce chapitre dévoilera ce qu'est la sémantique dénotationnelle et certains de ses aspects. La nouvelle définition de Lisp, dénotationnelle cette fois, à laquelle nous aboutirons diffère peu de celle du précédent interprète mais associe à tout programme son sens sous la forme d'un objet mathématiquement respectable : un terme du λ-calcul.

Qu'est-ce qu'un programme ? Un *programme* est la description d'un procédé calculatoire visant à obtenir un certain résultat ou effet. On confond souvent un programme avec ses incarnations exécutables sur telle ou telle machine aussi bien qu'on ne le distingue pas toujours du fichier contenant la forme physique que revêt sa définition. Un programme est exprimé dans un *langage* ; la définition d'un langage permet de donner un sens à tous les programmes que l'on peut, grâce à lui, exprimer. Le sens d'un programme ne se réduit pas à la valeur qu'il peut produire lors d'une exécution d'autant que cette exécution peut requérir des lectures ou des interactions avec le monde extérieur dont on ne peut savoir à l'avance ce qu'elles seront. Le sens d'un programme est une propriété beaucoup plus fondamentale qui fonde son essence.

Le sens d'un programme doit être un objet mathématiquement manipulable. Une transformation de programme comme, par exemple, la transformation par boîtes sera jugée **[☞ p. 108]** sûre dès lors qu'elle sera fondée par une démonstration montrant qu'elle préserve le sens des programmes auxquels elle est appliquée. Le *sens* d'un programme doit être un objet mathématiquement respectable de façon à ne pas être ambigu et à autoriser l'emploi d'outils que des siècles, voire des millénaires, de culture mathématique ont forgés. Accorder un sens à un programme, c'est lui associer un objet d'un autre espace plus évidemment mathématisable. Il serait par exemple judicieux, voire astucieux, que le sens du programme définissant `factorial` soit justement la factorielle usuelle des **[☞ p. 51]** mathématiques, car alors on saurait d'une part, que cette définition calcule bien la factorielle et d'autre part, si des variantes comme `meta-fact` lui sont bien équivalentes. **[☞ p. 60]**

Associer un sens à un programme revient à lui chercher un équivalent mathématique mais cette recherche nécessite de connaître les propriétés du langage de programmation employé. Le problème est donc d'associer à un langage de programmation, une méthode conférant à tout programme écrit dans ce langage son sens : on parle alors de

sémantique du langage de programmation. La sémantique d'un langage de programmation a plus d'un intérêt : elle permet de comprendre le langage pour l'implanter, pour y prouver des transformations, pour comparer ses caractéristiques à d'autres langages, etc. Nombreux sont ceux qui ont de l'intérêt pour elle. Mais la sémantique n'est pas unique et diverses méthodes existent.

La plus ancienne méthode de définition d'un langage est certainement l'existence d'une *implantation de référence*. Lorsque l'on a une question sur le langage comme, par exemple, ce que vaut ou effectue tel ou tel programme, on le soumet à l'implantation de référence et l'on en accepte l'oracle. L'ennui est que l'on ne peut guère bâtir de théorie sur cette implantation de référence surtout si on la considère comme une boîte noire. Si, en revanche, on s'intéresse à l'implantation et donc cherche à ouvrir cette fameuse boîte, on se retrouve face à un programme écrit dans un certain langage, l'ignorance ou l'ambiguité de ce nouveau langage reposant à nouveau tout le problème du sens.

Une autre voie est celle consistant à utiliser une *machine virtuelle* ; on ne s'affranchit pas totalement du problème précédent mais on le scinde en deux parties distinctes. D'une part, le langage est décrit en terme d'une machine virtuelle dotée d'une certaine architecture et d'un certain jeu d'instructions. Toutes les instructions du langage sont définies par un certain assemblage d'instructions de la machine virtuelle : si l'on comprend comment marche la machine virtuelle alors on comprend le langage. D'autre part, la machine virtuelle est elle-même décrite dans un formalisme ad hoc permettant d'envisager de l'implanter sur n'importe quel ordinateur raisonnable. De nombreux langages ont été ainsi définis, de PL1 (avec VDM) à Le-Lisp (et LLM3 [Cha80]), PSL [GBM82] ou encore PVM [FM90] [1]. La principale difficulté réside en l'élaboration d'une machine virtuelle qui doit être à la fois claire d'intention, simple à utiliser et triviale à implanter. Pour ce faire, on peut choisir des machines à pile(s), à registre(s), pilotées par des arbres ou des graphes, etc. Ce scénario correspond à une version idéalisée de la programmation en langage d'assemblage où le concepteur aurait la liberté de définir sa propre machine. Cette technique permet de comparer des programmes par confrontation de leurs traductions en assembleur virtuel ou par examen de leur trace d'exécution. Ce recours à l'exécution d'une machine, fut-elle virtuelle, justifie le nom de cette technique que l'on qualifie de *sémantique opérationnelle*.

La précédente voie avait le défaut de requérir une machine non standard ou plus exactement l'existence d'un formalisme calculatoire ; l'emploi d'une théorie connue de tous permettrait de faire l'économie de cette machine. Un programme est avant tout une fonction transformant ses entrées en ses sorties. « Exécuter » cette fonction ne requiert pas une machinerie compliquée : seulement quelques siècles de culture mathématique suffisent pour l'« appliquer ». L'idée est donc de transformer un programme en une fonction (d'un espace fonctionnel approprié), ce que l'on nomme sa *dénotation*. Le problème qui reste est de s'entendre sur l'espace des dénotations. Un bon choix est le λ-calcul qui possède des bases si simples que tout le monde ne peut qu'être d'accord sur la façon dont il fonctionne. La sémantique d'un langage est donc le processus par lequel on associe à un programme sa dénotation ; c'est encore une fonction. La dénotation d'un programme est le λ-terme qui représente son sens.

1. Sans oublier maintenant Java (et JVM [LY97]) ou C# (et CLR [Ric06]).

Muni de ces dénotations, la théorie du λ-calcul standard permet de décider ce qu'un programme calcule, si deux programmes sont égaux, etc. Il existe bien sûr de nombreuses variantes suivant la nature des dénotations, le type de λ-calcul choisi, la façon de construire la fonction sémantique qui associe programmes et dénotations mais la structure que nous venons d'expliciter est ce que l'on nomme conventionnellement la *sémantique dénotationnelle*.

Mentionnons encore une autre voie provenant des travaux sur la preuve de programmes d'après Floyd et Hoare et que l'on nomme la *sémantique axiomatique*. L'idée consiste à définir chaque forme élémentaire du langage par une formule logique : $\{P\}forme\{Q\}$. Cette formule indique que si P est vraie avant l'exécution de *forme* et si l'exécution de *forme* se termine, alors Q sera vraie. On peut ainsi spécifier toutes les formes élémentaires d'un langage et le définir par son axiomatique. On voit bien l'impact positif qu'une telle description peut avoir sur les techniques de preuve dans ce langage. En revanche, c'est un procédé non constructif qui ne révèle rien sur l'implantation d'un langage ainsi défini, pas même son existence !

Nous n'avons cependant toujours pas épuisé l'arsenal des sémantiques aujourd'hui existantes. Mentionnons la *sémantique naturelle* [Kah87] qui favorise la notion de relation plutôt que de fonctions dans un cadre dérivé de la sémantique dénotationnelle ou encore la *sémantique algébrique* [FF89] qui raisonne en terme d'équivalences de programmes au moyen de règles de réécriture.

5.1 Rappels sur le λ-calcul

La sémantique dénotationnelle consiste donc à définir, pour un langage, une fonction dite de *valuation* associant à chaque programme légal de ce langage un terme dans l'espace des dénotations. Pour ce dernier, nous choisirons le λ-calcul [2] en raison de sa simplicité structurelle et de sa faible distance par rapport à Scheme, dont il a été dit qu'il était un interprète efficace du λ-calcul [SS75, Wan84].

Nous ne rappellerons que très brièvement ce qu'est le λ-calcul [3]. Sa syntaxe est très simple : les termes du λ-calcul sont des variables, des abstractions ou des applications (ou combinaisons puisqu'en termes lispiens, les fonctions du λ-calcul sont monadiques). En notant **Variable** l'ensemble des variables possiblement utilisables, l'ensemble des termes du λ-calcul, Λ, se définit récursivement comme :

$$\begin{array}{lll} \textit{variable} : & \forall x \in \textbf{Variable}, & x \in \Lambda \\ \textit{abstraction} : & \forall x \in \textbf{Variable}, \forall M \in \Lambda, & \lambda x.M \in \Lambda \\ \textit{combinaison} & \forall M, N \in \Lambda, & (MN) \in \Lambda \end{array}$$

Comme toujours, vue de Lisp, la syntaxe n'a pas d'importance et l'on pourrait aussi bien écrire les termes du λ-calcul sous une forme parenthésée [4]. L'ensemble des termes du λ-calcul est donc syntaxiquement un sous-ensemble des termes de Scheme réduit à la seule forme spéciale `lambda` :

```
x          (lambda (x) M)          (M N)
```

2. C'est la traduction consacrée de *λ-calculus* et cela est loin d'être un calcul λ !
3. De très bonnes introductions existent en [Sto77, Gor88], la bible demeurant [Bar84].
4. C'est d'ailleurs ce qu'a fait McCarthy lui-même, en 1960, au MIT.

Le λ-calcul permet de décrire des fonctions ; une règle de calcul existe qui permet de les appliquer : la *β-réduction*. Celle-ci stipule simplement que lorsque l'on applique une fonction de corps M et de variable x à un terme N, on obtient un nouveau terme qui est le corps M de la fonction où la variable x a été remplacée par le terme N, ce que l'on note comme $M[x \rightarrow N]$. C'est le modèle usuel de substitution que l'on a toujours utilisé en mathématiques sans jamais le formaliser ; c'est aussi la règle dont use le sous-ensemble précédemment évoqué de Scheme (sans effet de bord et avec `lambda` pour seule forme spéciale) lorsque l'on applique une fonction à son argument.

$$\beta\text{-réduction :} \quad (\lambda x.M\ \ N) \xrightarrow{\beta} M[x \rightarrow N]$$

La substitution $M[x \rightarrow N]$ est une opération assez subtile qui se définit en prenant garde aux éventuelles captures de variables. Celles-ci ne peuvent se produire que lorsque l'on opère des substitutions dans le corps des abstractions : on retrouve là le problème lié aux variables libres dans le corps des fonctions. Il ne faut pas que les variables libres de N soient capturées par des variables ambiantes présentes dans M comme c'était le cas en liaison dynamique : le λ-calcul est, en termes lispiens, en liaison lexicale. Dans la définition de la substitution qui suit, quelques parenthèses superflues ont été ajoutées afin de mieux isoler les termes faisant l'objet d'une substitution :

$x[x \rightarrow N] = N$
$y[x \rightarrow N] = y$ avec $x \neq y$
$(\lambda x.M)[x \rightarrow N] = \lambda x.M$
$(\lambda y.M)[x \rightarrow N] = \lambda z.(M[y \rightarrow z][x \rightarrow N])$ avec $x \neq y$ et z non libre dans $(M\ N)$
$(M_1\ M_2)[x \rightarrow N] = (M_1[x \rightarrow N]\ M_2[x \rightarrow N])$

La β-réduction permet de supprimer un *redex*, c'est-à-dire une expression réductible (pour *reducible expression*) ou, plus précisément encore, une application dont le premier terme (le terme en position fonctionnelle) est une abstraction. Lorsqu'un terme ne contient plus aucun redex, on dit qu'il est en *forme normale*. Les termes du λ-calcul n'ont pas nécessairement de forme normale mais lorsqu'ils en ont une, elle est unique grâce à la propriété dite du diamant et connue sous le nom de Church-Rosser. Lorsqu'un terme a une forme normale, il existe une suite finie de β-réductions convertissant le terme original vers le terme en forme normale. Une *règle d'évaluation* est un procédé indiquant quel redex, s'il y en a plus d'un, doit être β-réduit. Il se trouve malheureusement qu'il y a de bonnes règles d'évaluation et de mauvaises. Une bonne règle est de toujours évaluer le redex dont la parenthèse ouvrante est la plus à gauche. Cette règle ne minimise pas les calculs mais termine toujours sur la forme normale si celle-ci existe, c'est pourquoi on la nomme *ordre normal*. Une *mauvaise* règle est celle de Scheme, dite d'*appel par valeur*, où l'on n'applique une fonction qu'après avoir évalué ses arguments.

Donnons quelques exemples. Un exemple de terme sans forme normale est :

$$(\omega\ \omega) \quad \text{avec } \omega = \lambda x.(x\ x) \qquad \text{puisque} \quad (\omega\ \omega) \xrightarrow{\beta} (\omega\ \omega) \xrightarrow{\beta} (\omega\ \omega) \xrightarrow{\beta} \ldots$$

Vu de Scheme, ce programme boucle et, par immédiate voie de conséquence, ne peut conduire à un quelconque terme, à plus forte raison en forme normale ! Un exemple de terme qui a une forme normale mais tel que la règle d'évaluation de Scheme ne la trouve pas est :

$$((\lambda x.\lambda y.y\ \ (\omega\omega))\ z) \quad \xrightarrow{\beta} (\lambda y.y\ \ z) \quad \xrightarrow{\beta} z$$

En Scheme, le premier argument $(\omega\ \omega)$ serait calculé et bouclerait sans fin : on ne parviendra donc jamais à la forme normale !

Inversement et toujours aussi malheureusement, le calcul d'un terme peut s'achever en Scheme sans avoir pour autant de forme normale. La raison est que l'on ne réduit pas, en Scheme, sous les `lambda` même s'il s'y trouve encore au moins un redex. Par exemple :

$$\lambda x.(\omega\ \omega)$$

Pourquoi conserver la règle d'évaluation de Scheme si ce n'est pas une bonne règle ? La raison est que le calcul en appel par valeur est beaucoup plus efficace que la bonne règle dite d'*appel par nom* même si celle-ci peut être améliorée en *appel par nécessité*. L'autre raison est que chaque fois que Scheme aboutit à une valeur en forme normale, et vu l'unicité de celle-ci, c'est la valeur même à laquelle aurait abouti la bonne règle. Il y a donc souvent et fortunément coïncidence entre les valeurs produites par les deux méthodes.

Une convention pratique du λ-calcul est d'autoriser syntaxiquement des fonctions à plusieurs variables en posant que $\lambda xy.M$ n'est autre que $\lambda x.\lambda y.M$. Réciproquement, pour faciliter l'application de ces fonctions, on posera que $(M N_1 N_2)$ est en fait $((M N_1) N_2)$. Le pénultième exemple devient alors plus léger d'écriture et de compréhension et devient tout simplement :

$$(\lambda xy.y\ \ (\omega\ \omega)\ \ z)$$

On voit bien ici l'inintérêt de calculer la valeur qui sera liée à la variable x qui n'apparaît pas dans la réponse finale.

Bien d'autres choses seraient à dire à la gloire du λ-calcul mais pour ceci nous renvoyons aux bons ouvrages [Bar84, Gor88, Dil88]. On peut entre autres enrichir le λ-calcul avec des termes supplémentaires comme les nombres entiers (on parle alors de λ-*calcul appliqué*) ainsi qu'avec de nouvelles règles entre ces termes (notamment que $2 + 2 = 4$ ce que l'on nomme une δ-*règle*). Ce n'est toutefois pas logiquement nécessaire puisque les entiers et les booléens peuvent être encodés en des λ-termes ainsi que les opérations arithmétiques ou logiques fondamentales, c'est aussi vrai de la structure de liste avec `cons`, `car` et `cdr`, voir par exemple [Gor88]. **[☞ Ex. 4.2]**

En conclusion, le λ-calcul est une théorie raffinée qui procure à la fois un cadre calculatoire simple mais puissant (la β-réduction a la puissance de la machine de Turing mais pas sa complexité) et une notion d'égalité (deux termes sont égaux s'ils se réduisent en un même troisième) permettant de comparer des termes. Le λ-calcul est donc un excellent espace de dénotations.

5.2 Sémantique de Scheme

Nous tenons donc, en la personne du λ-calcul, un remarquable candidat pour représenter des dénotations. Déjà le précédent interprète était écrit en un Scheme sans effet de bord. Il décrivait une fonction d'évaluation : `evaluate`, de signature suivante :

`evaluate` : **Programme** $\times$ **Environnement** $\times$ **Continuation** $\times$ **Mémoire** $\rightarrow$ **Valeur**

Un léger effort nous amène à imaginer que l'on peut modifier cette signature par curryfication sur le premier argument, le programme, et ainsi obtenir :

$$\textbf{Programme} \to (\textbf{Environnement} \times \textbf{Continuation} \times \textbf{Mémoire} \to \textbf{Valeur})$$

On peut donc par curryfication quasiment instantanée associer à tout fragment de programme une fonction attendant qu'on lui fournisse seulement un environnement, une continuation et une mémoire pour indiquer alors quelle valeur elle retournerait. Nous tenons donc là notre valuation, c'est-à-dire la sémantique de Scheme, ainsi que l'exacte nature de nos dénotations.

$$\begin{aligned} \textit{valuation} &: \textbf{Programme} \to \textbf{Dénotation} \\ \textbf{Dénotation} &: \textbf{Environnement} \times \textbf{Continuation} \times \textbf{Mémoire} \to \textbf{Valeur} \end{aligned}$$

Mais les sémanticiens dénotationalistes n'aiment pas les parenthèses et ont des goûts très précis sur le style que doivent arborer les dénotations. En premier, ils usent et abusent de lettres grecques, de notations courtes, elliptiques voire cryptiques. La raison est qu'avec beaucoup d'habitude, on arrive à faire tenir la sémantique d'un langage simple sur une unique page où on peut l'appréhender dans sa plénitude. Ce gain [5] qui permet d'avoir tout sous les yeux est incompatible avec des identificateurs longs ou de mots-clés verbeux. Le choix des lettres grecques vient aussi de ce que la dénotation est écrite dans un langage qu'il importe de ne point confondre avec le langage que l'on décrit. Comme les langages que l'on décrit sont informatiques et n'emploient bien souvent qu'un jeu restreint de caractères dits ASCII, le grec tranche agréablement et limite les confusions. Enfin, les dénotations sont pour plus de sécurité typées, le nom des variables servant à indiquer leur type. De tenaces conventions se sont créées dont nous suivons la trace. Les fonctions sont classiquement et suivant l'usage établi en mathématiques notées φ (comme fonction). Les autres ont souvent l'initiale grecque de leur nom anglo-saxon, par exemple, κ pour continuation α pour adresse, ν pour nom, π pour programme, σ comme mémoire (pour *store*) [6].

π	**Programme**	ρ	**Environnement**
ν	**Identificateur**	α	**Adresse**
σ	**Mémoire**	ε	**Valeur**
κ	**Continuation**	φ	**Fonction**

Chaque mot en gras dans le tableau ci-dessus nomme un *domaine* représentant les objets manipulés par les dénotations. On y retrouve tous les objets que les chapitres précédents avaient introduits. La table suivante définit les domaines.

$$\begin{aligned} \textbf{Environnement} &= \textbf{Identificateur} \to \textbf{Adresse} \\ \textbf{Mémoire} &= \textbf{Adresse} \to \textbf{Valeur} \\ \textbf{Valeur} &= \textbf{Fonction} + \textbf{Booléen} + \textbf{Entier} + \textbf{Paire} + \ldots \\ \textbf{Continuation} &= \textbf{Valeur} \times \textbf{Mémoire} \to \textbf{Valeur} \\ \textbf{Fonction} &= \textbf{Valeur}^* \times \textbf{Continuation} \times \textbf{Mémoire} \to \textbf{Valeur} \end{aligned}$$

Comme d'habitude, l'étoile indique une répétition : le domaine **Valeur*** est le domaine des séquences de **Valeur**. Le signe $\times$ désigne le produit cartésien. Enfin le signe $+$ représente la somme disjointe de domaines : un élément de **Valeur** est un élément de **Fonction** ou un élément de **Booléen** ou un **Entier**, etc. Tous les types

5. C'est aussi un avantage si l'on songe que la longueur moyenne d'une communication scientifique est de l'ordre d'une dizaine de pages.

6. J'ignore par contre pourquoi les environnements sont notés ρ.

de valeurs sont représentés par un domaine figurant dans cette somme. La somme disjointe de domaines a la propriété qu'il est possible lorsque l'on tient un élément de **Valeur** de connaître le domaine précis d'où il provient. L'impératif d'être bien typé nous impose donc de déclarer où et comment s'effectuent ces changements de domaines. L'expression $\epsilon\ in$**Valeur** signifie que l'on injecte le terme ϵ dans le domaine **Valeur** tandis que $\varepsilon\ |_{\textbf{Entier}}$ projettera la valeur ε dans le domaine d'où elle provient (en supposant bien sûr que ε soit un entier).

On pourra remarquer que ces domaines sont définis récursivement ce qui est mathématiquement délicat et leur a d'ailleurs valu le nom de domaine plutôt qu'ensemble. Sans entrer dans les détails, disons que le λ-calcul a été inventé par Alonzo Church dans les folles années trente mais que cette construction n'a reçu un modèle mathématique qu'à la suite des travaux de Dana Scott vers 1970. On savait déjà avant que le λ-calcul était utile toutefois, se voir doter d'un modèle mathématique c'était enfin exister pour de bon. Depuis, plusieurs modèles ont été donné avec des propriétés extensionnelles différentes : D^∞ ou $\mathcal{P}_\omega$ [Sco76, Sto77].

L'*extensionnalité* est la propriété $(\forall x, f(x) = g(x)) \Rightarrow (f = g)$ et est liée à la η-*conversion* que l'on prend souvent comme règle supplémentaire du λ-calcul.

$$\eta\text{-conversion :} \quad \lambda x.(M\ x) \xrightarrow{\eta} M \qquad \text{avec } x \text{ non libre dans } M$$

Il se trouve bizarrement que $\mathcal{P}_\omega$ est extensionnel (deux fonctions qui calculent la même chose en tout point sont égales) tandis que D^∞ ne l'est pas. La nature est-elle extensionnelle ?

Les travaux de Scott ont montré que tout système de domaines récursivement définis à l'aide des seules $\rightarrow$, $\times$, $+$ ou * avait une solution unique ; les domaines existent donc bien.

Un principe important de la sémantique dénotationnelle est le principe de compositionalité : le sens d'un fragment de programme ne peut dépendre que du sens de ses constituants. Ce principe fonde les preuves inductives que l'on peut effectuer dans le cadre d'un langage ainsi défini. C'est aussi un bon principe, du point de vue du langage lui-même, que l'on puisse comprendre ce que signifie un fragment de programme indépendamment de ce qui pourrait l'environner.

La fonction de valuation associée à un langage est la plupart du temps notée $\mathcal{E}$ et, pour renforcer la distinction entre programme et sémantique, les fragments de programmes sont enclos à l'intérieur des crochets dits sémantiques $[\![$ et $]\!]$. Enfin, la valuation est présentée cas par cas, c'est-à-dire forme élémentaire par forme élémentaire.

5.2.1 Référence à une variable

La plus simple des dénotations est celle qui concerne la valeur d'une variable.

$$\mathcal{E}[\![\nu]\!] = \lambda\rho\kappa\sigma.(\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

La dénotation d'une référence à une variable, ici ν, est un λ-terme qui si on lui donne un environnement ρ, une continuation κ et une mémoire σ, déterminera l'adresse associée à la variable dans l'environnement, $(\rho\ \nu)$, soumettra cette adresse à la mémoire afin d'obtenir la valeur associée, $(\sigma\ (\rho\ \nu))$ retournera enfin cette valeur à la continuation accompagnée de la mémoire non modifiée car lire une mémoire est non-destructif.

Nous avons fait usage de la convention syntaxique permettant d'écrire des fonctions à plusieurs arguments. Si on le souhaite cependant, on peut écrire plus exactement mais plus illisiblement :

$$\mathcal{E}[\![\nu]\!] = \lambda\rho.\lambda\kappa.\lambda\sigma.(\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

Nous ne ferons point étalage de formalisme et nous abandonnerons totalement ce pénible et abstrus style d'écriture. Au passage, et toujours pour simplifier, nous adopterons la convention d'écriture suivante, semblable à celle du `define` de Scheme qui permet d'enlever quelques niveaux de λ de plus :

$$\mathcal{E}[\![\nu]\!]\rho\kappa\sigma = (\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

Une erreur est possible dans la dénotation des références qui correspond au cas où la variable n'apparaît pas dans l'environnement. Cette situation erronée est traitée par l'environnement initial, ρ_0, qui est :

$$(\rho_0\ \nu) = \mathit{wrong}\ \text{“}\texttt{No such variable}\text{”}$$

Lorsqu'une variable n'est pas définie dans l'environnement, la rechercher invoquera la fonction *wrong* qui, à son tour, produit une valeur généralement notée $\perp$. Cette valeur a la particularité d'être absorbante, c'est-à-dire que $\forall f, f\perp = \perp$. Ainsi, lorsqu'une situation est erronée, le calcul total se résout en $\perp$ ce qui est le sûr indice qu'il s'est produit une erreur. En fait ce n'est pas tant $\perp$ qui a cette particularité d'absorption, que les fonctions que nous manipulons et qui sont dites *strictes* (f est stricte ssi $f\perp = \perp$) ; cela permet dans une certaine mesure de ne pas s'encombrer du traitement des erreurs et de ne laisser subsister que la partie la plus signifiante de la dénotation.

5.2.2 Séquence

La dénotation d'une séquence utilisera une valuation auxiliaire qui sera l'équivalent de la fontion `eprogn` vue dans les interprètes précédents et que nous noterons [7] $\mathcal{E}^+$. Cette valuation convertira une suite non vide de formes que l'on notera π^+, conformément à une même tradition, en une dénotation évaluant tous les termes dans l'ordre gauche-droite et retournant la valeur de la dernière des formes. Sa raison d'être est que deux cas définissent la séquence suivant qu'elle contient une seule forme ou plus d'une. Rappelons qu'en Scheme, le sens de la séquence vide `(begin)` n'est pas précisé, ce qui se traduit par son absence des cas ci-dessous considérés :

$$\mathcal{E}[\![\texttt{(begin}\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi\ \pi^+]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma_1)\ \sigma)$$

Lorsque la séquence ne contient qu'un seul et unique terme, celle-ci est équivalente à ce terme. Ce que l'on peut noter plus directement par :

$$\mathcal{E}^+[\![\pi]\!] = \mathcal{E}[\![\pi]\!]$$

---

7. Nous avons choisi ce nom car il y a nécessairement au moins un terme dans la séquence de formes qu'elle dénote.

Ce que nous venons d'écrire est, en termes λ-calculesques, une η-simplification. Nous éviterons toutes ces fioritures qui rendent le code (ou les dénotations) plus compliqué à lire en masquant complètement les arités naturelles des fonctions et qui souvent n'apparaissent que pour faire plus intelligent.

Lorsque la séquence comprend plus d'un terme, la dénotation calcule la valeur du premier de ces termes et l'oublie pour calculer les autres termes. C'est la définition usuelle que nous voyons resurgir une fois de plus avec tous les détails auxquels vous êtes maintenant habitués. On voit notamment bien, sur une seule ligne, que l'état mémoire résultant de l'évaluation du premier terme est celui qui sert pour l'évaluation des termes suivants. La mise en séquence des évaluations passe par l'emploi de continuations qui se passent tour à tour la mémoire servant aux évaluations.

5.2.3 Alternative

La dénotation de l'alternative est classique et ne pose guère de difficultés si l'on sait comment l'on plonge les booléens au sein du λ-calcul. Les booléens peuvent aisément être simulés en λ-calcul. On définira les valeurs de vérité Vrai et Faux comme les combinateurs :

$$\mathbf{T} = \lambda xy.x \qquad \text{et} \qquad \mathbf{F} = \lambda xy.y$$

L'intuition derrière ces définitions est que tous deux prennent deux valeurs en arguments et rendent la première ou la seconde. Cela rappelle le connecteur logique **If** qui répond aux équations :

$$\mathbf{If}(true, p, q) = p \qquad \text{et} \qquad \mathbf{If}(false, p, q) = q$$

Attention : ce connecteur logique n'a rien à voir avec la forme spéciale `if` de Scheme. Ici on ne parle point d'ordre d'évaluation mais seulement du fait que **If** est une *fonction* qui pourrait être définie par une table de vérité mais qui s'écrit plus simplement comme :

$$\mathbf{If}(c, p, q) = (\neg c \vee p) \wedge (c \vee q)$$

Une façon simple de simuler **If** en λ-calcul est, compte tenu du codage choisi pour les booléens, d'écrire en notant **IF** le nouveau combinateur :

$$\mathbf{IF}\ c\ p\ q = (c\ p\ q)$$

Tout comme en logique, cette définition ne dit rien sur l'ordre de calcul mais énonce seulement une relation entre trois valeurs. Vue comme une fonction, **If** rend son second argument si le premier vaut Vrai, son troisième sinon. En termes plus lispiens, cette fonction, que nous nommerons `ef` comme en [FW84], s'approxime en :

```
(define (ef v v1 v2)
  (v v1 v2) )
```

Afin de rendre plus lisible la notion d'alternative, on adoptera pour l'instant la syntaxe suivante $\epsilon_0 \rightarrow \epsilon_1[]\epsilon_2$ (extraite de [Sch86]) ; le R^{4}RS, quant à lui, use de $\varepsilon_1 \rightarrow \varepsilon_2, \varepsilon_3$.

Nous pouvons donc énoncer la dénotation de l'alternative comme :

$$\mathcal{E}[\![\texttt{(if } \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.\ (\mathit{boolify}\ \varepsilon)$$
$$\rightarrow\ (\mathcal{E}[\![\pi_1]\!]\ \rho\ \kappa\ \sigma_1)$$
$$[]\ (\mathcal{E}[\![\pi_2]\!]\ \rho\ \kappa\ \sigma_1)\ \ \sigma)$$

La fonction *boolify* convertit une valeur quelconque en un booléen puisque toute valeur Scheme est implicitement une valeur de vérité. L'alternative commence donc par évaluer sa condition avec une nouvelle continuation qui décidera quelle branche de l'alternative choisir suivant la valeur de la condition. Attention : l'alternative du λ-calcul semble apparemment se comporter comme le bon vieux `if-then-else` mais rien, absolument rien n'est impliqué sur l'ordre d'évaluation. On peut très bien au sein du λ-calcul décider d'évaluer en parallèle la condition et les deux branches de l'alternative afin de choisir entre elles lorsque tous les calculs sont achevés.

Les conséquences sont importantes car on ne peut voir le λ-calcul, en quelque sorte le langage dans lequel on décrit un nouvel interprète, comme l'on avait auparavant vu Scheme : il n'y a pas de notion d'ordre d'évaluation. Pour verbaliser la dénotation définissant l'alternative, on pourrait dire que la valeur d'une alternative est, des deux valeurs possibles, celle qui est indiquée par la valeur de la condition.

Remarquez qu'il n'y a pas de concurrence déloyale ou d'effet de bord occulte à évaluer en parallèle les deux branches de l'alternative puisque chacune a son propre jeu de paramètres définissant le calcul en cours et que nous sommes dans un langage pur dénué d'effet de bord. Il n'y a, par exemple, aucun problème avec l'expression suivante, dont on pourrait croire que, si l'on n'évalue pas ses composants dans le bon ordre, des catastrophes vont survenir :

```
(if (= 0 q) 1 (/ p q))
```

Cette expression teste, avant de diviser, si le diviseur est nul et dans ce dernier cas retourne la valeur 1. La dénotation dit seulement qu'entre la valeur 1 et le quotient de `p` par `q`, le choix sera effectué suivant la nullité de `q`.

La difficulté de cet opérateur tient à l'habitude que l'on peut avoir prise avec Scheme et au fait que l'on ne voit le λ-calcul qu'à travers la règle d'évaluation de Scheme. On peut dans une certaine mesure diminuer la distance entre Scheme et le λ-calcul en réécrivant la dénotation de l'alternative comme :

$$\mathcal{E}[\![\texttt{(if } \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.((\mathit{boolify}\ \varepsilon) \rightarrow\ \mathcal{E}[\![\pi_1]\!][]\ \mathcal{E}[\![\pi_2]\!]\rho\ \kappa\ \sigma_1)\ \sigma)$$

Un peu de séquentialité a été introduite puisqu'ici les redex ont été ordonnés : la valeur de la conditionnelle ne sert qu'à choisir la dénotation de la branche élue, celle-ci est indépendamment invoquée. Le problème de cette nouvelle définition est de savoir si elle est équivalente à la précédente. Il s'agit donc de prouver si, pour un même programme, les deux dénotations sont les mêmes ? Il suffit pour cela de montrer que :

$$(\mathit{boolify}\ \varepsilon) \rightarrow\ (\mathcal{E}[\![\pi_1]\!]\ \rho\ \kappa\ \sigma)[]\ (\mathcal{E}[\![\pi_2]\!]\ \rho\ \kappa\ \sigma) = ((\mathit{boolify}\ \varepsilon) \rightarrow\ \mathcal{E}[\![\pi_1]\!][]\ \mathcal{E}[\![\pi_2]\!]\rho\ \kappa\ \sigma)$$

Cette équivalence est manifestement fausse en Scheme du fait de l'ordre d'évaluation. Pour s'en convaincre, il suffit de prendre pour π_2 une expression qui boucle. Toutefois les dénotations sont des termes du λ-calcul et comme telles doivent être comparées suivant les lois du λ-calcul : c'est alors manifestement une simple distribution de l'application sur l'alternative.

À l'encontre de la crypticité de la précédente syntaxe, nous adopterons une notation plus parlante pour le `if-then-else` dénotationel :

$$\mathcal{E}[\![\texttt{(if } \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\ \textbf{if}\ (\mathit{boolify}\ \varepsilon)$$
$$\textbf{then}\ \mathcal{E}[\![\pi_1]\!]$$
$$\textbf{else}\ \mathcal{E}[\![\pi_2]\!]$$
$$\textbf{endif}\ \ \rho\ \kappa\ \sigma_1)\ \sigma)$$

5.2.4 Affectation

L'affectation se dénote simplement. La version ici présentée a pour valeur la valeur nouvellement affectée.

$$\mathcal{E}[\![\texttt{(set! } \nu\ \pi\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\kappa\ \varepsilon\ \sigma_1[(\rho\ \nu) \rightarrow \varepsilon])\ \sigma)$$

$$f[y \rightarrow z] = \lambda x.\ \textbf{if}\ y = x\ \textbf{then}\ z\ \textbf{else}\ (f\ x)\ \textbf{endif}$$

La mémoire est étendue pour refléter l'affectation réalisée : à l'adresse de la variable ν est désormais associée la valeur ε. Cette extension est réalisée par une notation ad hoc et suggestive : $\sigma[\alpha \rightarrow \varepsilon]$. Des notations similaires existent comme $[\alpha \rightarrow \varepsilon]\sigma$ [Sch86] ou $[\varepsilon/\alpha]\sigma$ [Sto77] ou encore $\sigma[\varepsilon/\alpha]$ [Gor88, CR91b].

Enrichissons notre λ-calcul de quelques fonctions supplémentaires. Les séquences sont écrites entre $\langle$ et $\rangle$, la séquence vide étant notée $\langle\rangle$. La concaténation de séquences se note §. Pour extraire des termes d'une séquence, on notera l'extraction du $i^{\text{ème}}$ terme de la séquence par $\langle\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\rangle \downarrow i$ et $\langle\varepsilon_1, \ldots, \varepsilon_n\rangle \dagger i$ la séquence $\langle\varepsilon_{i+1}, \ldots, \varepsilon_n\rangle$ tronquée de ses i premiers termes. La notation $\#\varepsilon^*$ désigne la longueur de la séquence ε^*. Toutes ces notations pourraient se définir en λ-calcul pur mais obscurciraient la présentation. Sans sacrilège aucun, $\downarrow_1$, $\dagger 1$, $\#$ et § sont les équivalents dénotationnels de `car`, `cdr`, `length` et `append`.

L'extension d'environnement sera elle-même étendue à une suite de points et d'images. On suppose dans ce qui suit que les deux séquences x^* et y^* ont même longueur.

$$f[y^* \stackrel{*}{\rightarrow} z^*] = \ \textbf{if}\ \#y^* > 0\ \textbf{then}\ f[y^* \dagger 1 \stackrel{*}{\rightarrow} z^* \dagger 1][y^* \downarrow_1 \rightarrow z^* \downarrow_1]\ \textbf{else}\ f\ \textbf{endif}$$

5.2.5 Abstraction

Dans un premier temps, nous ne dénoterons que les fonctions d'arité fixe, c'est-à-dire dépourvues de variable pointée.

$$\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{) } \pi^+\texttt{)}]\!]\rho\kappa\sigma =$$
$$(\kappa\ \textit{in}\textbf{Valeur}(\lambda\ \varepsilon^*\kappa_1\sigma_1.$$
$$\textbf{if}\ \#\varepsilon^* = \#\nu^*$$
$$\textbf{then}\ \ \textit{allocate}\ \sigma_1\ \#\nu^*$$
$$\lambda\ \sigma_2\alpha^*.$$
$$(\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \stackrel{*}{\rightarrow} \alpha^*]\ \kappa_1\ \sigma_2[\alpha^* \stackrel{*}{\rightarrow} \varepsilon^*])$$
$$\textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\ \ )\ \sigma)$$

On observera l'injection *in***Valeur** qui prend un λ-terme représentant une fonction et la convertit en une valeur. L'opération inverse apparaîtra lors de l'application fonctionnelle.

Lorsqu'une fonction est invoquée et après vérification de son arité, de nouvelles adresses sont allouées pour être associées aux variables de la fonction et contenir les valeurs qu'elles prennent pour cette invocation. L'allocation d'adresses au sein de la mémoire s'effectuera grâce à la fonction *allocate* qui prend une mémoire, le nombre d'adresses à allouer et une sorte de « continuation » qu'elle invoquera sur les adresses allouées et la nouvelle mémoire où ces adresses ont été allouées. *allocate* est une vraie fonction qui, à arguments égaux, fait correspondre des résultats égaux mais dont la définition exacte est généralement laissée dans l'ombre pour ne pas encombrer les dénotations et parce qu'elle n'a pas d'intérêt à cause de sa basse technicité.

La fonction *allocate* est polymorphe (α peut représenter n'importe quel type) et a pour signature :

$$\textbf{Mémoire} \times \textbf{EntierNaturel} \times (\textbf{Mémoire} \times \textbf{Adresse}^* \to \alpha) \to \alpha$$

Une définition précise de *allocate* apparaissait au chapitre précédent. **[☞ p. 125]**

5.2.6 Application fonctionnelle

Les fonctions sont faites pour être appliquées, voici la dénotation de l'application. Là encore nous utiliserons une nouvelle valuation auxiliaire : $\mathcal{E}^*$, une sorte de `evlis` dénotationnel.

$$\mathcal{E}[\![(\pi\ \pi^*)]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varphi\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma_2.(\varphi\ |_{\textbf{Fonction}}\ \varepsilon^*\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\ ]\!]\rho\kappa\sigma = (\kappa\ \langle\rangle\ \sigma)$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma_2.(\kappa\ \langle\varepsilon\rangle\S\varepsilon^*\ \sigma_2)\ \sigma_1)\ \sigma)$$

On notera que les continuations qu'utilise $\mathcal{E}^*$ attendent non pas une valeur mais une séquence de valeurs, ce qui n'était pas le cas de la valuation $\mathcal{E}^+$. Les valeurs de κ dans ces définitions ont donc le type suivant :

$$\textbf{Valeur}^* \times \textbf{Mémoire} \to \textbf{Valeur}$$

5.2.7 `call/cc`

Notre rapide tour des dénotations ne serait pas complet si l'on ne décrivait cette fonction essentielle à la sémantique de Scheme : `call/cc`. Celle-ci se décrit comme :

$$(\sigma_0(\rho_0[\![\texttt{call/cc}]\!])) =$$
$$\textit{in}\textbf{Valeur}(\lambda\ \varepsilon^*\kappa\sigma.$$
$$\quad\textbf{if}\ 1 = \#\varepsilon^*$$
$$\quad\textbf{then}\ (\ \varepsilon^*\downarrow_1|_{\textbf{Fonction}}$$
$$\quad\quad\langle\textit{in}\textbf{Valeur}(\lambda\ \varepsilon^*{}_1\kappa_1\sigma_1.$$
$$\quad\quad\quad\textbf{if}\ 1 = \#\varepsilon^*{}_1$$
$$\quad\quad\quad\textbf{then}\ (\kappa\ \varepsilon^*{}_1\downarrow_1\ \ \sigma_1)$$
$$\quad\quad\quad\textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\quad\quad\quad\textbf{endif}\quad)\rangle\ \kappa\ \sigma)$$

else *wrong* "`Incorrect arity`"
endif)

On notera les injections et projections succcessives entre les différents domaines **Valeur** et **Fonction**. La dénotation en elle-même n'est pas vraiment plus complexe que toutes les programmations antérieures de `call/cc` et notamment celle vue au chapitre 3. **[☞ p. 90]**

5.2.8 Conclusions partielles

Nous venons d'achever de définir, cas par cas, une fonction qui à tout programme associe un λ-terme. L'existence de cette fonction ne fait pas de doute car nous avons usé du principe de compositionalité. Si nous tolérions des programmes syntaxiquement récursifs comme en [Que92a], il nous faudrait un peu plus de théorie pour prouver que nous avons bien défini une fonction.

Nous pouvons maintenant prouver que la sémantique des formes spéciales primitives de Scheme tient en un seul regard (Tableau 5.1).

Bien sûr, il y manque la citation (la forme spéciale `quote`) qui pose des problèmes **[☞ p. 133]** et les fonctions d'arité variable (voir plus loin). Il y manque aussi la comparaison de fonctions par `eq?` et nombre de fonctions prédéfinies. On a toutefois fait figurer la fonction primitive `call/cc` qui appparaît sous sa plus simple expression. Les autres fonctions de la bibliothèque prédéfinie comme `cons`, `car`, `set-cdr!` sont simulables dans ce sous-ensemble par des fermetures et n'ajoutent point de nouveaux traits cachés. C'est pourquoi nous parlons d'essence indépendamment des fonctions additionnelles qui pourraient être ajoutées. La base du langage, c'est-à-dire ses formes spéciales et les fonctions primitives comme `call/cc`, suffit pour ancrer la compréhension.

On avouera tout de même que contempler l'essence de Scheme en un seul tableau et avec ce degré de précision valait bien cet austère encodage. Nous avons donc mis une sorte de terme à notre progression puisque partant de Scheme entier pour décrire approximativement Scheme au premier chapitre, nous sommes parvenus au λ-calcul pour décrire finement Scheme tout entier.

5.3 Sémantique du λ-calcul

Décrire l'essence d'un langage en si peu de signes est un atout pour Scheme et pour les langages fonctionnels en général, ils deviennent par là même de véritables laboratoires d'expérimentation linguistique pour l'introduction de constructions nouvelles et leur étude en confrontation avec les traits basiques, fondamentaux, bref communs. Suivant la nature des caractéristiques que l'on veut analyser, on pourra bien sûr partir d'une base linguistique plus restreinte : à savoir le λ-calcul lui-même ! Cela n'est pas tautologique et donnera un second exemple de dénotation d'un langage différent mais parent : la sémantique du λ-calcul lui-même !

Fixons tout d'abord la syntaxe. Comme cela n'a ici aucune importance sinon de fixer les idées, nous choisirons une syntaxe à la Scheme :

```
x        (lambda (x) M)        (M N)
```

$$\mathcal{E}[\![\nu]\!] = \lambda\rho\kappa\sigma.(\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

$$\mathcal{E}[\![\texttt{(set!}\ \nu\ \pi\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\kappa\ \varepsilon\ \sigma_1[(\rho\ \nu) \rightarrow \varepsilon])\ \sigma)$$

$$\begin{array}{l}\mathcal{E}[\![\texttt{(if}\ \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\kappa\sigma = \\ (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.\ \textbf{if}\ (\textit{boolify}\ \varepsilon) \\ \qquad\qquad \textbf{then}\ (\mathcal{E}[\![\pi_1]\!]\ \rho\ \kappa\ \sigma_1) \\ \qquad\qquad \textbf{else}\ (\mathcal{E}[\![\pi_2]\!]\ \rho\ \kappa\ \sigma_1) \\ \qquad\qquad \textbf{endif}\ \ \sigma)\end{array}$$

$$\begin{array}{l}\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{)}\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = \\ (\kappa\ \textit{in}\textbf{Valeur}(\lambda\ \varepsilon^*\kappa_1\sigma_1. \\ \qquad\qquad \textbf{if}\ \#\varepsilon^* = \#\nu^* \\ \qquad\qquad \textbf{then}\ \textit{allocate}\ \sigma_1\ \#\nu^* \\ \qquad\qquad\qquad \lambda\ \sigma_2\alpha^*. \\ \qquad\qquad\qquad\quad (\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*]) \\ \qquad\qquad \textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”} \\ \qquad\qquad \textbf{endif}\ \ )\ \sigma)\end{array}$$

$$\mathcal{E}[\![\texttt{(}\pi\ \pi^*\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varphi\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma_2.(\varphi\ |_{\textbf{Fonction}}\ \varepsilon^*\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma_2.(\kappa\ \langle\varepsilon\rangle\S\varepsilon^*\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\ ]\!]\rho\kappa\sigma = (\kappa\ \langle\rangle\ \sigma)$$

$$\mathcal{E}[\![\texttt{(begin}\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi\ \pi^+]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma_1)\ \sigma)$$

$$\begin{array}{l}(\sigma_0(\rho_0[\![\texttt{call/cc}]\!])) = \\ \textit{in}\textbf{Valeur}(\lambda\ \varepsilon^*\kappa\sigma. \\ \qquad\qquad \textbf{if}\ 1 = \#\varepsilon^* \\ \qquad\qquad \textbf{then}\ (\ \varepsilon^*\downarrow_1|_{\textbf{Fonction}} \\ \qquad\qquad\qquad \langle\textit{in}\textbf{Valeur}(\lambda\ \varepsilon^*{}_1\kappa_1\sigma_1. \\ \qquad\qquad\qquad\qquad\qquad \textbf{if}\ 1 = \#\varepsilon^*{}_1 \\ \qquad\qquad\qquad\qquad\qquad \textbf{then}\ (\kappa\ \varepsilon^*{}_1\downarrow_1\ \ \sigma_1) \\ \qquad\qquad\qquad\qquad\qquad \textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”} \\ \qquad\qquad\qquad\qquad\qquad \textbf{endif}\ \ )\rangle\ \kappa\ \sigma) \\ \qquad\qquad \textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”} \\ \qquad\qquad \textbf{endif}\ \ )\end{array}$$

TABLE 5.1 – Essence de Scheme

Déterminons maintenant les domaines que nous aurons à manipuler. Le λ-calcul ignore la notion d'affectation ainsi que la notion de continuation, nous en ferons donc l'économie. Par ailleurs, nous nous limiterons à un λ-calcul non appliqué avec des fermetures comme seules valeurs. Voici donc les domaines (que l'on peut voir comme une restriction des domaines de Scheme) :

$$\begin{array}{lll} \pi & \textbf{Programme} & \\ \nu & \textbf{Identificateur} & \\ \rho & \textbf{Environnement} & = \textbf{Identificateur} \rightarrow \textbf{Valeur} \\ \varepsilon & \textbf{Valeur} & = \textbf{Fonction} \\ \varphi & \textbf{Fonction} & = \textbf{Valeur} \rightarrow \textbf{Valeur} \end{array}$$

Nous nommerons $\mathcal{L}$ la fonction de valuation, c'est-à-dire celle qui associe à un λ-terme sa dénotation : un autre λ-terme. La valuation a donc pour signature :

$$\mathcal{L} : \textbf{Programme} \rightarrow (\textbf{Environnement} \rightarrow \textbf{Valeur})$$

Il ne reste plus qu'à définir cette fonction, cas par cas, en analysant les différentes syntaxes possibles :

$$\mathcal{L}[\![\nu]\!]\rho = (\rho\ \nu)$$

$$\mathcal{L}[\![\texttt{(lambda (}\nu\texttt{) }\pi\texttt{)}]\!]\rho = \lambda\varepsilon.(\mathcal{L}[\![\pi]\!]\ \rho[\nu \rightarrow \varepsilon])$$

$$\mathcal{L}[\![\texttt{(}\pi\ \pi'\texttt{)}]\!]\rho = ((\mathcal{L}[\![\pi]\!]\ \rho)(\mathcal{L}[\![\pi']\!]\ \rho))$$

TABLE 5.2 – Sémantique du λ-calcul

Notez que cette dénotation est scrupuleuse en ce qui concerne le respect de l'ordre d'évaluation des termes d'une application fonctionnelle (ou combinaison). En effet, l'application est transformée en une application et rien n'est dit sur l'ordre.

La valuation $\mathcal{L}$ est définie récursivement ; il n'y a pas de problème ici car, du fait de la compositionalité, on remarquera que tous les appels récursifs sont effectués sur des programmes plus petits, le cas terminal étant fourni par les références aux variables.

Le cas du λ-calcul est un cas particulier car nous avons déjà une idée très nette de la sémantique de ses termes. Ce que l'on peut prouver (voir [Sto77, page 158]), c'est que le passage à la dénotation conserve les propriétés et notamment la β-réduction.

La dénotation du λ-calcul est la base de la dénotation des langages fonctionnels sans effet de bord ni continuation. Lorsque ceux-ci sont introduits, il est en général nécessaire pour les maîtriser d'introduire un ordre d'évaluation explicite. Si l'on veut ajouter l'affectation, il faut alors scinder l'environnement en introduisant des adresses, les fameuses boîtes du chapitre 4 ou les références de ML. **[☞ p. 108]**

5.4 Fonctions d'arité variable

Dans cette section, nous allons montrer comment incorporer des fonctions d'arité variable à Scheme. Ces fonctions ont ceci de particulier qu'elles manipulent l'excès

d'arguments qu'elles reçoivent en une liste : il y a donc, à chaque invocation, une allocation déguisée de paires pointées qui peut, par ailleurs, coûter fort cher si ces fonctions sont très souvent utilisées. D'une certaine manière, l'antidote des fonctions à liste de variable pointée est la fonction primitive `apply` qui, elle, convertit une liste de valeurs en des arguments manquants. Les fonctions d'arité variable sont donc, en Scheme, indissociables des listes, aussi devons-nous définir les dénotations des fonctions usuelles sur les listes comme `cons`, `car` ou `set-cdr!`.

Les paires sont représentées par le domaine **Paire** qui apparaît comme l'un des composants de la somme disjointe définissant **Valeur**. Le domaine **Paire** sera lui-même défini comme au chapitre précédent par le produit cartésien de deux adresses.

$$\begin{aligned} \textbf{Valeur} &= \textbf{Fonction} + \textbf{Booléen} + \textbf{Entier} + \textbf{Paire} + \ldots \\ \textbf{Paire} &= \textbf{Adresse} \times \textbf{Adresse} \end{aligned}$$

Les dénotations des `cons`, `car` et autres `set-cdr!` (afin d'exhiber au moins un effet de bord sur les paires) sont classiques et ne diffèrent guère de la programmation qui en avait été donnée au chapitre précédent. Les seules difficultés sont maintenant notationnelles car il faut, par exemple, interpréter $\varepsilon^* \downarrow_1 |_{\textbf{Paire}} \downarrow_2$ comme suit : $\varepsilon^* \downarrow_1$ est le premier argument de la fonction : une valeur, qui est ensuite projetée sur le domaine des paires $|_{\textbf{Paire}}$ (on obtient $\perp$ si ce n'est pas une paire) ; on extrait enfin de cette paire l'adresse de son champ `cdr`, c'est-à-dire sa seconde composante $\downarrow_2$.

$$\begin{array}{l}
(\sigma_0(\rho_0[\![\texttt{cons}]\!])) = \\
\textit{in}\textbf{Valeur}(\lambda\varepsilon^*\kappa\sigma.\ \ \textbf{if}\ 2 = \#\varepsilon^* \\
\qquad\qquad\qquad\qquad \textbf{then}\ \textit{allocate}\ \sigma\ 2\ \ \lambda\sigma_1\alpha^*.(\kappa\ \textit{in}\textbf{Valeur}(\langle\alpha^*\downarrow_1, \alpha^*\downarrow_2\rangle)\ \sigma_1[\alpha^* \xrightarrow{*} \varepsilon^*]) \\
\qquad\qquad\qquad\qquad \textbf{else}\ \textit{wrong}\ \text{“}\texttt{incorrect arity}\text{”} \\
\qquad\qquad\qquad\qquad \textbf{endif}\ \ )
\end{array}$$

$$\begin{array}{l}
(\sigma_0(\rho_0[\![\texttt{car}]\!])) = \\
\textit{in}\textbf{Valeur}(\lambda\varepsilon^*\kappa\sigma.\ \ \textbf{if}\ 1 = \#\varepsilon^* \\
\qquad\qquad\qquad\qquad \textbf{then}\ (\kappa\ (\sigma\ \varepsilon^*\downarrow_1|_{\textbf{Paire}}\downarrow_1)\ \sigma) \\
\qquad\qquad\qquad\qquad \textbf{else}\ \textit{wrong}\ \text{“}\texttt{incorrect arity}\text{”} \\
\qquad\qquad\qquad\qquad \textbf{endif}\ \ )
\end{array}$$

$$\begin{array}{l}
(\sigma_0(\rho_0[\![\texttt{set-cdr!}]\!])) = \\
\textit{in}\textbf{Valeur}(\lambda\varepsilon^*\kappa\sigma.\ \ \textbf{if}\ 2 = \#\varepsilon^* \\
\qquad\qquad\qquad\qquad \textbf{then}\ (\kappa\ \varepsilon^*\downarrow_1\ \ \sigma[\varepsilon^*\downarrow_1|_{\textbf{Paire}}\downarrow_2 \rightarrow \varepsilon^*\downarrow_2]) \\
\qquad\qquad\qquad\qquad \textbf{else}\ \textit{wrong}\ \text{“}\texttt{incorrect arity}\text{”} \\
\qquad\qquad\qquad\qquad \textbf{endif}\ \ )
\end{array}$$

Une fois connue la structure des listes, `apply` devient simple à spécifier. On accumule les arguments à partir du second (puisque le premier est la fonction à invoquer) jusqu'au dernier exclu. Celui-ci doit être une liste que l'on désosse et dont on accumule les termes successifs dans la séquence des arguments sur laquelle on appliquera la fonction spécifiée.

$$\begin{array}{l}
(\sigma_0(\rho_0[\![\texttt{apply}]\!])) = \\
\textit{in}\textbf{Valeur}(\lambda\varepsilon^*\kappa\sigma.\ \ \textbf{if}\ \#\varepsilon^* \geq 2 \\
\qquad\qquad\qquad\qquad \textbf{then}\ \ (\varepsilon^*\downarrow_1|_{\textbf{Fonction}}\ (\textit{collect}\ \varepsilon^* \dagger 1)\ \kappa\ \sigma) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{whererec}\ \textit{collect} = \lambda{\varepsilon^*}_1.\ \ \textbf{if}\ {\varepsilon^*}_1 \dagger 1 = \langle\rangle
\end{array}$$

$$\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } (\mathit{flat}\ \varepsilon^*{}_1{\downarrow_1}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \langle \varepsilon^*{}_1{\downarrow_1}\rangle \S(\mathit{collect}\ \varepsilon^*{}_1 \dagger 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{endif} \\
\qquad\qquad \textbf{and } \mathit{flat} = \lambda\varepsilon.\ \textbf{if } \varepsilon \in \textbf{Pair} \\
\qquad\qquad\qquad\qquad\qquad \textbf{then } \langle(\sigma\ \varepsilon\ |_{\textbf{Paire}}{\downarrow}1)\rangle \S(\mathit{flat}\ (\sigma\ \varepsilon\ |_{\textbf{Paire}}{\downarrow}2)) \\
\qquad\qquad\qquad\qquad\qquad \textbf{else } \langle\rangle \\
\qquad\qquad\qquad\qquad\qquad \textbf{endif} \\
\quad \textbf{else } \mathit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”} \\
\quad \textbf{endif}\ )
\end{array}$$

Nous pouvons maintenant aborder les fonctions d'arité variable. Pour cela, nous allons changer la dénotation de la forme spéciale `lambda` et introduire une fonction de valuation particulière dont le seul rôle sera de créer les liaisons variable-valeur. Nous noterons $\mathcal{B}$ cette nouvelle valuation (avec $\mathcal{B}$ pour *binding*). Sa signature est apparentée à celle des dénotations, la signature de $\mathcal{E}$. De façon plus précise et en usant du raccourci abréviationnel $\tau \ldots$:

$$\begin{array}{rcl}
\tau \ldots & \equiv & \textbf{Valeur}^* \times \textbf{Environment} \times \textbf{Continuation} \times \textbf{Mémoire} \\
\mathcal{E}: & & \textbf{Programme} \rightarrow \tau \ldots \rightarrow \textbf{Valeur} \\
\mathcal{B}: & & \textbf{ListedeVariables} \rightarrow (\tau \ldots \rightarrow \textbf{Valeur}) \times \tau \ldots \rightarrow \textbf{Valeur}
\end{array}$$

La valuation de liaison, $\mathcal{B}$, ne liera les variables aux adresses qui contiendront leur valeur, qu'à l'issue d'une vérification d'arité effectuée par `lambda`. En cas de succès, la liste des variables est parcourue, les locations idoines sont tour à tour allouées ; l'environnement lexical de définition de la fonction est progressivement enrichi et finalement le corps de la fonction est évalué. Voici la nouvelle présentation des fonctions d'arité fixe.

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{)}\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = \\
(\kappa\ \mathit{in}\textbf{Valeur}(\lambda\varepsilon^*\kappa_1\sigma_1.\ \textbf{if } \#\varepsilon^* = \#\nu^* \\
\qquad\qquad\qquad\qquad \textbf{then } ((\mathcal{B}[\![\nu^*]\!]\ \lambda\varepsilon^*{}_1\rho_1\kappa_2\sigma_2.(\mathcal{E}^+[\![\pi^+]\!]\ \rho_1\ \kappa_2\ \sigma_2))\varepsilon^*\ \rho\ \kappa_1\ \sigma_1) \\
\qquad\qquad\qquad\qquad \textbf{else } \mathit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”} \\
\qquad\qquad\qquad\qquad \textbf{endif}\ )\ \sigma)
\end{array}$$

$$\mathcal{B}[\![\nu\ \nu^*]\!]\mu = (\mathcal{B}[\![\nu]\!]\ (\mathcal{B}[\![\nu^*]\!]\ \mu))$$

$$\mathcal{B}[\![\ ]\!]\mu = \mu$$

$$\begin{array}{l}
\mathcal{B}[\![\nu]\!]\mu = \\
\lambda\varepsilon^*\rho\kappa\sigma.\mathit{allocate}\ \sigma\ 1\ \ \lambda\sigma_1\alpha^*.\ \textbf{let } \alpha = \alpha^*{\downarrow_1} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } (\mu\ \varepsilon^* \dagger 1\ \rho[\nu \rightarrow \alpha]\ \kappa\ \sigma_1[\alpha \rightarrow \varepsilon^*{\downarrow_1}])
\end{array}$$

Pour traiter les fonctions d'arité variable, nous introduisons un nouveau cas de dénotation des `lambda` avec liste de variables pointée ainsi que la clause de liaison appropriée. Cette dernière prend une séquence de valeurs, la convertit en une liste de valeurs, une vraie liste formée de vraies paires, et la lie à la variable pointée. La coexistence des fonctions d'arité multiple, de la fonction `apply` et des effets de bord impose, comme spécifié en Scheme, que ce soient des paires pointées *fraîches* qui soient allouées. On ne doit point espérer que l'expression suivante retourne Vrai :

```
(let ((arguments (list 1 2 3)))
  (apply (lambda args (eq? args arguments)) arguments) )
```

Un évaluateur qui souhaiterait partager des paires pointées doit au préalable prouver qu'il n'altère pas, ce faisant, la sémantique du programme.

Voici donc les dénotations des fonctions d'arité multiple.

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda (}\nu^*\ .\ \nu\texttt{)}\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = \\
(\kappa\ \textit{in}\textbf{Valeur}(\lambda\,\varepsilon^*\kappa_1\sigma_1. \\
\qquad\qquad \textbf{if}\ \#\varepsilon^* \geq \#\nu^* \\
\qquad\qquad \textbf{then}\ (\ (\mathcal{B}[\![\nu^*]\!]\ (\mathcal{B}[\![.\ \nu]\!]\ \lambda\,\varepsilon^*{}_1\rho_1\kappa_2\sigma_2. \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathcal{E}^+[\![\pi^+]\!]\ \rho_1\ \kappa_2\ \sigma_2)\ ))) \\
\qquad\qquad\qquad\qquad \varepsilon^*\ \rho\ \kappa_1\ \sigma_1) \\
\qquad\qquad \textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”} \\
\qquad\qquad \textbf{endif}\quad)\ \sigma)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{B}[\![.\ \nu]\!]\mu = \\
\lambda\,\varepsilon^*\rho\kappa\sigma. \\
\quad (\textit{listify}\ \varepsilon^*\ \sigma\ \lambda\,\varepsilon\sigma_1. \\
\qquad\qquad \textit{allocate}\ \sigma_1\ 1 \\
\qquad\qquad\ \lambda\,\sigma_2\alpha^*. \\
\qquad\qquad\quad \textbf{let}\ \alpha = \alpha^*\downarrow_1 \\
\qquad\qquad\quad \textbf{in}\ (\mu\ \langle\rangle\ \rho[\nu \to \alpha]\ \kappa\ \sigma_2[\alpha \to \varepsilon]) \quad) \\
\quad \textbf{whererec}\ \textit{listify} = \lambda\,\varepsilon^*{}_1\sigma_1\kappa_1. \\
\qquad\qquad\qquad\qquad \textbf{if}\ \#\varepsilon^*{}_1 > 0 \\
\qquad\qquad\qquad\qquad \textbf{then}\ \ \textit{allocate}\ \sigma_1\ 2 \\
\qquad\qquad\qquad\qquad\ \lambda\,\sigma_2\alpha^*. \\
\qquad\qquad\qquad\qquad\quad \textbf{let}\ \kappa_2 = \lambda\,\varepsilon\sigma_3. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\kappa_1\ \textit{in}\textbf{Valeur}(\alpha^*)\ \sigma_3[\alpha^*\downarrow_2 \to \varepsilon]) \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\textit{listify}\ \varepsilon^*{}_1\ \dagger\ 1\ \sigma_2[\alpha^*\downarrow_1 \to \varepsilon^*{}_1\downarrow_1]\ \kappa_2) \\
\qquad\quad \textbf{else}\ (\kappa_1\ \textit{in}\textbf{Valeur}(\langle\rangle)\ \sigma_1) \\
\qquad\quad \textbf{endif}
\end{array}
$$

On le voit la dénotation de caractéristiques non triviales de Scheme telles que les fonctions d'arité variable nécessite un effort de programmation dénotationnelle non négligeable et apparente ce travail à l'écriture d'un véritable interprète. On comparera l'élégance de la description du noyau et les lignes précédentes qui, pour un trait intéressant mais mineur, double quasiment la taille (sans parler de la crypticité) de la description.

5.5 Ordre d'évaluation des applications

Il y a périodiquement dans les forums électroniques des flambées de violence scripturale sur cette caractéristique des différents documents normatifs de Scheme, ce parti pris jusqu'ici immuable, de ne point préciser l'ordre d'évaluation des termes d'une application fonctionnelle. Ne pas prescrire cet ordre décourage l'écriture de programmes dépendant de cet ordre mais rend particulièrement délicate la recherche de ces erreurs. Nombreux sont les programmes, mêmes écrits par des experts chevronnés, qui dépendent obscurément de l'ordre d'évaluation, surtout lorsque les continuations s'en mêlent. Pour notre part, nous préconisons l'ordre gauche-droite qui correspond

au sens naturel de lecture de nombreuses langues et qui rend la recherche d'erreur plus systématique.

Deux objections des tenants du parti opposé sont intéressantes. La première est que de nombreux langages n'imposent pas cet ordre et parmi eux le langage C. Ainsi si l'on compile directement `(foo (f x) (g x y))` en l'expression C : `foo(f(x),g(x,y))`, rien n'est assuré sur l'ordre produit et il peut être coûteux de l'imposer. L'autre argument est plus subtil : on peut, dans un monde sans ordre, en imposer un en séquentialisant explicitement par `begin` ou `let`. Si, par contre, l'ordre est imposé, alors il n'y a plus moyen d'écrire un programme dont l'ordre est laissé non spécifié. À moins d'utiliser un générateur aléatoire **[☞ Ex. 5.4]** prescrivant (coûteusement) à l'exécution l'ordre particulier à suivre. L'ordre n'est pas imposé en C afin de permettre au compilateur de considérer les termes d'une application dans l'ordre qu'il souhaite ou qu'il trouve plus efficace, notamment en matière d'allocation de registres.

L'ordre explicite simplifie la mise au point des programmes en éliminant une source d'indétermination. Si l'ordre n'est pas prescrit, deux exécutions d'un même programme peuvent être menées différemment et ne pas conduire aux mêmes résultats. Considérez par exemple [8] :

```
(define (dynamically-changing-evaluation-order?)
  (define (amb)
    (call/cc (lambda (k) ((k #t) (k #f)))) )
  (if (eq? (amb) (amb))
      (dynamically-changing-evaluation-order?)
      #t ) )
```

La fonction interne `amb` retourne Vrai ou Faux suivant l'ordre d'évaluation. Si l'ordre d'évaluation change dynamiquement alors la fonction `dynamically-changing-evaluation-order?` s'arrêtera sinon elle bouclera indéfiniment. Le R^{4}RS n'impose ni que ce programme boucle ni qu'il s'arrête. Si l'ordre est imposé, alors manifestement ce programme boucle sans rémission.

Il faut bien distinguer dans cette discussion le langage de l'implantation. Une implantation a obligatoirement à choisir un ordre d'évaluation lorsqu'elle évalue une application, elle peut décider d'adopter un ordre gauche-droite pour toutes les applications, ou un ordre droite-gauche (comme MacScheme), ou un ordre quelconque dépendant de son humeur du moment. Je ne connais pas d'implantation qui ayant choisi l'ordre pour une application particulière en change ensuite dynamiquement. En effet, l'ordre est souvent choisi à la compilation et n'est pas remis en cause par la suite. Le langage n'impose rien mais l'implantation est libre d'en choisir un et de publier son choix : c'est licite.

Le problème auquel nous nous intéressons maintenant est de savoir comment l'on peut spécifier qu'il n'y a pas d'ordre prescrit. La solution que nous proposons est de changer subtilement la structure de la dénotation. Une dénotation était un λ-terme qui attendait un environnement, une continuation et une mémoire pour rendre une valeur. Le choix de rendre une valeur était un peu limité car en fait le résultat d'une évaluation est double et comprend d'une part, une valeur, d'autre part, l'état mémoire résultant. On aurait donc pu tout aussi bien prendre le couple (valeur, mémoire) comme image des dénotations. On transcende généralement cette question en nommant le codomaine

8. Ce programme a été mis au point en collaboration avec Matthias Felleisen.

des dénotations : **Résultat** et en laissant vague sa définition.

Puisque plusieurs ordres d'évaluation sont possibles, alors, plutôt que de retourner une unique réponse, une dénotation pourrait retourner l'ensemble des réponses possibles au sein desquelles une serait choisie, suivant des critères obscurs laissés aux bons soins de l'implantation. Nous allons donc modifier la précédente valuation $\mathcal{E}$ pour retourner maintenant l'ensemble de toutes les réponses possibles et introduire $\mathcal{N}$ pour en choisir une parmi elles. On notera $\mathcal{P}(Q)$ l'ensemble des parties de Q. Les signatures deviennent :

$$\begin{aligned}\mathcal{E} : \quad & \textbf{Programme} \to \textbf{Valeur}^* \times \textbf{Environment} \times \textbf{Continuation} \times \textbf{Mémoire} \\ & \to \mathcal{P}(\textbf{Résultat}) \\ \mathcal{N} : \quad & \textbf{Programme} \to \textbf{Valeur}^* \times \textbf{Environment} \times \textbf{Continuation} \times \textbf{Mémoire} \\ & \to \textbf{Résultat}\end{aligned}$$

La valuation $\mathcal{N}$ se définit simplement, en faisant appel à la fonction *oneof*, dont la définition est laissée à l'implantation et qui se charge de choisir parmi tous les résultats possibles celui qu'elle souhaite.

$$\mathcal{N}[\![\pi]\!]\rho\kappa\sigma = (\mathit{oneof}\ (\mathcal{E}[\![\pi]\!]\ \rho\ \kappa\ \sigma))$$

Il faut maintenant modifier la dénotation de l'application fonctionnelle pour retourner toutes les valeurs possibles. La technique va s'inspirer de la sémantique. Dire qu'il n'y a pas d'ordre, c'est dire qu'en face d'une application, l'évaluateur choisit l'un des termes, disons π_{i_0}, l'évalue pour obtenir sa valeur ε_{i_0}, puis choisit un second terme, disons π_{i_1}, l'évalue à son tour en ε_{i_1} et ainsi de suite jusqu'au dernier des termes. Les valeurs $\varepsilon_{i_0}, \varepsilon_{i_1}, \ldots$ sont alors réordonnées en $\varepsilon_0, \varepsilon_1 \ldots$ et la première d'entre elles est appliquée aux autres. Notez l'ordre des choix tel qu'il a été décrit. Il serait tout à fait différent, comme cela apparaît dans les R[3,4]RS, de fixer l'ordre d'évaluation de tous les termes avant l'évaluation du tout premier. Pour prendre un exemple, la continuation que retourne la fonction suivante imprimera, lorsqu'invoquée, un chiffre ; ce chiffre n'est pas déterminé par la norme Scheme.

```
(define (one-two-three)
  (call/cc (lambda (k)
             ((begin (display 1)(call/cc k))
              (begin (display 2)(call/cc k))
              (begin (display 3)(call/cc k)) ) )) )
```

La dénotation de l'application sans ordre va exactement « implanter » ce que nous avons énoncé plus haut. Elle va considérer tous les choix possibles de termes et pour cela s'aider de la fonction *forall* qui applique son premier argument (une fonction ternaire) à toutes les coupes possibles de son second argument (une liste). La fonction *cut* coupe une liste en deux segments, le premier contenant les i premiers termes de celle-ci et tous les autres termes se situant dans le second. La continuation (le troisième argument de *cut*) est finalement appliquée à ces deux segments. La programmation est assez subtile et est un bon exemple de programmes usant d'un style par continuations. Les définitions qui suivent ont été obtenues en collaboration avec Sophie Anglade et Jean-Jacques Lacrampe [ALQ95].

$$\begin{aligned}&\mathcal{E}[\![(\pi_0\ \pi_1\ \ldots\ \pi_n)]\!]\rho\kappa\sigma = \\ &((\mathit{possible\text{-}paths}\ \langle\mathcal{E}[\![\pi_0]\!], \mathcal{E}[\![\pi_1]\!], \ldots \mathcal{E}[\![\pi_n]\!]\rangle)\rho\ \lambda\varepsilon^*\sigma_1.(\varepsilon^* \downarrow_1 |_{\textbf{Fonction}}\ \varepsilon^* \dagger 1\ \kappa\ \sigma_1)\ \sigma)\end{aligned}$$

$$
\begin{array}{l}
(\textit{possible-paths}\ \mu^+) = \\
\lambda\ \rho\kappa\sigma. \\
\quad \textbf{if}\ \#\mu^+ \dagger 1 > 0 \\
\quad \textbf{then}\ (\textit{forall}\ \lambda\ {\mu^+}_1 \mu \mu^+{}_2. \\
\qquad\qquad (\mu\ \rho\ \lambda\ \varepsilon\sigma_1. \\
\qquad\qquad\quad (\ (\textit{possible-paths}\ {\mu^+}_1 \S \mu^+{}_2) \\
\qquad\qquad\quad\ \rho\ \lambda\ \varepsilon^*\sigma_2. \\
\qquad\qquad\qquad\quad \textbf{let}\ \kappa_1 = \lambda\ {\varepsilon^*}_1 \varepsilon^*{}_2. \\
\qquad\qquad\qquad\qquad\quad (\kappa\ \varepsilon^*{}_1 \S \langle\varepsilon\rangle \S \varepsilon^*{}_2\ \sigma_2) \\
\qquad\qquad\qquad\quad \textbf{in}\ (\textit{cut}\ \#\mu^+{}_1\ \varepsilon^*\ \kappa_1)\ \ \sigma_1)\ \ \sigma)\ \ \mu^+) \\
\quad \textbf{else}\ (\ \mu^+ \downarrow_1 \\
\qquad\quad \rho\ \lambda\ \varepsilon\sigma_1. \\
\qquad\qquad\quad (\kappa\ \langle\varepsilon\rangle\ \sigma_1)\ \ \sigma) \\
\quad \textbf{endif}
\end{array}
$$

$$
\begin{array}{l}
(\textit{forall}\ \varphi\ l) = \\
\ (\textit{loop}\ \langle\rangle\ l\downarrow_1\ \ l \dagger 1) \\
\ \textbf{whererec}\ \textit{loop} = \lambda l_1 \varepsilon l_2. (\varphi\ l_1\ \varepsilon\ l_2) \cup\ \textbf{if}\ \#l_2 > 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ (\textit{loop}\ l_1 \S \langle\varepsilon\rangle\ l_2\downarrow_1\ \ l_2 \dagger 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \emptyset \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{endif}
\end{array}
$$

$$
\begin{array}{l}
(\textit{cut}\ \iota\ \varepsilon^*\ \kappa) = \\
\ (\textit{accumulate}\ \langle\rangle\ \iota\ \varepsilon^*) \\
\ \textbf{whererec}\ \textit{accumulate} = \lambda l \iota_1 l_1.\ \ \textbf{if}\ \iota_1 > 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ (\textit{accumulate}\ \langle l_1\downarrow_1\rangle \S l\ \iota_1 - 1\ l_1 \dagger 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (\kappa\ (\textit{reverse}\ l)\ l_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{endif}
\end{array}
$$

On notera que, pour toutes ces variantes relatives à l'ordre d'évaluation, toutes étaient séquentielles, ce qui est imposé par Scheme. Les termes peuvent être évalués dans le désordre mais les uns après les autres. Les seules réponses possibles au programme suivant sont `(3 5)` ou `(4 3)` mais en aucun cas `(3 3)` :

```
(let ((x 1)(y 2))
  (list (begin (set! x (+ x y)) x)
        (begin (set! y (+ x y)) y) ) )
```

En revanche, la nouvelle valuation $\mathcal{E}$ appliquée au programme suivant permet bien de prévoir les deux réponses différentes possibles : 1 ou 2. Une implantation n'en calculera qu'une seule mais ce sera l'une des deux prévues.

```
(call/cc (lambda (k) ((k 1) (k 2))))    → 1 or 2
```

5.6 Liaison dynamique

Le concept de liaison dynamique est important, utile et a si longtemps prévalu dans les interprètes Lisp qu'il nous semble bon d'en donner une des dénotations possibles. Nous procéderons par extension de la dénotation de Scheme précédemment exposée à laquelle nous ajouterons quelques formes spéciales manipulant ce nouveau type de

liaison. Il y a de nombreuses variétés de liaison dynamique usant de formes spéciales ou de fonctions spécialisées. [☞ **p. 47**] Par tradition, Scheme pousse à cette dernière solution afin de ne pas altérer la dénotation de son noyau. Malheureusement, la présence de certaines fonctions, quoique respectant le protocole d'appel fonctionnel, perturbe cet idéal. Il suffit de simplement penser à `call/cc` qui requiert la notion de continuation. La dénotation des liaisons dynamiques sous-entend la présence d'un environnement destiné à mémoriser ces liaisons. Nous allons donc introduire ce nouvel environnement partout où il est nécessaire. Le noyau d'un Scheme sans fioriture figure dans le tableau 5.3.

On remarquera que l'environnement dynamique, identifié par δ, suit un parcours très différent de ceux que suivent l'environnement lexical ρ (car il est passé en argument aux fonctions qui peuvent ainsi se saisir de l'environnement dynamique courant) ou la mémoire σ qui suit un parcours unifilaire (*single-threaded*) : tout pas de calcul prend la mémoire courante en entrée, la consulte ou la modifie et la passe à son successeur. La mémoire n'existe donc qu'en un unique exemplaire puisque l'on n'a jamais besoin d'une mémoire précédente. L'environnement dynamique n'est pas semblable à la mémoire car, par exemple, il est commun à tous les termes d'une application fonctionnelle qui le partagent tous.

L'environnement dynamique est défini par le domaine **DynEnv** :

$$\delta \quad \textbf{DynEnv} \quad = \textbf{Identificateur} \rightarrow \textbf{Valeur}$$

Les formes spéciales de manipulation des liaisons dynamiques sont au nombre de deux : la première `dynamic-let` établit une liaison dynamique entre une *variable* et une valeur *value* pendant le calcul de son corps *body*... ; `dynamic-let` ne sait traiter qu'une unique variable : cela ne diminue en rien la puissance du langage sinon le confort qu'il procure. La seconde `dynamic` retourne la valeur d'une variable dynamique (et provoque une erreur si la variable n'est pas définie). Comme nous n'avons pas procuré la possibilité de modifier une telle liaison, l'environnement dynamique associe directement des noms de variables à des valeurs sans l'intercession d'une adresse. Leur syntaxe suit ; leur sémantique est visible au tableau 5.4 :

```
(dynamic-let (variable value) body...)
(dynamic variable)
```

Les deux dénotations sont très simples ; elles enrichissent ou scrutent l'environnement dynamique et procure ainsi un nouvel espace de noms : les variables dynamiques qui ne peuvent être confondues avec les vraies variables lexicales. On mesurera, une fois de plus, sur ce petit exemple de quelques lettres grecques, le pouvoir des dénotations qui livrent leur génie à qui sait les déchiffrer.

La notion d'environnement dynamique est très importante, elle peut non seulement servir aux variables dynamiques mais aussi aux fonctions de traitement d'erreur, aux échappements à durée de vie dynamique ou à des concepts de contrôle de calculs [HD90, QD93]. Un protocole simple de rattrapage d'erreurs serait, à chaque survenue d'anomalie, que l'implantation construise un objet décrivant l'anomalie et lui applique la valeur de la variable dynamique `*error*`. On peut alors se protéger d'une éventuelle erreur grâce à une forme `dynamic-let` liant `*error*` à la fonction ad hoc. Pour donner un exemple de ce protocole, Scheme spécifie que l'ouverture d'un fichier inexistant signale une erreur mais ne fournit pas de fonction permettant de savoir si un fichier

$$\mathcal{E}[\![\nu]\!]\rho\delta\kappa\sigma = (\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

$$\mathcal{E}[\![\texttt{(if}\ \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\delta\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\ \textbf{if}\ (\textit{boolify}\ \varepsilon)$$
$$\textbf{then}\ \mathcal{E}[\![\pi_1]\!]$$
$$\textbf{else}\ \mathcal{E}[\![\pi_2]\!]$$
$$\textbf{endif}\ \ \rho\ \delta\ \kappa\ \sigma_1)\ \sigma)$$

$$\mathcal{E}[\![\texttt{(set!}\ \nu\ \pi\texttt{)}]\!]\rho\delta\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\kappa\ \varepsilon\ \sigma_1[(\rho\ \nu) \rightarrow \varepsilon])\ \sigma)$$

$$\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{)}\ \pi^+\texttt{)}]\!]\rho\delta\kappa\sigma =$$
$$(\kappa\ \textit{in}\textbf{Valeur}(\lambda\ \varepsilon^*\delta_1\kappa_1\sigma_1.$$
$$\textbf{if}\ \#\varepsilon^* = \#\nu^*$$
$$\textbf{then}\ \textit{allocate}\ \sigma_1\ \#\nu^*$$
$$\lambda\ \sigma_2\alpha^*.$$
$$(\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \delta_1\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*])$$
$$\textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\ \ )\ \sigma)$$

$$\mathcal{E}[\![\texttt{(}\pi\ \pi^*\texttt{)}]\!]\rho\delta\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\ \varphi\sigma_1.$$
$$(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \delta\ \lambda\ \varepsilon^*\sigma_2.$$
$$(\ \varphi\ |_{\textbf{Fonction}}$$
$$\varepsilon^*\ \delta\ \kappa\ \sigma_2)\ \ \sigma_1)\ \ \sigma)$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\rho\delta\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \delta\ \lambda\varepsilon^*\sigma_2.(\kappa\ \langle\varepsilon\rangle\S\varepsilon^*\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\ ]\!]\rho\delta\kappa\sigma = (\kappa\ \langle\rangle\ \sigma)$$

$$\mathcal{E}[\![\texttt{(begin}\ \pi^+\texttt{)}]\!] = \mathcal{E}^+[\![\pi^+]\!]$$

$$\mathcal{E}^+[\![\pi]\!]\rho\delta\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi\ \pi^+]\!]\rho\delta\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \delta\ \kappa\ \sigma_1)\ \sigma)$$

$$(\sigma_0(\rho_0[\![\texttt{call/cc}]\!])) =$$
$$\textit{in}\textbf{Valeur}(\lambda\ \varepsilon^*\delta\kappa\sigma.$$
$$\textbf{if}\ 1 = \#\varepsilon^*$$
$$\textbf{then}\ (\ \varepsilon^*\downarrow_1|_{\textbf{Fonction}}$$
$$\langle\textit{in}\textbf{Valeur}(\lambda\ \varepsilon^*{}_1\delta_1\kappa_1\sigma_1.$$
$$\textbf{if}\ 1 = \#\varepsilon^*{}_1$$
$$\textbf{then}\ (\kappa\ \varepsilon^*{}_1\downarrow_1\ \ \sigma_1)$$
$$\textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\ \ )\rangle\ \delta\ \kappa\ \sigma)$$
$$\textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\ \ )$$

TABLE 5.3 – Scheme et liaison dynamique

$$(\delta_0\ \nu) = \mathit{wrong}\ \text{“No such dynamic variable”}$$

$$\mathcal{E}[\![\texttt{(dynamic}\ \nu\texttt{)}]\!]\rho\delta\kappa\sigma = (\kappa\ (\delta\ \nu)\ \sigma)$$

$$\mathcal{E}[\![\texttt{(dynamic-let (}\nu\ \pi\texttt{)}\ \pi^{+}\texttt{)}]\!]\rho\delta\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\mathcal{E}^{+}[\![\pi^{+}]\!]\ \rho\ \delta[\nu \rightarrow \varepsilon]\ \kappa\ \sigma_1)\ \sigma)$$

TABLE 5.4 – Formes spéciales de liaison dynamique

existe. On peut donc programmer un prédicat indiquant si un fichier existe au moment exact où est appelé le prédicat en écrivant :

```
(define (probe-file filename)
  (dynamic-let (*error* (lambda (anomaly) #f))
    (call-with-input-file filename
      (lambda (port) #t) ) ) )
```

C'est une définition approximative car il faudrait en fait tester que l'anomalie correspond bien à l'erreur « fichier absent » et non, par exemple, à l'erreur « plus de canaux d'entrée disponibles ». Pour traiter correctement ces erreurs, l'évaluation doit coopérer en construisant de bons objets pour représenter les dysfonctionnements.

5.7 Environnement global

Nous allons, dans cette section, nous intéresser à l'environnement global et pour cela le dénoter de plusieurs façons. Tout comme nous avons procédé lors de l'introduction des liaisons dynamiques, nous allons dénoter l'essence de Scheme en y ajoutant un environnement global. Tout comme l'environnement local, ρ, cet environnement global, γ, transformera des identificateurs en des adresses. L'environnement global accompagnera fidèlement la mémoire : tout pas de calcul retournera une mémoire et un environnement global éventuellement modifié. Le tableau 5.5 définit un Scheme essentiel avec environnement global explicite. Nous avons cependant ôté de cette description, les dénotations concernant la référence à une variable ainsi que l'affectation.

5.7.1 L'environnement global en Scheme

Sur cette base, nous allons pouvoir construire les multiples définitions possibles pour l'environnement global. Le standard Scheme stipule *(i)* que l'on ne peut obtenir la valeur d'une variable que si elle existe ; *(ii)* que l'on ne peut modifier une variable que si elle existe ; *(iii)* que la redéfinition d'une variable est semblable à une affectation.

Pour exprimer les deux premières règles dénotationnellement, il nous faut pouvoir tester si une variable apparaît ou pas dans un environnement. Au lieu donc que le codomaine des environnements soit l'espace des adresses, nous enrichirons ce codomaine avec un point étranger aux adresses et qui dénotera l'absence d'une variable, ce qu'en termes dénotationnels, nous noterons :

$$\mathcal{E}[\![\texttt{(if } \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\gamma\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.\ \textbf{if}\ (\textit{boolify}\ \varepsilon)$$
$$\textbf{then}\ (\mathcal{E}[\![\pi_1]\!]\ \rho\ \gamma_1\ \kappa\ \sigma_1)$$
$$\textbf{else}\ (\mathcal{E}[\![\pi_2]\!]\ \rho\ \gamma_1\ \kappa\ \sigma_1)$$
$$\textbf{endif}\quad \sigma)$$

$$\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{) } \pi^+\texttt{)}]\!]\rho\gamma\kappa\sigma =$$
$$(\kappa\ \textit{in}\textbf{Valeur}(\lambda\ \varepsilon^*\gamma_1\kappa_1\sigma_1.$$
$$\textbf{if}\ \#\varepsilon^* = \#\nu^*$$
$$\textbf{then}\ \textit{allocate}\ \sigma_1\ \#\nu^*$$
$$\lambda\ \sigma_2\alpha^*.$$
$$(\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \gamma_1\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*])$$
$$\textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\quad)\ \gamma\ \sigma)$$

$$\mathcal{E}[\![\texttt{(}\pi\ \pi^*\texttt{)}]\!]\rho\gamma\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varphi\gamma_1\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \gamma_1\ \lambda\varepsilon^*\gamma_2\sigma_2.(\varphi\ |_{\textbf{Fonction}}\ \varepsilon^*\ \gamma_2\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\rho\gamma\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \gamma\ \lambda\varepsilon^*\gamma_2\sigma_2.(\kappa\ \langle\varepsilon\rangle\S\varepsilon^*\ \gamma_2\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\ ]\!]\rho\gamma\kappa\sigma = (\kappa\ \langle\rangle\ \gamma\ \sigma)$$

$$\mathcal{E}[\![\texttt{(begin } \pi^+\texttt{)}]\!]\rho\gamma\kappa\sigma = (\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \gamma\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi]\!]\rho\gamma\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi\ \pi^+]\!]\rho\gamma\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \gamma_1\ \kappa\ \sigma_1)\ \sigma)$$

$$(\sigma_0(\rho_0[\![\texttt{call/cc}]\!])) =$$
$$\textit{in}\textbf{Valeur}(\lambda\ \varepsilon^*\gamma\kappa\sigma.$$
$$\textbf{if}\ 1 = \#\varepsilon^*$$
$$\textbf{then}\ (\ \varepsilon^*\downarrow_1|_{\textbf{Fonction}}$$
$$\langle\textit{in}\textbf{Valeur}(\lambda\ {\varepsilon^*}_1\gamma_1\kappa_1\sigma_1.$$
$$\textbf{if}\ 1 = \#{\varepsilon^*}_1$$
$$\textbf{then}\ (\kappa\ {\varepsilon^*}_1\downarrow_1\ \ \gamma_1\ \sigma_1)$$
$$\textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\quad)\rangle\ \gamma\ \kappa\ \sigma)$$
$$\textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\quad)$$

TABLE 5.5 – Scheme et environnement global

$$\begin{array}{rl} \textbf{EnvironnementLocal} : & \textbf{Identificateur} \rightarrow \textbf{Adresse} + \{\textit{no-such-binding}\} \\ \textbf{EnvironnementGlobal} : & \textbf{Identificateur} \rightarrow \textbf{Adresse} + \{\textit{no-such-global-binding}\} \end{array}$$

Les dénotations de référence et de l'affectation d'une variable sont donc simples : on cherche tout d'abord si la variable est locale puis si elle est globale. Cette recherche produit une adresse qui est alors utilisée pour lire la valeur de la variable. Bien sûr les environnements initiaux sont accordés à ce codage.

$$(\rho_0\ \nu) = \textit{no-such-binding}$$

$$(\gamma_0\ \nu) = \textit{no-such-global-binding}$$

$$\begin{array}{l} \mathcal{E}[\![\nu]\!] = \\ \lambda\rho\gamma\kappa\sigma.\ \textbf{let}\ \alpha = (\rho\ \nu) \\ \qquad\qquad \textbf{in}\ \ \textbf{if}\ \alpha = \textit{no-such-binding} \\ \qquad\qquad\qquad \textbf{then}\ \ \textbf{let}\ \alpha_1 = (\gamma\ \nu) \\ \qquad\qquad\qquad\qquad\ \ \textbf{in}\ \ \textbf{if}\ \alpha_1 = \textit{no-such-global-binding} \\ \qquad\qquad\qquad\qquad\qquad\ \ \textbf{then}\ \textit{wrong}\ \text{“}\texttt{No such variable}\text{”} \\ \qquad\qquad\qquad\qquad\qquad\ \ \textbf{else}\ (\kappa\ (\sigma\ \alpha_1)\ \gamma\ \sigma) \\ \qquad\qquad\qquad\qquad\qquad\ \ \textbf{endif} \\ \qquad\qquad\qquad \textbf{else}\ (\kappa\ (\sigma\ \alpha)\ \gamma\ \sigma) \\ \qquad\qquad\qquad \textbf{endif} \end{array}$$

$$\begin{array}{l} \mathcal{E}[\![\texttt{(set!}\ \nu\ \pi\texttt{)}]\!]\rho\gamma\kappa\sigma = \\ (\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.\ \textbf{let}\ \alpha = (\rho\ \nu) \\ \qquad\qquad \textbf{in}\ \ \textbf{if}\ \alpha = \textit{no-such-binding} \\ \qquad\qquad\qquad \textbf{then}\ \ \textbf{let}\ \alpha_1 = (\gamma_1\ \nu) \\ \qquad\qquad\qquad\qquad\ \ \textbf{in}\ \ \textbf{if}\ \alpha_1 = \textit{no-such-global-binding} \\ \qquad\qquad\qquad\qquad\qquad\ \ \textbf{then}\ \textit{wrong}\ \text{“}\texttt{No such variable}\text{”} \\ \qquad\qquad\qquad\qquad\qquad\ \ \textbf{else}\ (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha_1 \rightarrow \varepsilon]) \\ \qquad\qquad\qquad\qquad\qquad\ \ \textbf{endif} \\ \qquad\qquad\qquad \textbf{else}\ (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha \rightarrow \varepsilon]) \\ \qquad\qquad\qquad \textbf{endif}\quad \sigma) \end{array}$$

Nous supposerons que les définitions de variables globales sont effectuées par l'opérateur spécial `define` et que donc la forme spéciale `(define` ν π`)` définit (ou redéfinit) la variable *globale* ν. Nous ignorons pour l'instant les définitions internes (introduites par des formes `(define ...)` internes) qui ne sont que pure syntaxe et pour cela nommerons `define-global` les définitions externes. Voici donc la dénotation de la définition et l'on notera soit l'effet d'introduction d'une nouvelle variable, soit l'effet modifiant une variable déjà existante.

$$\begin{array}{l} \mathcal{E}[\![\texttt{(define-global}\ \nu\ \pi\texttt{)}]\!]\rho\gamma\kappa\sigma = \\ (\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.\ \textbf{let}\ \alpha = (\gamma\ \nu) \\ \qquad\qquad\qquad \textbf{in}\ \ \textbf{if}\ \alpha = \textit{no-such-global-binding} \\ \qquad\qquad\qquad\qquad\ \ \textbf{then}\ \textit{allocate}\ \sigma_1\ 1\ \ \lambda\sigma_2\alpha^*.(\kappa\ \varepsilon\ \gamma_1[\nu \rightarrow \alpha^*\downarrow_1]\ \sigma_2[\alpha^*\downarrow_1 \rightarrow \varepsilon]) \\ \qquad\qquad\qquad\qquad\ \ \textbf{else}\ (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha \rightarrow \varepsilon]) \\ \qquad\qquad\qquad\qquad\ \ \textbf{endif}\quad \sigma) \end{array}$$

Ces définitions caractérisent le comportement spécifié par le standard Scheme à l'exception de la définition de l'environnement global initial γ_0 qui est laissé dans

l'ombre : il peut ne contenir que les variables du standard, en définir aussi quelques autres additionnelles ou même contenir toutes les variables imaginables dont on pourrait avoir besoin. Dans ce dernier cas, il ne saurait y avoir de variables inexistantes mais il est alors possible de rencontrer des variables non initialisées.

5.7.2 Environnement automatiquement extensible

Certains systèmes Lisp permettent que la première affectation sur une variable libre soit équivalente à la définition d'icelle. Il est simple de modifier la sémantique de l'affectation pour incorporer l'effet d'une définition si la variable n'existait pas.

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(set! } \nu\ \pi\texttt{)}]\!]\rho\gamma\kappa\sigma = \\
(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\ \varepsilon\gamma_1\sigma_1. \\
\qquad\qquad \textbf{let}\ \alpha = (\rho\ \nu) \\
\qquad\qquad \textbf{in}\ \ \textbf{if}\ \alpha = \textit{no-such-binding} \\
\qquad\qquad\qquad \textbf{then}\ \ \textbf{let}\ \alpha_1 = (\gamma_1\ \nu) \\
\qquad\qquad\qquad\qquad \textbf{in}\ \ \textbf{if}\ \alpha_1 = \textit{no-such-global-binding} \\
\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \ \textit{allocate}\ \sigma_1\ 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \lambda\ \sigma_2\alpha^*. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\kappa\ \varepsilon\ \gamma_1[\nu \to \alpha^*\downarrow_1]\ \sigma_2[\alpha^*\downarrow_1 \to \varepsilon]) \\
\qquad\qquad\qquad\qquad\qquad \textbf{else}\ (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha_1 \to \varepsilon]) \\
\qquad\qquad\qquad\qquad\qquad \textbf{endif} \\
\qquad\qquad\qquad \textbf{else}\ (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha \to \varepsilon]) \\
\qquad\qquad\qquad \textbf{endif}\quad \sigma)
\end{array}
$$

L'intérêt d'une telle variante est que la forme `define` n'est plus nécessaire et se confond avec `set!`. Le désavantage est qu'une coquille orthographique sur un nom de variable peut être difficile à trouver puisque ne conduisant pas systématiquement à une erreur à la première utilisation.

5.7.3 Environnement hyperstatique

Au sein d'un environnement global hyperstatique, les fonctions enclosent non seulement l'environnement local de définition mais aussi l'environnement global courant. C'est ce que l'on exprime très simplement en perturbant très légèrement la dénotation de l'abstraction telle qu'elle apparaissait au tableau 5.5.

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{)}\pi^+\texttt{)}]\!]\rho\gamma\kappa\sigma = \\
(\kappa\ \textit{in}\textbf{Valeur}(\lambda\ \varepsilon^*\gamma_1\kappa_1\sigma_1. \\
\qquad\qquad \textbf{if}\ \#\varepsilon^* = \#\nu^* \\
\qquad\qquad \textbf{then}\ \ \textit{allocate}\ \sigma_1\ \#\nu^* \\
\qquad\qquad\qquad\qquad \lambda\ \sigma_2\alpha^*. \\
\qquad\qquad\qquad\qquad\quad (\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \gamma\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*]) \\
\qquad\qquad \textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”} \\
\qquad\qquad \textbf{endif}\quad)\ \gamma\ \sigma)
\end{array}
$$

Observez que cette sémantique est compatible avec l'affectation de Scheme qui ne peut modifier que des variables existantes. En revanche, la définition à la volée de nouvelles variables par affectation prête à confusion. Considérez seulement :

```
(define (weird v)
  (set! a-new-variable v) )
```

L'affectation, qui apparaît à l'intérieur de la fonction `weird`, étend l'environnement global qu'elle a enclos et ce, à chaque invocation, puisque l'environnement global étendu résultant n'est pas mémorisé par la fonction `weird`.

Comme déjà mentionné, l'ennui avec les environnements hyperstatiques est que la définition de fonctions mutuellement récursives non locales ne peut se faire simplement. On pourrait introduire une nouvelle forme de codéfinition (autorisant la définition conjointe d'une multitude de liaisons) mais nous allons plutôt introduire la possibilité de référencer l'environnement global grâce à une nouvelle forme spéciale `(global` ν`)` qui permet *en tout contexte* de référencer la variable globale ν même s'il existe une variable locale nommée ν. La codéfinition des fonctions `odd?` et `even?` s'écrira donc :

```
(letrec ((odd? (lambda (n) (if (= n 0) #f (even? (- n 1)))))
         (even? (lambda (n) (if (= n 0) #t (odd? (- n 1))))) )
  (define-global odd? odd?)
  (define-global even? even?) )
```

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(global } \nu\texttt{)}]\!]\rho\gamma\kappa\sigma = \\
\quad \textbf{let } \alpha = (\gamma\ \nu) \\
\quad \textbf{in } \ \textbf{if } \alpha = \textit{no-such-global-binding} \\
\qquad\quad \textbf{then } \textit{wrong } \text{“}\texttt{No such variable}\text{”} \\
\qquad\quad \textbf{else } (\kappa\ (\sigma\ \alpha)\ \gamma\ \sigma) \\
\qquad\quad \textbf{endif}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(define-global } \nu\ \pi\texttt{)}]\!]\rho\gamma\kappa\sigma = \\
(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.\ \textbf{let } \alpha = (\gamma\ \nu) \\
\qquad\qquad\qquad\qquad \textbf{in } \ \textbf{if } \alpha = \textit{no-such-global-binding} \\
\qquad\qquad\qquad\qquad\quad \textbf{then } \textit{allocate } \sigma_1\ 1\ \ \lambda\sigma_2\alpha^*.(\kappa\ \varepsilon\ \gamma_1[\nu \to \alpha^*\downarrow_1]\ \sigma_2[\alpha^*\downarrow_1 \to \varepsilon]) \\
\qquad\qquad\qquad\qquad\quad \textbf{else } (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha \to \varepsilon]) \\
\qquad\qquad\qquad\qquad\quad \textbf{endif}\quad \sigma)
\end{array}
$$

On le voit encore sur ces précédents exemples, la sémantique dénotationnelle permet de spécifier fort élégamment les divers environnements en jeu dans un langage. On peut bien sûr combiner les différents environnements ici esquissés pour offrir comme COMMON LISP de multiples espaces de noms avec toute leur spécificité.

5.8 Les dessous de ce chapitre

Le plus grand but de ce chapitre était de démythifier la sémantique dénotationnelle. Nous avons pris un parti non formel voire sacrilège mais qui nous semble le plus propice à stimuler l'emploi de la sémantique dénotationnelle. L'enrichissement progressif des traits linguistiques que spécifiaient nos divers interprètes et l'appauvrissement non moins progressif du langage de description nous ont permis insensiblement de parvenir à la dénotation de ce chapitre, ou peut-être devrions-nous dire à l'interprète dénotationnel. Car si Scheme et le λ-calcul ne sont point semblables ; ils ne sont point si dissemblables et il est généralement possible d'obtenir une dénotation

exécutable commode pour corriger les erreurs qui pourraient s'être faufilées au sein de ces équations sémantiques. Ce caractère exécutable est très important : lui seul permet de fonder que le langage décrit se comporte bien comme le souhaite son concepteur : il peut l'exercer, l'expérimenter, le tester afin de s'assurer de la conformité des ses équations à ses rêves. Que la dénotation soit immédiatement exécutable telle qu'elle apparaît dans les équations permet aussi de s'affranchir d'une phase d'implantation des équations dont on devrait en toute rigueur prouver la non-distorsion.

S'astreindre à écrire un interprète dénotationnel est donc rassurant et ne fait pas perdre les bénéfices du λ-calcul mais ajoute toutefois une contrainte supplémentaire : l'interprète dénotationnel doit être exécutable en appel par valeur, la « mauvaise » méthode d'évaluation qu'emploie Scheme. C'est souvent possible : à preuve, toutes les dénotations de ce chapitre correspondent à du code écrit en Scheme et ont été passé à travers une moulinette (LiSP2TEX) qui les imprime en grec [Que93d]. Imaginez un peu ce que serait d'écrire la dénotation de l'application préservant le caractère désordonné de l'évaluation sans passer par un langage applicatif dans lequel on puisse tester ces fonctions !

Voici à titre d'exemple la dénotation de l'abstraction simple (sans arité variable) que l'on comparera à son image hellénistique en page 152 :

```
(define ((meaning-abstraction n* e+) r k s)
  (k (inValue (lambda (v* k1 s1)
                (if (= (length v*) (length n*))
                    (allocate s1 (length n*)
                              (lambda (s2 a*)
                                ((meaning*-sequence e+)
                                 (extend* r n* a*)
                                 k1
                                 (extend* s2 a* v*) ) ) )
                    (wrong "Incorrect arity") ) ))
      s ) )
```

Nous avons utilisé ici la forme désuette `define` qui autorisait un premier argument en situation d'appel. La forme `(define (`ν `.` *variables*`)` π^*`)` est récursivement équivalente à `(define` ν `(lambda` *variables* π^*`))`, où ν peut encore être une forme d'appel.

Les dénotations sont énoncées cas par cas, ce qui est automatiquement réalisé par une fonction appropriée qui réalise l'analyse syntaxique et dont la structure vous est maintenant familière :

```
(define (meaning e)
  (if (atom? e)
      (if (symbol? e) (meaning-reference e)
                      (meaning-quotation e) )
      (case (car e)
        ((quote)  (meaning-quotation (cadr e)))
        ((lambda) (meaning-abstraction (cadr e) (cddr e)))
        ((if)     (meaning-alternative
                    (cadr e) (caddr e) (cadddr e) ))
        ((begin)  (meaning-sequence (cdr e)))
        ((set!)   (meaning-assignment (cadr e) (caddr e)))
        (else     (meaning-application (car e) (cdr e))) ) ) )

(define (meaning-abstraction n* e+)
  (let ((m (meaning*-sequence e+))
```

```
      (arity (length n*)) )
  (lambda (r k s)
    (k (inValue (lambda (v* k1 s1)
                  (if (= (length v*) arity)
                      (allocate s1 arity
                                (lambda (s2 a*)
                                  (m (extend* r n* a*)
                                     k1
                                     (extend* s2 a* v*) ) ) )
                      (wrong "Incorrect arity") ) ))
         s ) ) ) )
```

5.9 λ-calcul et Scheme

Le souci d'exécutabilité des dénotations pose le problème des relations entre le langage Scheme et le λ-calcul. On ne considérera bien sûr que le sous-ensemble dit pur de Scheme sans affectation, ni effet de bord. La différence réside dans la stratégie d'évaluation. Il n'y en a pas de fixée pour le λ-calcul, attention de ne pas en prendre une mauvaise. En revanche, il y en a une de spécifiée pour Scheme et elle est différente : Scheme utilise l'appel par valeur, c'est-à-dire que les arguments d'une application sont évalués avant d'être soumis à la fonction ; de plus, Scheme n'évalue pas sous les `lambda`.

On peut simuler l'appel par nom en Scheme en utilisant des *suspensions* (ou *thunk*). Une suspension est une fonction sans variable que l'on nomme aussi une « promesse » ou un « glaçon » suivant les terminologies. Une syntaxe existe en Scheme, `delay`, qui permet de construire une promesse, c'est-à-dire un objet représentant un certain calcul que l'on pourra déclencher (ou `force`r) le moment venu grâce à la fonction `force`. On peut définir `delay` et `force` comme :

```
(define-syntax delay
  (syntax-rules ()
    ((delay expression) (lambda () expression)) ) )
(define (force promise) (promise))
```

La forme `delay` enclôt une expression dans son contexte lexical en une suspension que l'on peut invoquer à la demande par `force`. On peut simuler l'appel par valeur avec ce mécanisme à condition de transformer le programme comme suit : toute application `(`f a `...`z`)` est réécrite en `(`f `(delay` a`) ...(delay` z`))` et toute variable apparaissant dans le corps d'une fonction est explicitement dégelée au moyen de `force`. Pour donner un exemple, le calcul naguère problématique pour Scheme était :

```
(((lambda (x) (lambda (y) y))                    ; ((λx.λy.y (ωω)) z)
  ((lambda (x) (x x)) (lambda (x) (x x))) )
 z )
```

Ce calcul se réécrit en ce qui suit qui maintenant suspend le calcul sans fin sans l'effectuer et retourne ce que vaut la variable libre `z`.

```
(((lambda (x) (lambda (y) (force y)))
  (delay ((lambda (x) ((force x) (delay (force x))))
          (lambda (x) ((force x) (delay (force x))))) ))
 (delay z) )
```

Quoique correcte [DH92], cette réécriture introduit une certaine inefficacité. Mise à part la construction de double glaçons comme `(delay (force x))` qui est en fait

équivalente au déjà glacé `x`, chaque fois que l'on a besoin de la valeur d'une variable, il faut forcer son calcul alors qu'il suffirait de ne le faire qu'une seule fois et de mémoriser le résultat obtenu. Cette technique se nomme appel par nécessité et correspond à modifier `delay` pour mémoriser la valeur à laquelle il conduit. Fortunément, Scheme étant un excellent langage d'implantation de langages, nous allons pouvoir programmer cela en appelant à la rescousse une affectation.

```
(define-syntax memo-delay
  (syntax-rules ()
    ((memo-delay expression)
(let ((already-computed? #f)
      (value 'wait) )
  (lambda ()
    (if (not already-computed?)
        (begin
          (set! value expression)
          (set! already-computed? #t) ) )
    value ) ) ) ) )
```

On peut bien sûr ne pas effectuer ces transformations à la main et concevoir des macros pour ce faire [DFH86]. On peut si l'on veut vraiment être efficace procéder à une analyse de nécessité (pour *strictness analysis*) pour déterminer les cas où les variables seront toujours forcées, ce qui permet d'éviter de construire des promesses pour elles [BHY87]. On peut enfin adopter des codages astucieux pour les glaçons afin d'éliminer le coût à l'exécution du test `(unless already-computed? ...)`. Les promesses permettent en Scheme de faire comme si l'on travaillait dans un langage paresseux où les transformations énoncées plus haut seraient automatiquement effectuées. Toutefois ce style de programmation pose des problèmes de mise au point car le processus de calcul est très distribué ; d'autre part, ce style se marie très mal avec les affectations et les continuations. Que l'on compare seulement ces deux programmes [KW90, Mor92] ne différant que d'un seul `delay` mais calculant des résultats fort différents.

```
(pair? (call/cc (lambda (k) (list (k 33)))))
(pair? (call/cc (lambda (k) (list (delay (k 33))))))
```

5.9.1 Passage de continuations

Une rapide considération des dénotations permet de se rendre compte qu'elles sont écrites dans un style que l'on nomme *programmation par continuations* ou CPS (pour *Continuation Passing Style*). Sachant que Scheme tout entier (y compris `call/cc`) peut être dénoté par le λ-calcul qui peut lui-même être vu comme un certain sous-ensemble de Scheme excluant `call/cc`, on peut valablement s'interroger sur la possibilité de transformer des programmes automatiquement (de Scheme vers Scheme) pour faire disparaître `call/cc`.

L'intérêt de ce style est donc de faire apparaître explicitement les continuations. Sachant que ces mêmes continuations peuvent être représentées par des formes `lambda`, la question est *peut-on se passer de la primitive* `call/cc` *?* La réponse est oui et l'on peut donc développer une transformation de programmes qui convertit un programme avec des `call/cc` en un autre équivalent qui ne les contient pas. Quelques compilateurs utilisent cette dernière forme comme langage quasi-intermédiaire [App92a]

car elle ne contient plus que des constructions syntaxiques simplissimes. D'autres ne préfèrent pas car cette forme en CPS n'est plus lisible et a fixé l'ordre d'évaluation trop tôt. C'est de toute façon équivalent comme il est montré en [SF92]. La version de cette transformation détaillée plus bas s'inspire fortement de [DF90]. Une version différente sera utilisée au paragraphe 10.10.2. **[☞ p. 384]**

Décrivons la technique. Toute fonction est supposée être appelée avec un argument supplémentaire qui est la continuation [9]. Ainsi $_k$`(foo bar ...hux)` sera transformé en `(foo` k `bar ...hux)`. Il faut donc avoir en tout point une représentation de la continuation. Il faut donc savoir aussi traiter les formes spéciales et donc procéder à une analyse du texte à traduire comme l'ont fait tous les interprètes précédents. Comme son équivalent dénotationnel, la continuation attend une valeur et retourne le résultat final du calcul. La continuation enclôt donc le reste des calculs à effectuer.

```
(define (cps e)
  (if (pair? e)
      (case (car e)
        ((quote)  (cps-quote (cadr e)))
        ((if)     (cps-if (cadr e) (caddr e) (cadddr e)))
        ((begin)  (cps-begin (cdr e)))
        ((set!)   (cps-set! (cadr e) (caddr e)))
        ((lambda) (cps-abstraction (cadr e) (caddr e)))
        (else     (cps-application e)) )
      (lambda (k) (k `,e)) ) )
```

La transformation par passage de continuation prend un programme et retourne une fermeture qui convertit un programme en un programme. En d'autres termes, le type de la fonction `cps` est :

```
Programme → (( Programme → Programme ) → Programme )
```

La citation est aisée à traiter.

```
(define (cps-quote data)
  (lambda (k)
    (k `(quote ,data)) ) )
```

L'affectation devient :

```
(define (cps-set! variable form)
  (lambda (k)
    ((cps form)
     (lambda (a)
       (k `(set! ,variable ,a)) ) ) ) )
```

Elle convertit la forme dont la valeur servira à affecter la variable et l'insère à sa place dans l'affectation qui doit être engendrée.

L'alternative se traite similairement :

```
(define (cps-if bool form1 form2)
  (lambda (k)
    ((cps bool)
     (lambda (b)
       `(if ,b ,((cps form1) k)
               ,((cps form2) k) ) ) ) ) )
```

et la séquence pareillement :

```
(define (cps-begin e)
  (if (pair? e)
```

9. On le placera en premier argument ce qui simplifie la gestion des fonctions d'arité variable.

```
(if (pair? (cdr e))
    (let ((void (gensym "void")))
      (lambda (k)
        ((cps (car e))
         (lambda (a)
           ((cps-begin (cdr e))
            (lambda (b)
              (k `((lambda (,void) ,b) ,a)) ) ) ) ) ) ) )
    (cps (car e)) )
(cps '()) ) )
```

On notera au passage que les formes `begin` sont supprimées au profit d'un codage par fermetures.

La partie complexe est le traitement de la forme `lambda` qui doit être convertie en une nouvelle fonction prenant une variable supplémentaire ainsi que l'application fonctionnelle qui doit fournir à la fonction invoquée la continuation comme argument supplémentaire. Une légère amélioration a été apportée lorsque la fonction appelée est triviale, c'est-à-dire lorsqu'elle effectue un calcul simple, court et qui termine toujours. Ces fonctions ont été regroupées dans la liste valeur de `primitives` [10].

```
(define (cps-application e)
  (lambda (k)
    (if (memq (car e) primitives)
        ((cps-terms (cdr e))
         (lambda (t*)
           (k `(,(car e) ,@t*)) ) )
        ((cps-terms e)
         (lambda (t*)
           (let ((d (gensym)))
             `(,(car t*) (lambda (,d) ,(k d))
                         . ,(cdr t*) ) ) ) ) ) ) )
(define primitives '( cons car cdr list * + - = pair? eq? ))
(define (cps-terms e*)
  (if (pair? e*)
      (lambda (k)
        ((cps (car e*))
         (lambda (a)
           ((cps-terms (cdr e*))
            (lambda (a*)
              (k (cons a a*)) ) ) ) ) )
      (lambda (k) (k '())) ) )
(define (cps-abstraction variables body)
  (lambda (k)
    (k (let ((c (gensym "cont")))
         `(lambda (,c . ,variables)
            ,((cps body)
              (lambda (a) `(,c ,a)) ) ) )) ) )
```

La transformation est maintenant complète et peut être expérimentée sur la factorielle :

```
(set! fact (lambda (n)
                   (if (= n 1) 1
                       (* n (fact (- n 1))) ) ))
→ (set! fact
```

10. Le traitement correct de ces appels à des fonction primitives prédéfinies sera abordé au paragraphe 6.1.8.

```
(lambda (cont112 n)
  (if (= n 1)
      (cont112 1)
      (fact (lambda (g113) (cont112 (* n g113)))
            (- n 1) ) ) ) )
```

On retrouve automatiquement ce que l'on avait écrit plus haut à la main. On le constate, il ne reste aucun calcul compliqué après transformation, seulement des applications triviales comme des comparaisons ou opérations arithmétiques simples ou des applications avec continuation. Le reste n'est plus composé que de variables ou de fermetures.

Dans ce monde, une forme $_k$`(call/cc f)` serait devenue `(call/cc` k `f)`. La fonction `call/cc` n'est donc plus que `(lambda (k f) (f k k))` et peut être simplifiée directement pour ne plus apparaître. Il faut cependant lier la variable globale `call/cc` à `(lambda (k f) (f k k))` de manière à pouvoir, par exemple, écrire : `(procedure? (apply call/cc (list call/cc)))`.

Une vertu d'un programme écrit en style CPS est que, comme on l'a totalement séquentialisé en indiquant explicitement quand les termes doivent être évalués et à qui ils doivent retourner leur valeur, il est désormais insensible à la stratégie d'évaluation. On peut évaluer en appel par valeur ou par nom, on obtient les mêmes résultats. En combinant donc promesses et CPS, on peut amenuiser les différences entre Scheme et λ-calcul [DH92].

5.9.2 Environnement dynamique

La précédente section avait montré comment la dénotation de `call/cc` permettait d'inventer une transformation de programmes éliminant ce même `call/cc`. La dénotation de l'environnement dynamique permet d'envisager d'appliquer la même technique pour s'affranchir complètement des formes spéciales `dynamic` et `dynamic-let`. Pour cela, il suffit d'introduire explicitement l'environnement dynamique partout où il est nécessaire.

Supposons disposer d'un identificateur n'intervenant nul part ailleurs, disons δ. L'environnement dynamique sera, en tout point, la valeur de δ et sera représenté par une fonction transformant des symboles (nommant les variables dynamiques) en des valeurs. On supposera aussi que la fonction `update` qui permet d'étendre fonctionnellement un environnement **[☞ p. 123]** sera partout disponible et non masquable. La transformation, nommée $\mathcal{D}$ (et l'utilitaire $\mathcal{D}^*$), apparaît dans le tableau 5.6.

$$
\begin{array}{lll}
\mathcal{D}^*[\![]\!] & \rightarrow & \\
\mathcal{D}^*[\![\pi\ \pi^*]\!] & \rightarrow & \mathcal{D}[\![\pi]\!]\ \mathcal{D}^*[\![\pi^*]\!] \\
\mathcal{D}[\![\texttt{(if}\ \pi_0\ \pi_1\ \pi_2\texttt{)}]\!] & \rightarrow & \texttt{(if}\ \mathcal{D}[\![\pi_0]\!]\ \mathcal{D}[\![\pi_1]\!]\ \mathcal{D}[\![\pi_2]\!]\texttt{)} \\
\mathcal{D}[\![\texttt{(begin}\ \pi^*\texttt{)}]\!] & \rightarrow & \texttt{(begin}\ \mathcal{D}^*[\![\pi^*]\!]\texttt{)} \\
\mathcal{D}[\![\texttt{(}\pi\ \pi^*\texttt{)}]\!] & \rightarrow & \texttt{(}\mathcal{D}[\![\pi]\!]\ \delta\ \mathcal{D}^*[\![\pi^*]\!]\texttt{)} \\
\mathcal{D}[\![\texttt{(lambda (}\nu^*\texttt{)}\ \pi^*\texttt{)}]\!] & \rightarrow & \texttt{(lambda (}\delta\ \nu^*\texttt{)}\ \mathcal{D}^*[\![\pi^*]\!]\texttt{)} \\
\mathcal{D}[\![\texttt{(dynamic}\ \nu\texttt{)}]\!] & \rightarrow & \texttt{(}\delta\ \texttt{(quote}\ \nu\texttt{))} \\
\mathcal{D}[\![\texttt{(dynamic-let (}\nu\ \pi\texttt{)}\ \pi^+\texttt{)}]\!] & \rightarrow & \texttt{(let ((}\delta\ \texttt{(update}\ \delta\ \texttt{(quote}\ \nu\texttt{)}\ \pi\texttt{)))}\ \mathcal{D}^*[\![\pi^+]\!]\texttt{)}
\end{array}
$$

TABLE 5.6 – Transformation supprimant l'environnement dynamique

Cette transformation permet de simuler simplement les liaisons dynamiques si l'on

n'en dispose point. L'ultime détail est de spécifier l'environnement dynamique initial ; le programme π à transformer le sera donc en :

```
(let ((δ (lambda (n) (error "No such dynamic variable" n)))) 𝒟⟦π⟧)
```

On peut donc le mesurer : de nombreuses dénotations conduisent à des transformations de programme visant à éliminer les formes mêmes qu'elles définissaient.

5.10 Conclusions

Ce chapitre constitue la culmination d'une longue suite d'interprètes visant à définir un langage de la classe de Scheme de plus en plus précisément et avec des moyens linguistiques de plus en plus restreints. La sémantique dénotationnelle telle que nous en avons donné des exemples tout au long de ce chapitre est d'une concision remarquable en ce qui concerne la description du noyau des langages fontionnels. Elle s'avère souvent mal adaptée et d'une complexité indue dès qu'il s'agit de décrire tout un langage dans ses moindres détails. La présentation de Scheme que nous avons ici faite excluait la comparaison de fonctions, la dénotation des constantes, toutes sortes de caractéristiques grevant l'ensemble de la définition de détails qui ne trouvent que très ponctuellement leur intérêt. La sémantique dénotationnelle est à son mieux lorsqu'il s'agit de décrire les grandes lignes d'un langage mais devient assez soporifique s'il s'agit de tout décrire.

L'éventail de ce que l'on peut décrire dénotationnellement est assez vaste. On peut décrire des caractéristiques introduisant le parallélisme avec la technique des suspensions ou pas de calcul [Que90c] ; on peut décrire des effets de distribution de données [Que92b]. Il y a aussi des limitations [McD93] ; il n'est, par exemple, pas facile de décrire dénotationnellement ce qu'est une inférence de types, c'est d'ailleurs une des raisons de l'émergence de la sémantique naturelle [Kah87].

5.11 Exercices

Exercice 5.1 : Voici une autre manière d'écrire la dénotation de l'application fonctionnelle. Montrez qu'elle est équivalente à celle figurant au paragraphe 5.2.6.

$$\mathcal{E}[\![(\pi\ \pi^*)]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varphi\sigma_1.(\overline{\mathcal{E}}[\![\pi^*]\!]\ \langle\rangle\ \rho\ \lambda\varepsilon^*\sigma_2.(\varphi\ |_{\textbf{Fonction}}\ \varepsilon^*\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\overline{\mathcal{E}}[\![\,]\!]\varepsilon^*\rho\kappa\sigma = (\kappa\ (\mathit{reverse}\ \varepsilon^*)\ \sigma)$$

$$\overline{\mathcal{E}}[\![\pi\ \pi^*]\!]\varepsilon^*\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\overline{\mathcal{E}}[\![\pi^*]\!]\ \langle\varepsilon\rangle\S\varepsilon^*\ \rho\ \kappa\ \sigma_1)\ \sigma)$$

Exercice 5.2 : Définir des fonctions récursives au sein du λ-calcul tel que présenté au paragraphe 5.3, est peu efficace. On peut enrichir ce langage de la forme spéciale `label` comme en Lisp 1.5 ; la forme `(label` ν `(lambda ...))` est, en termes schemiens, équivalente à :

```
(letrec ((ν (lambda ...))) ν)
```

Définissez la sémantique cet opérateur `label`.

Exercice 5.3 : Modifiez la dénotation de la forme spéciale `dynamic` afin que, lorsque la variable dynamique n'est pas trouvée dans l'environnement dynamique, sa valeur dans l'environnement global soit rendue.

Exercice 5.4 : Écrivez une macro, permettant de simuler un ordre d'évaluation non prescrit, dans une implantation de Scheme évaluant de la gauche vers la droite. On supposera bénéficier d'une fonction unaire `random-permutation` prenant un entier n et retournant une permutation aléatoire de $1 \ldots n$.

5.12 Lectures recommandées

Les incontournables sont [Sto77] et [Sch86] ; l'un comme l'autre sont des mines de renseignements sur la sémantique dénotationnelle et présentent des exemples de dénotations de langage. Les fervents du λ-calcul se tourneront vers [Bar84].

6

Interprétation rapide

Au chapitre précédent était apparu un interprète dénotationnel qui alliait à une extrême précision une remarquable lenteur. Ce chapitre analyse les raisons de cette lenteur et propose quelques nouveaux interprètes palliant ce défaut par un prétraitement des programmes, bref une compilation rudimentaire. Seront analysés successivement : la représentation des environnements lexicaux, le protocole d'appel des fonctions et la réification des continuations. Le prétraitement distingue puis élimine les calculs qu'il juge statiques et produit un résultat ne comportant plus que les seules opérations qu'il estime nécessaires à l'exécution : des combinateurs spécialisés sont introduits à cette occasion et jouent le rôle d'une sorte de langage intermédiaire voire de jeu d'instructions d'une hypothétique machine virtuelle.

L'interprète dénotationnel du chapitre précédent avait représenté la culmination d'une longue série d'interprètes visant à une précision inexorablement croissante. Nous allons nous efforcer de corriger cette lenteur exécrable. Suivant toujours notre technique de modification incrémentielle et, surtout, parce que l'interprète dénotationnel précédent est l'étalon linguistique auquel nous devons nous conformer, nous opérerons en présentant successivement trois interprètes relâchant peu à peu, au profit des us et coutumes des implanteurs, quelques-uns des soucis descriptifs premiers.

6.1 Interprète rapide

Comme maintenant nous nous intéressons à produire un interprète efficace, nous allons supposer que le langage d'implantation contient un minimum de concepts et notamment une mémoire. Nous allons donc supprimer ce que nous avions ajouté au chapitre 4, **[☞ p. 105]** à savoir l'explicitation de la mémoire sauf en ce qui concerne les liaisons des variables. Cela revient à déléguer à `cons` la gestion des paires pointées, à `vector` celle des vecteurs, à `lambda` celles des fermetures et ainsi de suite. La mémoire ne sera donc plus un gigantesque fatras où se trouvaient pêle-mêle les arguments des fonctions, les structures de données ou, même, les étiquettes des fermetures. La gestion de ces différentes données gagne à être spécialisée [MB93], aussi la mémoire ne contiendra-t-elle plus que les seules liaisons ou blocs d'activation.

6.1.1 Migration des dénotations

La plus grande source d'inefficacité de l'interprète dénotationnel est que les programmes n'arrêtent pas d'être dénotés, redénotés, re-redénotés, etc. La parade est simple et consiste à faire migrer les calculs en des positions où ils sont effectués aussitôt que faisables [Deu89]. Cette technique de migration de code (pour λ-*hoisting* [Tak88] ou λ-*drifting* [Roz92]) ne peut être sans précaution appliquée à Scheme en raison des effets de bord qui pourraient être effectués au mauvais moment et/ou un mauvais nombre de fois. Un problème lié est que la migration de code peut aussi modifier la terminaison du programme entier si le calcul qui est migré est un calcul bouclant sans fin : l'expression `(lambda (x) (ω ω))`, où `(ω ω)` est un calcul sans fin, n'est pas équivalente à `(let ((tmp (ω ω))) (lambda (x) tmp))`, même si la variable `x` n'apparaît pas dans la forme définissant `tmp`. Ces raisons n'en sont pas pour les dénotations car, d'une part, elles sont exemptes d'effets de bord et, d'autre part, les dénotations qui sont migrées terminent toujours puisque les dénotations sont compositionnelles et que les programmes traités sont des arbres finis [1].

Pour donner un exemple de migration, voici une nouvelle version de l'abstraction. On peut y observer que la dénotation du corps de l'abstraction `(meaning*-sequence e+)` n'est calculée qu'une seule fois ainsi que le nombre de variables de l'abstraction : `(length n*)`.

```
(define (meaning-abstraction n* e+)
  (let ((m (meaning*-sequence e+))
        (arity (length n*)) )
    (lambda (r k s)
      (k (inValue (lambda (v* k1 s1)
                    (if (= (length v*) arity)
                        (allocate s1 arity
                                  (lambda (s2 a*)
                                    (m (extend* r n* a*)
                                       k1
                                       (extend* s2 a* v*) ) ) )
                        (wrong "Incorrect arity") ) ))
           s ) ) ) ) )
```

6.1.2 Bloc d'activation

Diminuer la consommation en mémoire améliore aussi, dans une large mesure, la vitesse d'interprétation. Une cause importante de cette consommation est le protocole d'appel des fonctions tel qu'il est exprimé dans l'interprète dénotationnel. Invoquer une fonction nécessite de lui fournir des valeurs qui deviendront ses arguments, c'est-à-dire les valeurs de ses variables. L'interprète dénotationnel fournissait ces valeurs sous forme d'une liste ; ces mêmes valeurs étaient ensuite ajoutées (par concaténation) à l'environnement, lui-même représenté par une liste. Cette boulimie de listes se justifie pour une spécification ou un prototype par la facilité d'écriture qu'elle confère mais disqualifie toute recherche sérieuse de quelque vélocité.

---

1. Voir toutefois [Que92a] pour des programmes non représentés par des arbres finis mais par des graphes finis éventuellement cycliques.

Si certaines de ces allocations sont superfétatoires, d'autres ne peuvent être évitées. S'appuyer sur ces dernières pour réaliser aussi l'effet des premières fait partie des réflexes de tout implanteur. Comme fournir des valeurs à une fonction est l'acte même de l'invocation, on ne peut s'en dispenser et ces valeurs doivent bien apparaître quelque part : dans des registres, une pile ou encore un *bloc d'activation* (pour *activation record*). Un bloc d'activation sera représenté par un objet qui contiendra les valeurs fournies à une fonction. On peut donc poser, temporairement :

TEMPORARY
```
(define-class activation-frame Object
  ( (* argument) ) )
```

Cette définition fait appel à une caractéristique non encore mentionnée de notre système d'objets qui est la possibilité de spécifier que certains champs sont indexés c'est-à-dire qu'ils ne sont pas réduits à une seule valeur mais en contiennent une séquence ordonnée dont la taille est déterminée lors de leur création. Voir la section 11.2 pour plus de détails.

Cette représentation est plus intéressante que la liste dès que le nombre d'arguments est grosso modo supérieur à deux. Elle autorise de plus un accès direct aux arguments plutôt qu'un accès séquentiel linéaire comme dans le cas des listes. Un bloc d'activation doit permettre d'étendre un environnement lexical et l'on peut, bien sûr, faire en sorte qu'un environnement soit une liste de blocs d'activation (Figure 6.1).

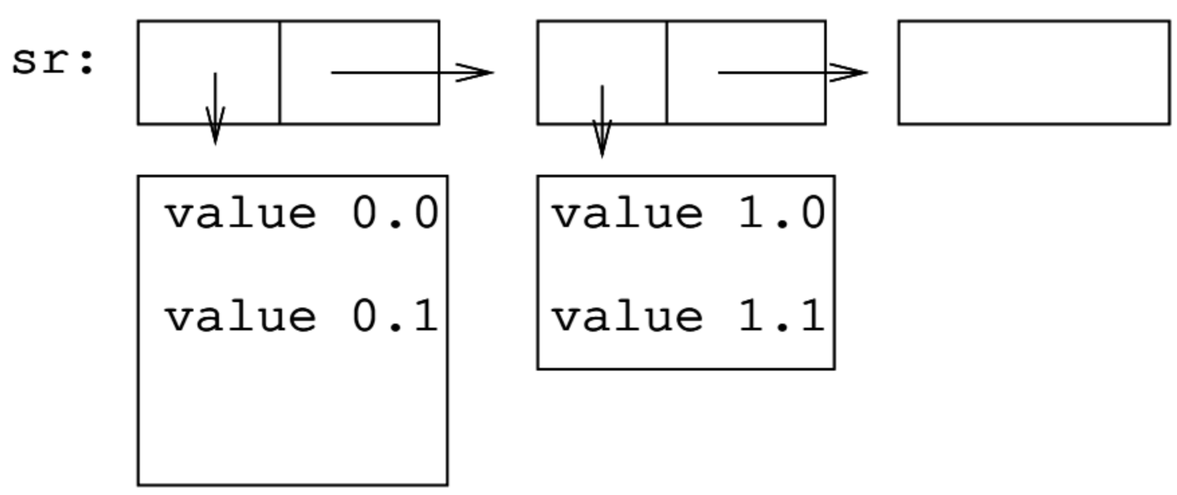


FIGURE 6.1 – Environnement et listes de blocs d'activation

On peut, avec encore plus de bonheur, réserver un champ supplémentaire dans les blocs d'activation pour les chaîner. Comme le temps nécessaire pour l'allocation de blocs d'activation ne dépend guère de leur taille (si l'on excepte les problèmes d'initialisation des champs vis-à-vis du gestionnaire de mémoire), on troque deux allocations pour une seule plus grande d'un pointeur. On adoptera donc la définition suivante pour les blocs d'activation où l'on a inséré le champ de chaînage en premier, afin de pouvoir le suivre ou le modifier au moyen d'un décalage, ce qui ne perturbe aucunement l'accès aux arguments qui suivent (Figure 6.2) :

```
(define-class environment Object
  ( next ) )
(define-class activation-frame environment
  ( (* argument) ) )
```

Les valeurs présentes dans les environnements lexicaux sont donc agglomérées en une chaîne de blocs d'activation. Ce chaînage est réalisé par la fonction `sr-extend*` :

```
(define (sr-extend* sr v*)
  (set-environment-next! v* sr)
  v* )
```

Les blocs d'activation sont modifiés physiquement ; ils ne peuvent donc être réemployés puisque l'invocation fonctionnelle doit allouer des liaisons fraîches.

Si l'on se remémore la représentation des environnements lexicaux dans l'interprète dénotationnel, on se souviendra que celui-ci était composé de deux parties distinctes : ρ et σ. L'environnement ρ associait à un nom de variable une adresse qui permettait de déterminer sa valeur dans la mémoire σ. Ici nous n'avons conservé de la mémoire que la seule gestion des blocs d'activation. Toute valeur peut être retrouvée dans la chaîne des blocs d'activation par une « adresse » composée de deux nombres : le premier indiquant dans quel bloc d'activation chercher, le second fixant alors l'index de l'argument cherché. C'est ce qu'illustre la figure 6.2 et les deux fonctions de lecture et d'écriture suivantes :

```
(define (deep-fetch sr i j)
  (if (= i 0)
      (activation-frame-argument sr j)
      (deep-fetch (environment-next sr) (- i 1) j) ) )
(define (deep-update! sr i j v)
  (if (= i 0)
      (set-activation-frame-argument! sr j v)
      (deep-update! (environment-next sr) (- i 1) j v) ) )
```

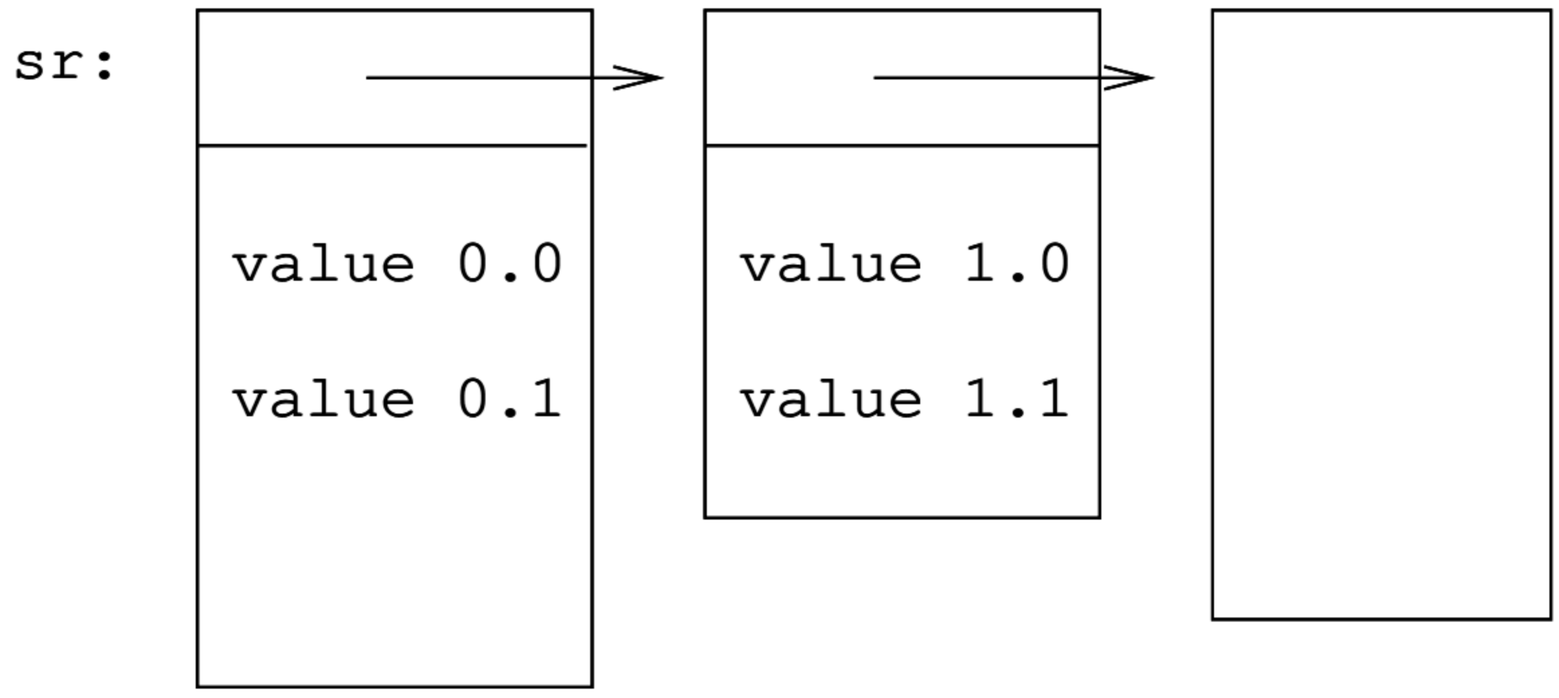


FIGURE 6.2 – Environnement et blocs d'activation chaînés

L'environnement associe donc à un identificateur une adresse permettant de retrouver la valeur associée. La mémoire, que l'on veut faire disparaître du nouvel interprète, ne subsiste donc plus que pour la seule représentation des blocs d'activation. En conséquence, nous nommerons toujours `r` les environnements lexicaux mais `sr` (pour `s`*tore* associé à l'environnement `r`) la représentation à l'exécution des valeurs de cet environnement. Quoique cette dernière soit représentée par une suite chaînée de blocs d'activation, nous adopterons pour l'environnement une représentation « en côtelettes », c'est-à-dire la liste des listes de variables des abstractions. **[☞ Ex. 1.3]** L'extension de l'environnement s'effectuera par `r-extend*` (à ne pas confondre avec `sr-extend*`) tandis que la recherche des coordonnées désignant la valeur sera le lot du semi-prédicat `local-variable?` :

```
(define (r-extend* r n*)
```

```
  (cons n* r) )
(define (local-variable? r i n)
  (and (pair? r)
       (let scan ((names (car r))
                  (j 0) )
         (cond ((pair? names)
                (if (eq? n (car names))
                    `(local ,i . ,j)
                    (scan (cdr names) (+ 1 j)) ) )
               ((null? names)
                (local-variable? (cdr r) (+ i 1) n) )
               ((eq? n names) `(local ,i . ,j)) ) ) ) )
```

L'intérêt de ce schéma est que la forte structure des blocs d'activation permet que l'adresse associée à un identificateur ne soit plus une adresse absolue mais un couple de nombres qui s'interprète relativement à la chaîne des blocs d'activation. En conséquence, les calculs d'adresses deviennent des calculs statiques qui peuvent donc migrer lors du prétraitement des expressions à évaluer. Cette scission de l'environnement lexical en une partie statique et une partie dynamique n'est pas une propriété dépendant des représentations que nous venons de choisir ; c'est une propriété plus profonde seulement révélée. Avec la représentation élue pour les environnements de l'interprète dénotationnel, il était déjà possible de prédire, en tout point, la « position » (position comptée en nombre de comparaisons par `eq?`) d'une variable quelconque au sein de l'environnement puisque celui-ci n'était autre qu'une suite chaînée de fermetures de corps `(lambda (u) (if (eq? u` x`)` y `(`f `u)))`.

6.1.3 L'interprète : le début

Nous en savons assez maintenant pour exposer l'ossature de l'interprète et de quelques-unes de ses formes spéciales. Comme de bien entendu, nous commençons par la fonction `meaning` qui aura maintenant la signature suivante :

$$\texttt{meaning} : \underbrace{\textbf{Programme} \times \textbf{Environnement}}_{statique} \rightarrow (\underbrace{\textbf{Bloc-dActivation} \times \textbf{Continuation} \rightarrow \textbf{Valeur}}_{dynamique})$$

Cette signature montre bien la scission de l'environnement en ses composantes statique et dynamique : **Environnement** et **Bloc-dActivation**. Le résultat du prétraitement que subit le programme est une fonction binaire attendant une liste chaînée de blocs d'activation (une mémoire) et une continuation pour pouvoir calculer une valeur. C'est une représentation non standard des programmes, assez éloignée de la S-expression originelle quoique, dans le fond, structurellement comparable.

Pour simplifier l'analyse syntaxique, nous supposerons comme toujours que les expressions soumises sont des programmes syntaxiquement respectables. Enfin, avant de vous livrer en pâture les fonctions qui vont suivre, nous mentionnons nos règles de dénomination des variables :

`e...`	expression, forme
`r...`	environnement
`sr...`, `v*...`	bloc d'activation
`v...`	valeur (entier, paire, fermeture, etc.)
`k...`	continuation
`f...`	fonction
`n...`	identificateur

Voici donc la fonction d'analyse syntaxique :

```
(define (meaning e r)
  (if (atom? e)
      (if (symbol? e) (meaning-reference e r)
                      (meaning-quotation e r) )
      (case (car e)
        ((quote)  (meaning-quotation (cadr e) r))
        ((lambda) (meaning-abstraction (cadr e) (cddr e) r))
        ((if)     (meaning-alternative
                    (cadr e) (caddr e) (cadddr e) r ))
        ((begin)  (meaning-sequence (cdr e) r))
        ((set!)   (meaning-assignment (cadr e) (caddr e) r))
        (else     (meaning-application (car e) (cdr e) r)) ) ) )
```

La citation redevient à nouveau triviale :

```
(define (meaning-quotation v r)
  (lambda (sr k)
    (k v) ) )
```

Voici l'alternative, bel exemple de migration de code où l'on peut remarquer que l'on prétraite les deux branches de l'alternative quels que puissent être les choix qui seront éventuellement effectués :

```
(define (meaning-alternative e1 e2 e3 r)
  (let ((m1 (meaning e1 r))
        (m2 (meaning e2 r))
        (m3 (meaning e3 r)) )
    (lambda (sr k)
      (m1 sr (lambda (v)
               ((if v m2 m3) sr k) )) ) ) )
```

La séquence se décompose en ses deux classiques sous-cas :

```
(define (meaning-sequence e+ r)
  (if (pair? e+)
      (if (pair? (cdr e+))
          (meaning*-multiple-sequence (car e+) (cdr e+) r)
          (meaning*-single-sequence (car e+) r) )
      (static-wrong "Illegal syntax: (begin)") ) )
(define (meaning*-single-sequence e r)
  (meaning e r) )
(define (meaning*-multiple-sequence e e+ r)
  (let ((m1 (meaning e r))
        (m+ (meaning-sequence e+ r)) )
    (lambda (sr k)
      (m1 sr (lambda (v)
               (m+ sr k) )) ) ) )
```

Les choses se corsent un peu avec l'application qui va nécessiter de décrire finement le protocole d'invocation.

Application

Pour prétraiter une application, nous devons faire explicitement apparaître la création, le remplissage, la passation, bref la manipulation des blocs d'activation. Si le prétraitement du terme fonctionnel est classique, celui des arguments le sera moins mais sera du ressort de la fonction `meaning*`. Pour simplifier, les fonctions seront représentées par des fermetures.

```
(define (meaning-regular-application e e* r)
  (let* ((m (meaning e r))
         (m* (meaning* e* r (length e*))) )
    (lambda (sr k)
      (m sr (lambda (f)
              (if (procedure? f)
                  (m* sr (lambda (v*)
                           (f v* k) ))
                  (wrong "Not a function" f) ) )) ) ) )
```

Notre recherche de calculs que l'on peut rendre statiques [2], nous permet de voir que la taille du bloc d'activation qui doit être alloué est aisément prévisible puisque directement déduite du nombre de termes de l'application. Par contre, ce qui est moins aisé de découvrir est de savoir quand celui-ci doit être alloué ? Deux instants sont possibles : **[☞ Ex. 6.4]**

1. On peut allouer le bloc avant d'évaluer les arguments. Dans ce cas, chaque argument calculé y est immédiatement rangé à sa bonne place ;
2. On peut allouer le bloc après avoir évalué tous les arguments mais dans ce cas la consommation de mémoire est double puisqu'il faut mémoriser dans la continuation les valeurs qui seront rangées toutes ensemble dans le bloc nouvellement alloué.

La première des deux stratégies semble la plus efficace car moins consommatrice en mémoire. Malheureusement la présence de `call/cc` en Scheme la ruine totalement : elle n'est adaptée qu'à Lisp. La raison est qu'il est possible en Scheme d'appeler une continuation plus d'une fois. **[☞ p. 77]** Si le bloc est alloué en premier et avant le calcul des arguments, alors, si l'un de ces calculs capture sa continuation, il capturera aussi ce bloc qui figurera dans cette continuation. Le bloc sera donc partagé par toutes les invocations du terme fonctionnel, ce qui est contraire à la sémantique de l'abstraction qui doit allouer de nouvelles addresses pour ses variables locales à chaque invocation. Le programme suivant doit retourner `(2 1)` ; partager les blocs d'activation l'amènerait à retourner incorrectement `(2 2)` [3]. En effet, la forme `call/cc` capture l'application `((lambda (a) ...) ...)` et notamment le bloc d'activation si préalloué. Comme cette continuation est utilisée deux fois, les deux suspensions créées sur `(lambda () a)` partageraient la variable close `a` en lui conférant la dernière des valeurs que `k` a reçue.

```
(let ((k 'wait)
```

---

2. C'est d'ailleurs une large part de l'activité des concepteurs de langages que de chercher à favoriser les caractéristiques les plus statiques.

3. Cet exemple, comme Manuel Serrano l'a découvert, dépend subtilement de l'ordre d'évaluation : la forme `cons` doit être évaluée de la gauche vers la droite, c'est ce que réalise la forme `(let ((g ...)) ...)`.

```
      (f '()) )
(set! f (let ((g ((lambda (a) (lambda () a))
                  (call/cc (lambda (nk) (set! k nk) (nk 1))) )))
          (cons g f) ))
;;  f ≈ (list (lambda () 1))
(if (null? (cdr f)) (k 2))
;;  f ≈ (list (lambda () 2) (lambda () 1))
(list ((car f)) ((cadr f))) )
```

En fait tout n'est pas perdu en Scheme, il suffit dans l'implantation de `call/cc` de dupliquer les blocs d'activation capturés lorsque les continuations sont invoquées. Nous ne pouvons le programmer ici car les continuations sont représentées par des fermetures, dont nous ne pouvons extraire les blocs d'activation enclos. On mesure bien, sur cet exemple, l'impact décisif de `call/cc`.

La fonction `meaning*` prendra donc un argument supplémentaire correspondant à la taille du bloc d'activation qu'il faudra allouer après l'évaluation de tous les arguments. Pour des raisons qui apparaîtront lorsque nous discuterons l'implantation des fonctions **[☞ p. 187]** d'arité variable, les blocs d'activation contiendront un champ de plus que nécessaire mais, comme l'on n'initialise pas ce champ excédentaire, ce dernier ne pénalise en rien ces fonctions.

```
(define (meaning* e* r size)
  (if (pair? e*)
      (meaning-some-arguments (car e*) (cdr e*) r size)
      (meaning-no-argument r size) ) )
(define (meaning-no-argument r size)
  (let ((size+1 (+ size 1)))
    (lambda (sr k)
      (let ((v* (allocate-activation-frame size+1)))
        (k v*) ) ) ) )
```

Notez le précalcul de `size+1`, il serait en effet dommage de laisser ce calcul à l'exécution ! Notez aussi la non-migration de la forme d'allocation qui doit allouer un nouveau bloc d'activation à chaque invocation.

Chaque terme de l'application est rangé, juste après l'allocation du bloc d'activation, à sa bonne position, ce qui est aisément calculable en fonction des variables `size` et `e*`.

```
(define (meaning-some-arguments e e* r size)
  (let ((m (meaning e r))
        (m* (meaning* e* r size))
        (rank (- size (+ (length e*) 1))) )
    (lambda (sr k)
      (m sr (lambda (v)
              (m* sr (lambda (v*)
                       (set-activation-frame-argument! v* rank v)
                       (k v*) )) )) ) ) )
```

Nous pouvons enfin décrire les abstractions qui s'énoncent maintenant clairement. Notez la vérification d'arité qui s'effectue par inspection de la taille du bloc d'activation. Là encore, on a précalculé tout ce qu'on a pu de manière à laisser le minimum à l'exécution.

```
(define (meaning-fix-abstraction n* e+ r)
  (let* ((arity (length n*))
         (arity+1 (+ arity 1))
```

```
        (r2 (r-extend* r n*))
        (m+ (meaning-sequence e+ r2)) )
  (lambda (sr k)
    (k (lambda (v* k1)
         (if (= (activation-frame-argument-length v*) arity+1)
             (m+ (sr-extend* sr v*) k1)
             (wrong "Incorrect arity") ) )) ) ) )
```

6.1.4 Classification des variables

Les précédentes définitions n'ont réglé que le cas des variables locales, c'est-à-dire les variables des formes `lambda`. Or il y a aussi des variables globales et parmi celles-là les variables prédéfinies (que nous supposerons immuables) comme `cons` ou `car`. Le seul accès actuellement possible serait de suivre le chaînage des blocs d'activation, ce qui rendrait particulièrement lent l'accès aux variables globales puisque situées dans l'ultime bloc d'activation. Nous allons donc traiter différemment ces variables dont la classification est un processus statique.

Nous allons supposer que la variable globale `g.current` contient la liste des variables globales modifiables tandis que `g.init` contient la liste des variables prédéfinies immuables telles que `car`, `cons`, etc. La fonction `compute-kind` classifie les variables et retourne un descripteur qualifiant les variables.

```
(define (compute-kind r n)
  (or (local-variable? r 0 n)
      (global-variable? g.current n)
      (global-variable? g.init n) ) )
(define (global-variable? g n)
  (let ((var (assq n g)))
    (and (pair? var) (cdr var)) ) )
```

On remarquera que l'on teste `g.current` avant `g.init` afin de pouvoir masquer les variables prédéfinies si besoin est. Considérer les primitives comme valeurs de variables globales immuables est sain et bon. Toutefois, certaines implantations de Scheme autorisent qu'un programme puisse redéfinir une telle variable globale (par exemple `car`) à condition que cela ne modifie que cette variable à l'exclusion de toute fonction prédéfinie (même celles qui semblent utiliser `car` comme, par exemple, `map`). Seules les fonctions figurant explicitement dans le programme verront la nouvelle valeur de `car`. Cet effet peut être simplement obtenu par `compute-kind` mais pose le problème de savoir comment l'on peut insérer une telle variable dans l'environnement modifiable. On peut aussi inventer une nouvelle forme spéciale `redefine` pour cet usage. **[☞ Ex. 6.5]**

On ajoutera des variables globales à ces environnements grâce aux deux fonctions `g.current-extend!` et `g.init-extend!` :

```
(define (g.current-extend! n)
  (let ((level (length g.current)))
    (set! g.current (cons (cons n `(global . ,level)) g.current))
    level ) )
(define (g.init-extend! n)
  (let ((level (length g.init)))
    (set! g.init (cons (cons n `(predefined . ,level)) g.init))
    level ) )
```

Les environnements `g.current` et `g.init` ne retournent que les adresses des variables, leurs valeurs doivent être cherchées dans des conteneurs de valeurs appropriés. Ces conteneurs seront des simples vecteurs, valeurs des variables `sg.current` et `sg.init`. Nous avons laissé un `s` initial car ces vecteurs représentent des mémoires traditionnellement préfixées par cette lettre. Voici ces conteneurs et les fonctions d'accès associées (sans toutefois donner `predefined-update!` qui n'a pas de sens pour des variables immuables)[4].

```
(define sg.current (make-vector 100))
(define sg.init (make-vector 100))
(define (global-fetch i)
  (vector-ref sg.current i) )
(define (global-update! i v)
  (vector-set! sg.current i v) )
(define (predefined-fetch i)
  (vector-ref sg.init i) )
```

Pour aider à la définition des environnements globaux, on fournira deux fonctions, nommées `g.current-initialize!` et `g.init-initialize!`, qui enrichissent (ou modifient) les environnements statiques et dynamiques de façon synchrone.

```
(define (g.current-initialize! name)
  (let ((kind (compute-kind r.init name)))
    (if kind
        (case (car kind)
          ((global)
           (vector-set! sg.current (cdr kind) undefined-value) )
          (else (static-wrong "Wrong redefinition" name)) )
        (let ((index (g.current-extend! name)))
          (vector-set! sg.current index undefined-value) ) ) )
  name )
(define (g.init-initialize! name value)
  (let ((kind (compute-kind r.init name)))
    (if kind
        (case (car kind)
          ((predefined)
           (vector-set! sg.init (cdr kind) value) )
          (else (static-wrong "Wrong redefinition" name)) )
        (let ((index (g.init-extend! name)))
          (vector-set! sg.init index value) ) ) )
  name )
```

Notre arsenal est maintenant paré pour nous permettre d'aborder le prétraitement des variables et des affectations. Les deux ont une structure semblable, analysent la classification retournée par `compute-kind` et lui associent la fonction d'accès correcte. On remarquera le petit coup de pouce consistant à ne pas utiliser les fonctions `deep-fetch` et `deep-update!` mais les accesseurs directs équivalents lorsque la variable recherchée appartient au premier bloc d'activation. Une autre astuce (mais plus contestable) consiste, pour les variables prédéfinies, à chercher leur valeur qui sera lue de suite (comme pour une citation) et non à l'exécution : on y gagne un accès indexé au vecteur des variables globales constantes.

```
(define (meaning-reference n r)
```

4. Par souci de simplicité, on limitera à 100 le nombre de variables globales modifiables. Cette limitation sera levée au paragraphe 6.1.9.

```
(let ((kind (compute-kind r n)))
  (if kind
      (case (car kind)
        ((local)
         (let ((i (cadr kind))
               (j (cddr kind)) )
           (if (= i 0)
               (lambda (sr k)
                 (k (activation-frame-argument sr j)) )
               (lambda (sr k)
                 (k (deep-fetch sr i j)) ) ) ) )
        ((global)
         (let ((i (cdr kind)))
           (if (eq? (global-fetch i) undefined-value)
               (lambda (sr k)
                 (let ((v (global-fetch i)))
                   (if (eq? v undefined-value)
                       (wrong "Uninitialized variable" n)
                       (k v)  ) ) )
               (lambda (sr k)
                 (k (global-fetch i)) ) ) ) )
        ((predefined)
         (let* ((i (cdr kind))
                (value (predefined-fetch i)) )
           (lambda (sr k)
             (k value) ) ) ) )
      (static-wrong "No such variable" n) ) ) )
```

L'affectation est en tout point comparable :

```
(define (meaning-assignment n e r)
  (let ((m (meaning e r))
        (kind (compute-kind r n)) )
    (if kind
        (case (car kind)
          ((local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (lambda (sr k)
                   (m sr (lambda (v)
                           (k (set-activation-frame-argument!
                               sr j v )) )) )
                 (lambda (sr k)
                   (m sr (lambda (v)
                           (k (deep-update! sr i j v)) )) ) ) ) )
          ((global)
           (let ((i (cdr kind)))
             (lambda (sr k)
               (m sr (lambda (v)
                       (k (global-update! i v)) )) ) ) )
          ((predefined)
           (static-wrong "Immutable predefined variable" n) ) )
        (static-wrong "No such variable" n) ) ) )
```

Erreurs statiques

L'intérêt de ce prétraitement est que des erreurs comme de tenter de modifier une variable immuable ou de lire une variable inexistante sont signalées dès le prétraitement et non à l'exécution (ces erreurs peuvent même rester inaperçues si les formes erronées ne sont pas évaluées). Ces erreurs sont signalées par la fonction `static-wrong` et non `wrong`, que l'on réserve pour les situations imprévues pouvant survenir lors de l'exécution. Le concept d'erreur statique est intéressant mais marque bien la différence entre un langage libertaire tel que Lisp et la plupart des autres. Si un programme est valide en ML, alors toutes ses exécutions possibles sont exemptes d'erreur de types. Inversement, si l'on ne sait pas prouver que toutes les évaluation d'un programme conduisent à des erreurs, on aura alors tendance à penser qu'il est licite en Lisp. Considérez, par exemple, la définition suivante :

```
(define (statically-strange n)
  (if (integer? n)
      (if (= n 0) 1 (* n (statically-strange (- n 1))))
      (cons) ) )
```

Bien que statiquement erronée, cette fonction rend de réels services lorsqu'appliquée à des entiers (positifs !). Autoriser ou non une telle fonction dépend du génie du langage et est un compromis entre la sécurité que l'on recherche et la liberté que l'on est prêt à lui sacrifier. Il est très important d'être prévenu de ses erreurs le plus vite possible et c'est ce qu'apporte ML mais, si l'on ne souhaite pas être limité dans son arsenal programmatique et désire connaître la saveur de la facilité et l'ivresse du danger, alors on peut préférer Lisp.

La fonction `static-wrong` doit s'entendre comme délivrant un message d'anomalie mais engendrant un résultat, valide pour le prétraitement qui, lui, signalera une erreur si d'aventure son évaluation est requise. On combine donc avertissement au prétraitement et erreur à l'exécution. Nous pouvons expliciter cet effet en définissant `static-wrong` comme :

```
(set! static-wrong
      (lambda (message . culprits)
        (display `(*static-error* ,message . ,culprits))(newline)
        (lambda (sr k)
          (apply wrong message culprits) ) ) )
```

Ainsi peut-on accéder au nirvana des implanteurs : conserver le beurre et son équivalent monétaire.

On peut aussi remarquer le coût d'accès aux variables globales modifiables qui nécessite de vérifier que celles-ci sont bien initialisées, ce qui n'est jamais nécessaire avec les variables locales ou les variables globales initialisées et immuables. Dans le cas d'une compilation incrémentielle, comme par exemple celle d'une boucle d'interaction compilante (une boucle `(display (compile-then-run (read)))`), on peut un petit peu améliorer ce prétraitement pour les variables globales modifiables déjà initialisées : c'est ce qui a été fait plus haut dans `meaning-reference`[5] **[☞ Ex. 7.6]**.

---

5. S'il est possible de modifier le code en place, on peut aussi imaginer rustiner (pour *patch*) l'instruction qui vérifie l'initialisation d'une variable globale, pour ne plus le faire si c'est bien le cas. Bigloo [Ser94] adopte cette solution.

6.1.5 Lancement de l'interprète

L'interprète lit une expression, la prétraite puis l'évalue. Il réalise ainsi une boucle d'interaction compilante.

```
(define r.init '())
(define sr.init '())
(define (chapter61-interpreter)
  (define (compile e) (meaning e r.init))
  (define (run c) (c sr.init display))
  (define (toplevel)
    (run (compile (read)))
    (toplevel) )
  (toplevel) )
```

Mais avant de lancer cet interprète, il faut un peu enrichir son environnement initial. Nous supposerons encore disposer de macros permettant de cacher les détails d'implantation, ce qui permet aux définitions qui suivent de ressembler à ce que toujours elles furent. En voici quelques-unes auxquelles on nous pardonnera d'ajouter l'inénarrable `call/cc` :

```
(definitial t #t)
(definitial f #f)
(definitial nil '())
(defprimitive cons cons 2)
(defprimitive car car 1)
(definitial call/cc
(let* ((arity 1)
       (arity+1 (+ arity 1)) )
  (lambda (v* k)
    (if (= arity+1 (activation-frame-argument-length v*))
        ((activation-frame-argument v* 0)
         (let ((frame (allocate-activation-frame (+ 1 1))))
           (set-activation-frame-argument!
            frame 0
            (lambda (values kk)
              (if (= (activation-frame-argument-length values)
                     arity+1 )
                  (k (activation-frame-argument values 0))
                  (wrong "Incorrect arity" 'continuation) ) ) )
           frame )
         k )
        (wrong "Incorrect arity" 'call/cc) ) ) ) )
```

6.1.6 Fonction d'arité variable

Manquaient à notre interprète les fonctions d'arité variable qui induisent, comme toujours, quelques difficultés. Tels que nous les avions utilisés, les blocs d'activation contiennent les valeurs des arguments mais servent aussi de réceptacles aux liaisons qui sont à créer. Dans le cas des fonctions d'arité variable, la correspondance entre ces deux effets n'est pas assurée car il n'y a pas de rapport direct entre le nombre d'arguments soumis à la fonction et son arité. Par exemple, une fonction ayant `(a b . c)` comme liste de variables peut recevoir sans erreur, deux, ou trois arguments ou davantage mais elle ne liera jamais que ces trois variables. Il est donc nécessaire que

le bloc d'activation contienne toujours (et pour cet exemple) au moins trois champs. Une parade simple est que, pour une application `(`*f a b*`)`, le bloc d'activation alloué soit doté d'un champ superflu, soit trois champs en tout, ce qui autorisera l'invocation de toute fonction capable de recevoir au moins deux valeurs, c'est-à-dire les fonctions ayant des listes de variables congruentes à `(x y)` ou `(x y . z)` ou `(x . y)` ou encore `x`.

Les fonctions d'arité variable manipuleront donc le bloc d'activation qu'elles reçoivent afin de procéder à la mise en liste des arguments excédentaires. Cette fonction nommée `listify!` ne sera utilisée que pour son effet. Sa programmation n'est pas complexe mais requiert de valser avec les divers indices numérotant les termes de l'application, les variables à lier et les champs du bloc d'activation. La valeur de la variable `arity` représente le nombre minimal d'arguments attendus.

```
(define (meaning-dotted-abstraction n* n e+ r)
  (let* ((arity (length n*))
         (arity+1 (+ arity 1))
         (r2 (r-extend* r (append n* (list n))))
         (m+ (meaning-sequence e+ r2)) )
    (lambda (sr k)
      (k (lambda (v* k1)
           (if (>= (activation-frame-argument-length v*) arity+1)
               (begin (listify! v* arity)
                      (m+ (sr-extend* sr v*) k1) )
               (wrong "Incorrect arity") ) )) ) ) )
(define (listify! v* arity)
  (let loop ((index (- (activation-frame-argument-length v*) 1))
             (result '()) )
    (if (= arity index)
        (set-activation-frame-argument! v* arity result)
        (loop (- index 1)
              (cons (activation-frame-argument v* (- index 1))
                    result ) ) ) ) )
```

On peut maintenant prétraiter toutes les formes `lambda` possibles grâce à l'analyseur statique suivant :

```
(define (meaning-abstraction nn* e+ r)
  (let parse ((n* nn*)
              (regular '()) )
    (cond
     ((pair? n*) (parse (cdr n*) (cons (car n*) regular)))
     ((null? n*) (meaning-fix-abstraction nn* e+ r))
     (else (meaning-dotted-abstraction
            (reverse regular) n* e+ r )) ) ) )
```

6.1.7 Formes réductibles

Une amélioration classique des compilateurs, que notre prétraiteur peut incorporer sans problème, concerne les formes réductibles, c'est-à-dire les applications dont le terme fonctionnel est une forme `lambda`. Il est alors inutile de créer une fermeture pour l'appliquer de suite, il suffit de l'assimiler à un bloc avec variables locales comme en Algol. D'ailleurs `((lambda .....) ...)` n'est autre qu'un `let` déguisé, ouvrant un bloc pourvu de variables locales. Le cas des fonctions d'arité fixe est simplissime

ce qui, une fois encore, n'est pas l'apanage des fonctions d'arité variable [6]. On remarquera que ne pas fournir le bon nombre d'arguments à la fonction est maintenant une erreur statique qui peut être signalée dès le prétraitement.

```
(define (meaning-closed-application e ee* r)
  (let ((nn* (cadr e)))
    (let parse ((n* nn*)
                (e* ee*)
                (regular '()) )
      (cond ((pair? n*)
             (if (pair? e*)
                 (parse (cdr n*) (cdr e*) (cons (car n*) regular))
                 (static-wrong "Too less arguments" e ee*) ) )
            ((null? n*)
             (if (null? e*)
                 (meaning-fix-closed-application
                  nn* (cddr e) ee* r )
                 (static-wrong "Too much arguments" e ee*) ) )
            (else (meaning-dotted-closed-application
                   (reverse regular) n* (cddr e) ee* r )) ) ) ) ) )
(define (meaning-fix-closed-application n* body e* r)
  (let* ((m* (meaning* e* r (length e*)))
         (r2 (r-extend* r n*))
         (m+ (meaning-sequence body r2)) )
    (lambda (sr k)
      (m* sr (lambda (v*)
               (m+ (sr-extend* sr v*) k) )) ) ) )
```

Pour les fonctions d'arité variable, on peut éviter d'user de `listify!` puisqu'ici à la fois l'arité de la fonction et le nombre d'arguments sont statiquement connus. La solution passe par une variante de `meaning*`, nommée ici `meaning-dotted*`, qui se comporte comme elle pour les arguments obligatoires mais qui fabrique à la volée la liste des arguments excédentaires en insérant les nécessaires appels à `cons` ; c'est certainement beaucoup de code pour un cas assez rare. Il faut par contre explicitement initialiser avec `()` le champ superflu des blocs d'activation ce que réalise la fonction `meaning-no-dotted-argument`. Cela revient à convertir au vol :

```
((lambda (a b . c) ...)                ≡   ((lambda (a b c) ...)
 α β γ δ ...)                                α β (cons γ (cons δ ...)) )
```

Voici donc les fonctions :

```
(define (meaning-dotted-closed-application n* n body e* r)
  (let* ((m* (meaning-dotted* e* r (length e*) (length n*)))
         (r2 (r-extend* r (append n* (list n))))
         (m+ (meaning-sequence body r2)) )
    (lambda (sr k)
      (m* sr (lambda (v*)
               (m+ (sr-extend* sr v*) k) )) ) ) )
(define (meaning-dotted* e* r size arity)
  (if (pair? e*)
      (meaning-some-dotted-arguments (car e*) (cdr e*) r size arity)
      (meaning-no-dotted-argument r size arity) ) )
(define (meaning-some-dotted-arguments e e* r size arity)
  (let ((m (meaning e r))
        (m* (meaning-dotted* e* r size arity))
```

6. C'est pourquoi de nombreux langages ne les procurent pas malgré leur intérêt.

```
        (rank (- size (+ (length e*) 1))) )
    (if (< rank arity)
        (lambda (sr k)
          (m sr (lambda (v)
                  (m* sr (lambda (v*)
                           (set-activation-frame-argument! v* rank v)
                           (k v*) )) )) )
        (lambda (sr k)
          (m sr (lambda (v)
                  (m* sr (lambda (v*)
                           (set-activation-frame-argument!
                            v* arity
                            (cons v (activation-frame-argument
                                     v* arity )) )
                           (k v*) )) )) ) ) ) )
(define (meaning-no-dotted-argument r size arity)
  (let ((arity+1 (+ arity 1)))
    (lambda (sr k)
      (let ((v* (allocate-activation-frame arity+1)))
        (set-activation-frame-argument! v* arity '())
        (k v*) ) ) ) )
```

6.1.8 Intégration des primitives

Une autre source importante de gains est le prétraitement astucieux des appels aux fonctions prédéfinies de l'environnement global immuable. Une application telle que `(car` α`)` impose, actuellement, l'incroyablement pénible séquence suivante :

1. déréférencer la variable globale `car` ;
2. évaluer α ;
3. allouer un bloc d'activation à deux champs ;
4. remplir le premier champ avec la valeur de α ;
5. vérifier que la valeur de `car` est bien une fonction ;
6. vérifier que la valeur de `car` accepte bien un unique argument ;
7. appliquer la valeur de `car` sur l'argument (ce qui conduira à tester en plus que l'argument est bien une paire pointée dont il est légitime de prendre le `car`).

Beaucoup de ces points sont éliminés si l'on est dans un langage à typage fort et c'est ce qui les rend si véloces. Par contre, certaines de ces vérifications peuvent être supprimées dans un langage comme Lisp par prétraitement car elles sont statiquement vérifiables. Comme `car` est une variable globale non modifiable, on peut déjà vérifier que c'est une fonction qui peut accepter un unique argument : on gagne donc le test de fonctionnalité et celui d'arité (points 5 et 6). On peut gagner encore plus en n'allouant pas le bloc d'activation (points 3 et 4) et en insérant le code même de la primitive appelée (point 1). C'est une intégration (pour *inlining*) qui permet d'appeler directement la primitive sans passer par le protocole complet mais lourd. Il ne reste plus que les points 2 et 7. Un bon compilateur peut encore gagner un peu sur le point 7 en factorisant les tests de types de manière à ne jamais les dupliquer. Par exemple, dans le programme `(if (pair? e) (car e) ...)`, il est inutile que `car` vérifie que son argument est une paire car cela est sûrement vrai. Une méthode pour ce faire est

de considérer `(car x)` comme une macro équivalente à `(let ((v x))(if (pair? v) (unsafe-car v) (error ...)))`, où `unsafe-car` extrait le `car` de son argument si celui-ci est une paire et conduit à des effets imprévisibles si ce n'est pas le cas[7]. Il ne suffit plus que de reprendre le code pour faire migrer les tests de types le plus en amont possible et d'éliminer les tests redondants. Là encore, il n'y a pas de problèmes à faire migrer ces calculs car les tests de types sont des calculs immédiats et sans erreur possible.

Un véritable compilateur n'a pas accès aux valeurs des variables globales immuables puisque celles-ci appartiennent au monde de l'exécution et non à celui du prétraitement. D'ailleurs nous n'avons pas tant besoin de ces valeurs que de seulement savoir si ce sont des fonctions et si leur arité est compatible avec l'application que l'on chercher à prétraiter. Nous allons ajouter un nouvel environnement, dont le rôle sera de décrire les valeurs des variables globales immuables. Cet environnement, nommé `desc.init`, associera à une variable, dont la valeur est une fonction, un descripteur. Le descripteur d'une fonction sera une liste, dont le premier terme sera le symbole `function`, le second terme sera l'« adresse » de la primitive (ce qu'il faudra réellement invoquer) et les termes suivants indiqueront l'arité. Nous noierons la gestion de ces descriptions au sein de la macro `defprimitive` que l'on accompagnera d'une seule de ses sous-macros, ici `defprimitive3`, donnant le goût de ce que sont ses consœurs :

```
(define-syntax defprimitive
  (syntax-rules ()
    ((defprimitive name value 0) (defprimitive0 name value))
    ((defprimitive name value 1) (defprimitive1 name value))
    ((defprimitive name value 2) (defprimitive2 name value))
    ((defprimitive name value 3) (defprimitive3 name value)) ) )
(define-syntax defprimitive3
  (syntax-rules ()
    ((defprimitive3 name value)
     (definitial name
       (letrec ((arity+1 (+ 3 1))
                (behavior
                 (lambda (v* k)
                   (if (= (activation-frame-argument-length v*)
                          arity+1 )
                       (k (value (activation-frame-argument v* 0)
                                 (activation-frame-argument v* 1)
                                 (activation-frame-argument v* 2) ))
                       (wrong "Incorrect arity" 'name) ) ) ) )
         (description-extend!                          ; ** Modified **
          'name `(function ,value a b c))
         behavior ) ) ) ) )
```

Les fonctions de gestion de l'environnement seront :

```
(define desc.init '())
(define (description-extend! name description)
  (set! desc.init (cons (cons name description) desc.init)))
```

---

7. Des primitives analogues à `unsafe-car` existent dans la plupart des implantations afin de servir de cibles à des transformations de programmes. Ces transformations garantissent que `unsafe-car` est toujours employée dans des contextes sains. Il est par contre essentiel qu'un compilateur performant puisse recourir à ces primitives et ainsi supprimer d'inutiles vérifications de types.

```
  name )
(define (get-description name)
  (let ((p (assq name desc.init)))
    (and (pair? p) (cdr p)) ) )
```

Nous pouvons entièrement expliciter comment sont prétraitées les applications, c'est-à-dire comment elles sont analysées afin d'être aiguillées vers le bon prétraiteur. Notez que si une primitive figure dans une application avec une mauvaise arité, l'anomalie sera signalée statiquement.

```
(define (meaning-application e e* r)
 (cond
  ((and (symbol? e)
        (let ((kind (compute-kind r e)))
          (and (pair? kind)
               (eq? 'predefined (car kind))
               (let ((desc (get-description e)))
                 (and desc
                      (eq? 'function (car desc))
                      (if (= (length (cddr desc)) (length e*))
                          (meaning-primitive-application e e* r)
                          (static-wrong "Incorrect arity for" e) ) )
                 ) ) ) ))
  ((and (pair? e)
        (eq? 'lambda (car e)) )
   (meaning-closed-application e e* r) )
  (else (meaning-regular-application e e* r)) ) )
```

Les applications impliquant des fonctions primitives connues sont gérées comme suit :

```
(define (meaning-primitive-application e e* r)
  (let* ((desc (get-description e)) ; desc = (function address . variables-list)
         (address (cadr desc))
         (size (length e*)) )
    (case size
      ((0) (lambda (sr k) (k (address))))
      ((1)
       (let ((m1 (meaning (car e*) r)))
         (lambda (sr k)
           (m1 sr (lambda (v)
                    (k (address v)) )) ) ) )
      ((2)
       (let ((m1 (meaning (car e*) r))
             (m2 (meaning (cadr e*) r)) )
         (lambda (sr k)
           (m1 sr (lambda (v1)
                    (m2 sr (lambda (v2)
                             (k (address v1 v2)) )) )) ) ) )
      ((3)
       (let ((m1 (meaning (car e*) r))
             (m2 (meaning (cadr e*) r))
             (m3 (meaning (caddr e*) r)) )
         (lambda (sr k)
           (m1 sr (lambda (v1)
                    (m2 sr (lambda (v2)
                             (m3 sr (lambda (v3)
                                      (k (address v1 v2 v3))
                                      )) )) )) ) ) )
```

```
(else (meaning-regular-application e e* r)) ) ) )
```

L'intégration précédente ne porte que sur les primitives d'arité fixe. Les primitives d'arité variable comme `append`, `for-each`, `list`, `map`, `*`, `+` et quelques autres, hors `apply`, peuvent en général être considérées comme des macros dont l'expansion se ramène aux cas déjà étudiés. Par exemple, `(append` π_1 π_2 π_3`)` peut être réécrit en `(append` π_1 `(append` π_2 π_3`))`. La transformation permet de se passer de formes intégrables d'arité variable mais n'implique pas que ces fonctions soient d'arité fixe : `(apply append` π`)` est un exemple où `append` sera appelée avec une arité variable.

On remarquera que nous n'avons intégré que les appels comportant trois arguments ou moins. La raison en est simple : il n'y a pas en Scheme de fonctions essentielles d'arité fixe possédant plus de trois arguments !

6.1.9 Variantes sur environnements

L'accès aux variables locales profondes, c'est-à-dire à celles qui n'appartiennent pas au premier bloc d'activation subissent un coût linéaire puisqu'il faut parcourir le chaînage des blocs. Une technique simple dite du *display* permet d'accéder à ces variables en temps constant. Il suffit pour cela que tout bloc d'activation profond soit accessible depuis le premier (Figure 6.3). Ainsi toute variable profonde peut être lue ou écrite en une indirection (déterminant le bloc) et un décalage (au sein de ce bloc). Si maintenant l'accès aux variables profondes est plus rapide, le coût en allocation et en chaînage est plus important car tout bloc d'activation doit référencer tous les blocs profonds. Certes, il est possible de ne mettre que les chaînages réellement utilisés mais cela demande une analyse des variables consultées. D'autre part, cette technique ruine notre interprète puisque l'on ne sait plus quelle doit être la taille du bloc d'activation à créer avant que ne soit consultée la profondeur de l'environnement clos par la fonction à invoquer. Il faudrait donc allouer le *display* ailleurs ou bien limiter la profondeur maximale autorisée (ce qui n'est pas très lispien).

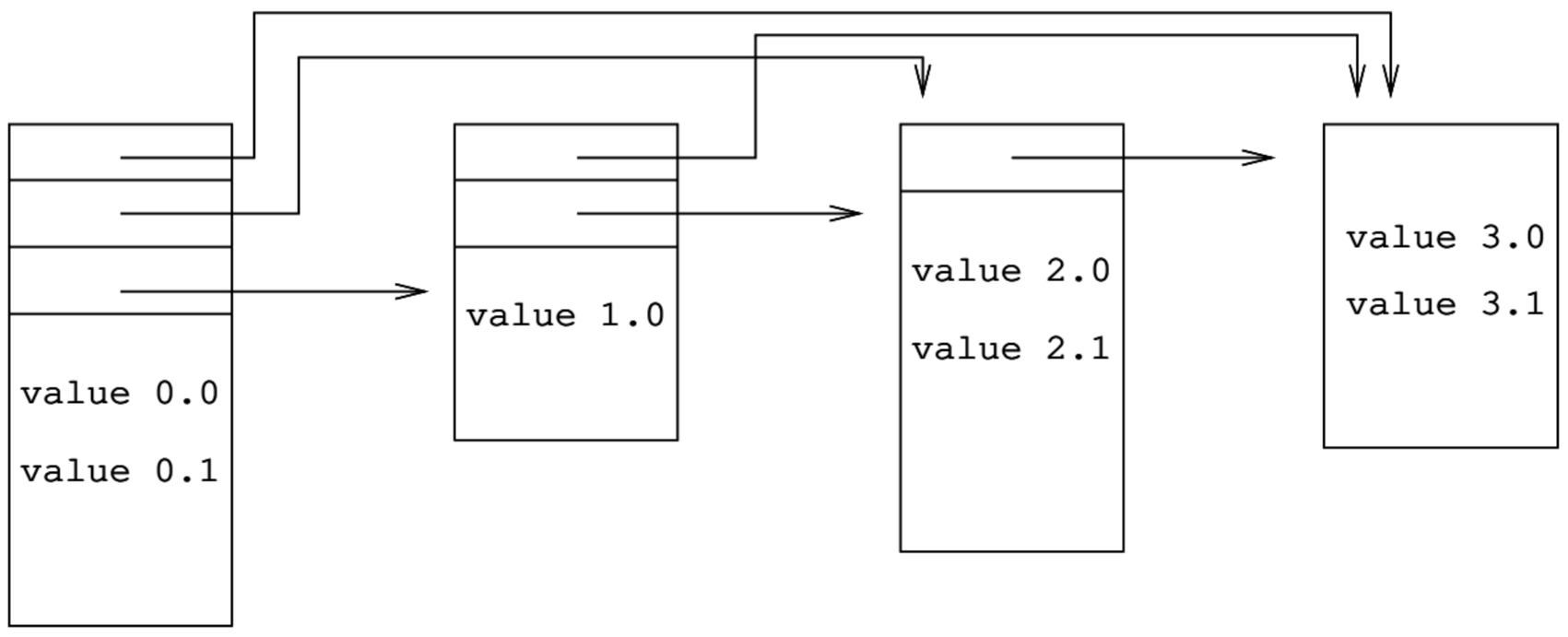


FIGURE 6.3 – Technique du *display*

Environnement plat

Une autre technique serait d'adopter des environnements plats. Lorsqu'une fermeture est créée et au lieu de clore l'environnement tout entier, on pourrait forger un nouvel environnement contenant uniquement les variables réellement closes. Le coût de fabrication des fermetures est alors plus élevé mais, par contre, l'environnement ne comporte au maximum que deux blocs d'activation, celui qui contient les arguments et celui qui contient les variables closes. Afin cependant de conserver les éventuels partages de variables, il est bon d'effectuer une transformation par boîte. [☞ p. 108]

Enfin, il est possible de mélanger toutes ces possibilités afin de les adapter aux cas particuliers où elles excellent. Cette adaptation nécessite cependant des analyses plus fines des programmes, bref un véritable compilateur et non un interprète avec préconditionnement comme le nôtre.

Définition de variables globales

L'interprète actuel lorsqu'il rencontre une variable globale la reconnaît en raison de son appartenance à l'environnement global prédéfini ou modifiable. Il faut donc déclarer (dans la définition de l'interprète) toutes les variables globales que l'on va utiliser grâce à la macro `defvariable` suivante :

```
(define-syntax defvariable
  (syntax-rules ()
    ((defvariable name) (g.current-initialize! 'name)) ) )
```

Quoiqu'il soit de règle dans beaucoup de langages, cet usage est considéré comme tout à fait déplacé dans un langage lispien. Nous voulons donc, lorsqu'un identificateur est utilisé comme une variable globale, que celle-ci soit automatiquement créée. Une solution simple est d'enrichir la fonction `compute-kind` pour détecter ce cas :

```
(define (compute-kind r n)
  (or (local-variable? r 0 n)
      (global-variable? g.current n)
      (global-variable? g.init n)
      (adjoin-global-variable! n) ) )
(define (adjoin-global-variable! name)
  (let ((index (g.current-extend! name)))
    (vector-set! sg.current index undefined-value)
    (cdr (car g.current)) ) )
```

Le problème est que, ce faisant, nous avons un peu violé la discipline de prétraitement que nous avions adoptée. En effet la variable globale est créée lors du prétraitement et non pendant l'exécution. Analysons plus avant le processus de création d'une variable. La prise en compte d'une nouvelle variable résulte de deux effets différents :

1. Son nom est ajouté à l'environnement des variables globales modifiables (valeur de `g.current`) et, au passage, un numéro lui est conféré.
2. Une addresse lui est allouée, qui contiendra sa valeur, conformément au numéro conféré.

Si l'on considère le programme suivant :

```
(begin (define foo (lambda () (bar)))
       (define bar (lambda () (if #t 33 hux))) )
```

Le prétraitement révèle l'existence de trois nouvelles variables `foo`, `bar` et `hux`. Ces variables sont donc ajoutées, dès que rencontrées, par le prétraitement à l'environnement global et reçoivent un numéro. Sont-elles alors, après prétraitement, créées ou seulement potentiellement existantes ? La variable `foo` figurant dans la forme `define` est-elle à demi-créée lorsque `hux` ne le serait qu'au quart voire même pas du tout ?

Exécuter le programme prétraité suppose que l'environnement global modifiable d'exécution contient les valeurs de ces nouvelles variables. Il faut donc l'étendre à quelque moment judicieux : au début de la phase d'exécution est en fait le seul instant possible. Avant l'exécution du programme prétraité, la taille de l'environnement global d'exécution est adaptée au nombre de variables globales modifiables présentes. C'est ce qu'illustre la fonction `stand-alone-producer` qui prétraite un programme en une fermeture `(lambda (sr k) ...)`, dont le premier acte est d'allouer l'environnement global ad hoc.

```
(define (stand-alone-producer e)
  (set! g.current (original.g.current))
  (let* ((m (meaning e r.init))
         (size (length g.current)) )
    (lambda (sr k)
      (set! sg.current (make-vector size undefined-value))
      (m sr k) ) ) )
```

Comme il se peut qu'il y ait dans l'environnement global modifiable des variables prédéfinies pouvant servir de lieu de communication entre le système et le programme (comme par exemple la base de numération à utiliser pour lire ou écrire des nombres), on supposera que ces variables figurent dans l'environnement retourné par `(original-.g.current)`.

Avec cette programmation, toutes les variables existent dès qu'elles sont citées. On peut aussi, de façon plus conforme à Scheme, ne permettre l'introduction de variables que par la forme `define`. Dans ce dernier cas et pour l'exemple ci-dessus, le prétraitement doit signaler une erreur statique : `hux` est une variable non définie, alors même qu'elle n'est ni lue ni écrite. Ce n'est pas le cas de `bar` qui, quoique non encore définie dans son premier emploi, l'est par la suite.

Pour détailler alors le fonctionnement de `define`, on ajoutera cette nouvelle forme spéciale au prétraiteur `meaning` :

```
... ((define) (meaning-define (cadr e) (caddr e) r)) ...
```

L'analyse particulière des définitions teste si la variable n'existe pas déjà et, dans le cas contraire, se comporte comme une affectation.

```
(define (meaning-define n e r)
  (let ((b (memq n *defined*)))
    (if (pair? b)
        (static-wrong "Already defined variable" n)
        (set! *defined* (cons n *defined*)) ) )
  (meaning-assignment n e r) )
```

Le conditionneur de programmes a maintenant pour charge de vérifier qu'il n'y a pas de variables globales subitement apparues qui n'aient pas fait l'objet d'une définition explicite. La liste des variables globales définies est donc synthétisée en début de prétraitement

```
(define (stand-alone-producer e)
  (set! g.current (original.g.current))
```

```
(let ((originally-defined (append (map car g.current)
                                  (map car g.init) )))
  (set! *defined* originally-defined)
  (let* ((m (meaning e r.init))
         (size (length g.current))
         (anormals (set-difference (map car g.current)
                                   *defined* )) )
    (if (null? anormals)
        (lambda (sr k)
          (set! sg.current (make-vector size undefined-value))
          (m sr k) )
        (static-wrong "Not explicitely defined variables"
                      anormals ) ) ) ) )
(define (set-difference set1 set2)
  (if (pair? set1)
      (if (memq (car set1) set2)
          (set-difference (cdr set1) set2)
          (cons (car set1) (set-difference (cdr set1) set2)) )
      '() ) )
```

En résumé, il faut bien distinguer ce qui est du ressort du prétraitement et ce qui appartient à l'exécution. Une définition est à l'exécution rigoureusement analogue à une affectation. En revanche, son prétraitement induit des vérifications instantanées ou différées jusqu'à la fin du prétraitement. Les formes spéciales classiques, autres que `define`, n'ont d'effet qu'à l'exécution : elles ont une sémantique dynamique. La définition ressortit à la sémantique statique (celle qu'implante le prétraitement) en énonçant qu'il ne saurait y avoir de variable globale ne faisant pas l'objet d'une définition concomitante. C'est en outre une erreur statique que de contrevenir à cette règle.

Comme précédemment dit pour les variables locales de `letrec`, **[☞ p. 56]** il faut bien distinguer les notions d'existence et d'initialisation de variables. Que les variables existent ne signifie pas qu'elles soient initialisées comme le montre l'exemple suivant :

```
(begin (define foo bar)
       (define bar (lambda () 33)) )
```

Dans le cas d'une boucle d'interaction compilante, une sorte de compilateur incrémentiel évaluant immédiatement ce qu'il vient de compiler, tous ces problèmes sont simplifiés car la fin du prétraitement coïncide avec le début de l'exécution du code prétraité et ces deux phases se passent dans le même espace mémoire. On peut donc adopter la première variante de `adjoin-global-variable!` ainsi que l'amélioration présente dans `meaning-reference` qui consistait pour les variables déjà initialisées à supprimer le test d'initialisation.

6.1.10 Conclusions sur l'interprète à calculs migrés

Il est difficile de mesurer exactement le gain qu'apporte cet interprète mais il est de l'ordre de 10 à 50. Le prétraitement est assez rapide encore que l'on puisse largement améliorer sa vitesse. Outre ces gains non négligeables sont apparues les notions de calculs statiques (objets du prétraitement) ou dynamiques (laissés à l'exécution). Le rôle d'un bon compilateur est, d'une certaine manière, de laisser le minimum de

choses à faire à l'exécution. De nombreuses analyses existent qui recherchent des propriétés qui seront valables à l'exécution de manière à inventer les meilleurs schémas de prétraitement : citons seulement, l'interprétation abstraite [CC77], l'évaluation partielle [JGS93], l'analyse de flot de contrôle [Shi91]. Un autre champ d'amélioration est le choix de bonnes structures de données comme a pu le montrer la discussion sur les blocs d'activation.

Le prétraitement permet de découvrir plus vite certaines erreurs indépendamment de leur possible exécution. Il ouvre la porte vers une programmation plus sûre et plus efficace (car moins de vérifications sont laissées à l'exécution). Celui exposé ici est assez rudimentaire mais pourrait être étendu à la vérification de types comme en ML.

L'interprète finalement obtenu ressemble fort à celui obtenu au chapitre 3 sauf que l'on a remplacé les objets par des fermetures. Bien que les fermetures et les objets aient des représentations internes tout à fait similaires, les fermetures sont des « objets du pauvre », opaques, qui ne peuvent être qu'invoquées et à qui l'on ne saurait conférer de nouveaux comportements, par exemple, pour introduire un peu de réflexivité.

6.2 Rejet de l'environnement

Toute expression est évaluée dans un unique environnement résiduel, valeur de `sr`. Comme celui-ci ne change que lors des invocations de fonctions qui réinstallent puis étendent leur environnement de définition, on peut estimer qu'il serait plus utile d'introduire la notion d'environnement courant, valeur de la variable globale (on pourrait dire, le registre) `*env*`. Cette globalisation éviterait de passer les environnements explicitement en argument à toutes les fermetures résultant du prétraitement. Autrement dit, le résultat du prétraitement ne serait plus une abstraction `(lambda (sr k) ...)` mais, plus simplement, `(lambda (k) ...)`.

Pour réaliser cette transformation, visant à supprimer les variables locales `sr` au profit d'une variable globale `*env*`, il suffit de modifier la fonction `meaning` pour y introduire la gestion de cette variable. A priori les modifications semblent simples :

1. Les variables locales sont recherchées dans `*env*` et non plus dans `sr`.
2. Lorsqu'une fonction est invoquée, elle affecte `*env*` avec son environnement de définition convenablement étendu.

Mais ce n'est pas suffisant car si `*env*` est affectée, il sera de temps à autre nécessaire de restaurer sa valeur précédente, ne serait-ce que pour reprendre un calcul temporairement interrompu le temps d'une invocation. Une solution triviale est que, lorsqu'une fonction affecte `*env*`, elle mémorise sa valeur précédente et la restaure au moment où elle retourne sa valeur finale. Le problème de cette solution est que cela n'est pas conforme à la définition de Scheme qui exige que les appels terminaux soient évalués à continuation constante.

Ce problème est semblable au problème bien connu de la définition du protocole d'appel fonctionnel en termes des registres de la machine. Les registres doivent être sauvegardés pendant le calcul de la fonction invoquée mais qui doit le faire ?

— L'appelé sait exactement quels registres il utilise et par conséquent il peut se limiter à ne préserver que ceux-là. L'ennui est que l'appelé commence donc par

sauvegarder ces registres, évalue son corps, sauve la valeur produite, restaure les registres puis retourne la valeur finale. Le corps de la fonction est donc évaluée avec une continuation qui est celle de l'appelant plus le fait qu'il faut restaurer les registres.

— L'appelant sait exactement quels sont les registres dont il aura besoin après l'invocation et peut ne sauvegarder qu'eux lui-même. L'appelé n'a donc qu'à évaluer son corps et retourner la valeur produite. Si l'appelant n'a pas lui-même à restaurer des registres, on voit que l'appelé n'ajoute aucune contrainte. Mais cette technique est manifestement trop pointilleuse puisque l'appelé peut n'avoir besoin que de peu de registres et pourrait se suffire des registres non utilisés par l'appelant sans requérir que celui-ci sauve quoi que ce soit.

Il a été proposé [SS80] que l'appelant étiquette les registres en « doit être sauvegardé » ou « peut être sans dommage écrasé ». L'appelé peut alors ne sauvegarder que ce qui est vraiment nécessaire et changer l'étiquetage en « doit être restauré » ou « n'a pas changé ». Cette technique revient à faire de chaque registre une pile dont seul le sommet est accessible et dont les profondeurs ont engrangé les valeurs qu'il faudra restaurer. Lors du retour, une instruction spéciale de la machine restaure les registres en fonction de leur étiquetage.

Il est important que les appels terminaux soient exécutés à continuation constante (c'est-à-dire sans accroissement de la pile de récursion) afin que les itérations puissent être efficaces et non bornées par la taille de la pile de récursion. Nous adopterons donc la seconde stratégie conduisant l'appelant à être responsable de la préservation de l'environnement.

Fort heureusement, savoir si une évaluation est terminale est une propriété statique. Nous adjoindrons donc à la fonction `meaning` un argument supplémentaire, un booléen nommé `tail?`, indiquant si l'expression est en position terminale ou pas. Si l'expression est en position terminale, il n'est pas nécessaire de restaurer l'environnement. Si par contre l'expression n'est pas terminale, il est alors obligatoire de restaurer l'environnement. Notez qu'il est n'est pas interdit en position terminale de préserver l'environnement, c'est superflu mais non proscrit.

Comment déterminer les expressions en position terminale ? Il suffit de regarder la dénotation de Scheme : toute sous-forme, évaluée avec une continuation différente de celle de la forme qui la contient, n'est pas en position terminale. Dans une affectation `(set!` x π`)`, la sous-forme π a une continuation différente de celle de l'affectation (puisque sa valeur doit être écrite dans la variable x), elle n'est donc pas en position terminale. Ainsi dans une alternative `(if` π_0 π_1 π_2`)`, la condition π_0 n'est pas en position terminale. Dans une séquence `(begin` π_0 $\ldots\pi_{n-1}$ π_n`)`, les formes $\pi_0 \ldots\pi_{n-1}$ ne sont pas en position terminale : il faut donc préserver l'environnement entre les calculs des différents termes de la séquence. Enfin dans une application `(`π_0 $\ldots\pi_n$`)`, aucun terme n'est en position terminale puisque l'invocation reste à faire. À l'inverse, le corps des fonctions est en position terminale ainsi que l'évaluation du programme tout entier puisque ni l'un ni l'autre n'ont besoin de restaurer l'environnement les précédant.

Voici donc la nouvelle fonction `meaning` suivie de la fonction permettant de lancer ce nouvel interprète.

```
(define (meaning e r tail?)
  (if (atom? e)
      (if (symbol? e) (meaning-reference e r tail?)
                      (meaning-quotation e r tail?) )
      (case (car e)
        ((quote)  (meaning-quotation (cadr e) r tail?))
        ((lambda) (meaning-abstraction (cadr e) (cddr e) r tail?))
        ((if)     (meaning-alternative (cadr e) (caddr e) (cadddr e)
                                       r tail? ))
        ((begin)  (meaning-sequence (cdr e) r tail?))
        ((set!)   (meaning-assignment (cadr e) (caddr e) r tail?))
        (else
         (meaning-application (car e) (cdr e) r tail?) ) ) ) )
(define *env* sr.init)
(define (chapter62-interpreter)
  (define (toplevel)
    (set! *env* sr.init)
    ((meaning (read) r.init #t) display)
    (toplevel) )
  (toplevel) )
```

La référence aux variables use maintenant du registre `*env*`. Nous ne donnerons pas la nouvelle version de `meaning-assignment` qui s'en déduit aisément :

```
(define (meaning-reference n r tail?)
  (let ((kind (compute-kind r n)))
    (if kind
        (case (car kind)
          ((local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (lambda (k)
                   (k (activation-frame-argument *env* j)) )
                 (lambda (k)
                   (k (deep-fetch *env* i j)) ) ) ) )
          ((global)
           (let ((i (cdr kind)))
             (if (eq? (global-fetch i) undefined-value)
                 (lambda (k)
                   (let ((v (global-fetch i)))
                     (if (eq? v undefined-value)
                         (wrong "Uninitialized variable" n)
                         (k v)  ) ) )
                 (lambda (k)
                   (k (global-fetch i)) ) ) ) )
          ((predefined)
           (let* ((i (cdr kind))
                  (value (predefined-fetch i)) )
             (lambda (k)
               (k value) ) ) ) )
        (static-wrong "No such variable" n) ) ) )
```

Sautons la citation qui n'a que faire d'être ou non en position terminale et passons à l'alternative. La condition n'est pas en position terminale et cela se voit !

```
(define (meaning-alternative e1 e2 e3 r tail?)
  (let ((m1 (meaning e1 r #f))          ; restore environment!
        (m2 (meaning e2 r tail?))
```

```
      (m3 (meaning e3 r tail?)) )
  (lambda (k)
    (m1 (lambda (v)
          ((if v m2 m3) k) )) ) ) )
```

La dernière expression d'une séquence préserve l'environnement si la séquence le doit. On remarquera que la mécanique de restauration ne restaure l'environnement courant que s'il y a eu des applications susceptibles de l'avoir modifié. En particulier, une séquence comme `(begin a (car x) ...)` ne demande pas que l'environnement soit préservé pendant le calcul de `a` ou `(car x)` car il ne saurait être modifié.

```
(define (meaning-sequence e+ r tail?)
  (if (pair? e+)
      (if (pair? (cdr e+))
          (meaning*-multiple-sequence (car e+) (cdr e+) r tail?)
          (meaning*-single-sequence (car e+) r tail?) )
      (static-wrong "Illegal syntax: (begin)") ) )
(define (meaning*-single-sequence e r tail?)
  (meaning e r tail?) )
(define (meaning*-multiple-sequence e e+ r tail?)
  (let ((m1 (meaning e r #f))
        (m+ (meaning-sequence e+ r tail?)) )
    (lambda (k)
      (m1 (lambda (v)
            (m+ k) )) ) ) )
```

L'abstraction doit capturer l'environnement courant qui est l'environnement de naissance de la fermeture créée. Elle devra le restaurer pour l'étendre aux instants d'invocations. Le cas des fonctions d'arité variable est similaire à celui des fonctions d'arité fixe.

```
(define (meaning-fix-abstraction n* e+ r tail?)
  (let* ((arity (length n*))
         (arity+1 (+ arity 1))
         (r2 (r-extend* r n*))
         (m+ (meaning-sequence e+ r2 #t)) )
    (lambda (k)
      (let ((sr *env*))
        (k (lambda (v* k1)
             (if (= (activation-frame-argument-length v*) arity+1)
                 (begin (set! *env* (sr-extend* sr v*))
                        (m+ k1) )
                 (wrong "Incorrect arity") ) )) ) ) ) )
```

Le seul cas vraiment complexe est celui de l'application puisque c'est elle qui doit gérer si oui ou non il faut restaurer l'environnement après invocation.

```
(define (meaning-regular-application e e* r tail?)
  (let* ((m (meaning e r #f))
         (m* (meaning* e* r (length e*) #f)) )
    (if tail?
        (lambda (k)
          (m (lambda (f)
               (if (procedure? f)
                   (m* (lambda (v*)
                         (f v* k) ))
                   (wrong "Not a function" f) ) )) )
        (lambda (k)
          (m (lambda (f)
```

```
                (if (procedure? f)
                    (m* (lambda (v*)
                          (let ((sr *env*))          ; save environment
                            (f v* (lambda (v)
                                    (set! *env* sr)  ; restore environment
                                    (k v) )) ) ))
                    (wrong "Not a function" f) ) )) ) ) ) )
(define (meaning* e* r size tail?)
  (if (pair? e*)
      (meaning-some-arguments (car e*) (cdr e*) r size tail?)
      (meaning-no-argument r size tail?) ) )
(define (meaning-some-arguments e e* r size tail?)
  (let ((m (meaning e r #f))
        (m* (meaning* e* r size tail?))
        (rank (- size (+ (length e*) 1))) )
    (lambda (k)
      (m (lambda (v)
           (m* (lambda (v*)
                 (set-activation-frame-argument! v* rank v)
                 (k v*) ) )) )) ) )
(define (meaning-no-argument r size tail?)
  (let ((size+1 (+ size 1)))
    (lambda (k)
      (let ((v* (allocate-activation-frame size+1)))
        (k v*) ) ) ) )
```

Et voici achevé ce nouvel interprète à environnement mis en registre. Les environnements globaux initiaux sont les mêmes que les précédents.

6.2.1 Conclusion sur l'interprète à environnement mis en registre

Cette transformation n'est pas toujours intéressante. Si l'on voulait ajouter quelque parallélisme au langage, la variable globale `*env*` serait une ressource unique partagée entre toutes les tâches et nécessiterait d'être sauvegardée dans le contexte des tâches. D'un autre côté, le fait que l'environnement soit toujours disponible est avantageux pour ajouter un peu de réflexivité au langage car alors on peut imaginer des primitives y accédant.

Ce nouvel interprète n'est pas plus rapide que le précédent. Certes les invocations ne comportent plus qu'une seule variable au lieu de deux mais au prix d'une référence à une variable globale muable chaque fois que l'environnement doit être consulté. Elle a toutefois le mérite d'éclaircir les notions de position terminale et surtout de nous mener doucement vers notre troisième interprète ...

6.3 Dilution des continuations

Il n'est pas rare que le langage d'implantation procure des notions de continuation que l'on pourrait directement utiliser plutôt que de les manipuler explicitement comme dans l'interprète précédent. Cette situation n'est pas si idiote. Si l'on dispose d'un compilateur pour Scheme, l'obtention d'un interprète passe en général par l'écriture de celui-ci, en Scheme même, qui alors utilise la bibliothèque d'exécution et notamment le `call/cc` qui s'y trouve. D'autre part, si l'on ne compile que du Lisp,

les seules continuations dont on a besoin sont équivalentes aux `setjmp`/`longjmp` de la bibliothèque du langage C. Dans ces deux cas, `call/cc` peut être considéré comme un opérateur magique et il est alors totalement inutile de réifier partout les continuations. Les avoir toujours sous la main requiert en effet un extraordinaire taux d'allocation dont on se passerait bien si l'on cherche à augmenter la vitesse d'interprétation.

Ce troisième interprète va donc revenir à un style direct sans continuation explicite. Nous en profiterons pour apporter deux nouvelles modifications : *(i)* les fonctions seront maintenant explicitement représentées par des objets ad hoc ; *(ii)* nous ferons apparaître le résultat du prétraitement sous forme de combinateurs (calligraphiés en majuscules) rappelant les instructions d'une hypothétique machine virtuelle.

Nous reprendrons toutes les améliorations dont était doté le premier interprète (appel aux primitives, formes réductibles) puis redonnerons toutes les définitions.

6.3.1 Fermetures

Les fermetures seront comme habituellement représentées par des objets à deux champs, un champ pour leur code et un autre pour leur environnement de définition. Le protocole d'invocation sera adapté à cette représentation grâce à la fonction `invoke` :

```
(define-class closure Object
  ( code
    closed-environment
    ) )
(define (invoke f v*)
  (if (closure? f)
      ((closure-code f) v* (closure-closed-environment f))
      (wrong "Not a function" f) ) )
```

Le code des fermetures interprétées sera donc représenté par une fermeture à deux variables : l'une pour le bloc d'activation et l'autre pour l'environnement de définition ; ainsi ce dernier pourra-t-il être rendu disponible pour être étendu. Toute invocation de fermeture devra donc passer par la fonction d'invocation, la justement nommée : `invoke`.

6.3.2 Le prétraiteur

Le prétraitement des programmes est réalisé par la fonction `meaning` qui au lieu de retourner une fermeture `(lambda (k) ...)` renvoie maintenant `(lambda () ...)`, objet que l'on peut interpréter, dans le droit fil de la tradition de Forth, comme une adresse de code sur laquelle il n'y a plus qu'à sauter pour l'exécuter. On le voit : la suppression de toutes les variables permet d'obtenir des suspensions (des *thunks*) et il n'est pas impossible d'avoir un protocole d'invocation plus élaboré que l'actuel qui réalise une sorte de simple mais efficace `GOTO` **[☞ Ex. 6.6]** pour invoquer ces suspensions.

La fonction `meaning` apparaît page 198.

6.3.3 Citation

La citation est fidèle à elle-même mais apparaît comme encore plus lisible grâce au combinateur `CONSTANT` :

```
(define (meaning-quotation v r tail?)
  (CONSTANT v) )
(define (CONSTANT value)
  (lambda () value) )
```

6.3.4 Référence

Le prétraitement des variables consiste toujours à les catégoriser puis à leur associer le bon lecteur. Ces lecteurs seront engendrés par les combinateurs appropriés. L'utilisation de combinateurs allège cette définition et, partant, la rend plus claire.

```
(define (meaning-reference n r tail?)
  (let ((kind (compute-kind r n)))
    (if kind
        (case (car kind)
          ((local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (SHALLOW-ARGUMENT-REF j)
                 (DEEP-ARGUMENT-REF i j) ) ) )
          ((global)
           (let ((i (cdr kind)))
             (CHECKED-GLOBAL-REF i) ) )
          ((predefined)
           (let ((i (cdr kind)))
             (PREDEFINED i) ) ) )
        (static-wrong "No such variable" n) ) ) )
(define (SHALLOW-ARGUMENT-REF j)
  (lambda () (activation-frame-argument *env* j)) )
(define (PREDEFINED i)
  (lambda () (predefined-fetch i)) )
(define (DEEP-ARGUMENT-REF i j)
  (lambda () (deep-fetch *env* i j)) )
(define (GLOBAL-REF i)
  (lambda () (global-fetch i)) )
(define (CHECKED-GLOBAL-REF i)
  (lambda ()
    (let ((v (global-fetch i)))
      (if (eq? v undefined-value)
          (wrong "Uninitialized variable")
          v ) ) ) )
```

Il est à noter cependant que, dans le combinateur `CHECKED-GLOBAL-REF`, lorsque la variable est non initialisée et comme on ne dispose que de l'index de la variable dans le vecteur `sg.current`, il n'est plus possible d'indiquer simplement le nom de la variable erronée. Il faudrait pour cela conserver ce que l'on nomme classiquement une « table des symboles », ici la liste `g.current`, indiquant les noms des variables et les endroits où elles ont été rangées. **[☞ Ex. 6.1]** On pourrait alors, connaissant l'index, retrouver le nom de la variable fautive.

6.3.5 Alternative

L'alternative s'éclaire elle-aussi grâce au combinateur `ALTERNATIVE` qui prend en arguments le résultat du prétraitement de ses trois sous-formes.

```
(define (meaning-alternative e1 e2 e3 r tail?)
  (let ((m1 (meaning e1 r #f))
        (m2 (meaning e2 r tail?))
        (m3 (meaning e3 r tail?)) )
    (ALTERNATIVE m1 m2 m3) ) )
(define (ALTERNATIVE m1 m2 m3)
  (lambda () (if (m1) (m2) (m3))) )
```

6.3.6 Affectation

La structure de l'affectation est semblable à celle de la référence, excepté qu'il y a une sous-forme à évaluer et qui est fournie en argument à tous les combinateurs écrivains.

```
(define (meaning-assignment n e r tail?)
  (let ((m (meaning e r #f))
        (kind (compute-kind r n)) )
    (if kind
        (case (car kind)
          ((local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (SHALLOW-ARGUMENT-SET! j m)
                 (DEEP-ARGUMENT-SET! i j m) ) ) )
          ((global)
           (let ((i (cdr kind)))
             (GLOBAL-SET! i m) ) )
          ((predefined)
           (static-wrong "Immutable predefined variable" n) ) )
        (static-wrong "No such variable" n) ) ) )
(define (SHALLOW-ARGUMENT-SET! j m)
  (lambda () (set-activation-frame-argument! *env* j (m))) )
(define (DEEP-ARGUMENT-SET! i j m)
  (lambda () (deep-update! *env* i j (m))) )
(define (GLOBAL-SET! i m)
  (lambda () (global-update! i (m))) )
```

6.3.7 Séquence

La séquence s'exprime clairement en terme du combinateur `SEQUENCE` qui correspond à un `begin` binaire.

```
(define (meaning-sequence e+ r tail?)
  (if (pair? e+)
      (if (pair? (cdr e+))
          (meaning*-multiple-sequence (car e+) (cdr e+) r tail?)
          (meaning*-single-sequence (car e+) r tail?) )
      (static-wrong "Illegal syntax: (begin)") ) )
(define (meaning*-single-sequence e r tail?)
  (meaning e r tail?) )
```

```
(define (meaning*-multiple-sequence e e+ r tail?)
  (let ((m1 (meaning e r #f))
        (m+ (meaning-sequence e+ r tail?)) )
    (SEQUENCE m1 m+) ) )
(define (SEQUENCE m m+)
  (lambda () (m) (m+)) )
```

6.3.8 Abstraction

Les fermetures sont créées par les combinateurs `FIX-CLOSURE` ou `NARY-CLOSURE`. Les diffférences portent sur la vérification d'arité et la mise en liste des arguments excédentaires.

```
(define (meaning-abstraction nn* e+ r tail?)
  (let parse ((n* nn*)
              (regular '()) )
    (cond
     ((pair? n*) (parse (cdr n*) (cons (car n*) regular)))
     ((null? n*) (meaning-fix-abstraction nn* e+ r tail?))
     (else       (meaning-dotted-abstraction
                  (reverse regular) n* e+ r tail? )) ) ) )
(define (meaning-fix-abstraction n* e+ r tail?)
  (let* ((arity (length n*))
         (r2 (r-extend* r n*))
         (m+ (meaning-sequence e+ r2 #t)) )
    (FIX-CLOSURE m+ arity) ) )
(define (meaning-dotted-abstraction n* n e+ r tail?)
  (let* ((arity (length n*))
         (r2 (r-extend* r (append n* (list n))))
         (m+ (meaning-sequence e+ r2 #t)) )
    (NARY-CLOSURE m+ arity) ) )
(define (FIX-CLOSURE m+ arity)
  (let ((arity+1 (+ arity 1)))
    (lambda ()
      (define (the-function v* sr)
        (if (= (activation-frame-argument-length v*) arity+1)
            (begin (set! *env* (sr-extend* sr v*))
                   (m+) )
            (wrong "Incorrect arity") ) )
      (make-closure the-function *env*) ) ) )
(define (NARY-CLOSURE m+ arity)
  (let ((arity+1 (+ arity 1)))
    (lambda ()
      (define (the-function v* sr)
        (if (>= (activation-frame-argument-length v*) arity+1)
            (begin
              (listify! v* arity)
              (set! *env* (sr-extend* sr v*))
              (m+) )
            (wrong "Incorrect arity") ) )
      (make-closure the-function *env*) ) ) )
```

6.3.9 Application

Il ne reste plus qu'à traiter les applications. La nature de celles-ci est analysée par `meaning-application` qui reconnaît les formes réductibles, les appels aux primitives et les applications quelconques.

```
(define (meaning-application e e* r tail?)
  (cond ((and (symbol? e)
              (let ((kind (compute-kind r e)))
                (and (pair? kind)
                     (eq? 'predefined (car kind))
                     (let ((desc (get-description e)))
                       (and desc
                            (eq? 'function (car desc))
                            (or (= (length (cddr desc)) (length e*))
                                (static-wrong
                                 "Incorrect arity for primitive" e )
                                ) ) ) ) ) ) )
         (meaning-primitive-application e e* r tail?) )
        ((and (pair? e)
              (eq? 'lambda (car e)) )
         (meaning-closed-application e e* r tail?) )
        (else (meaning-regular-application e e* r tail?)) ) )
```

Les applications quelconques sont justiciables de quatre combinateurs seulement. Notez dans le combinateur `TR-REGULAR-CALL` la séquentialisation du calcul du terme fonctionnel et des arguments. Comme toujours, l'ordre d'évaluation est gauche-droite.

```
(define (meaning-regular-application e e* r tail?)
  (let* ((m (meaning e r #f))
         (m* (meaning* e* r (length e*) #f)) )
    (if tail? (TR-REGULAR-CALL m m*) (REGULAR-CALL m m*)) ) )
(define (meaning* e* r size tail?)
  (if (pair? e*)
      (meaning-some-arguments (car e*) (cdr e*) r size tail?)
      (meaning-no-argument r size tail?) ) )
(define (meaning-some-arguments e e* r size tail?)
  (let ((m (meaning e r #f))
        (m* (meaning* e* r size tail?))
        (rank (- size (+ (length e*) 1))) )
    (STORE-ARGUMENT m m* rank) ) )
(define (meaning-no-argument r size tail?)
  (ALLOCATE-FRAME size) )

(define (TR-REGULAR-CALL m m*)
  (lambda ()
    (let ((f (m)))
      (invoke f (m*)) ) ) )
(define (REGULAR-CALL m m*)
  (lambda ()
    (let* ((f (m))
           (v* (m*))
           (sr *env*)
           (result (invoke f v*)) )
      (set! *env* sr)
      result ) ) )
(define (STORE-ARGUMENT m m* rank)
```

```
  (lambda ()
    (let* ((v (m))
           (v* (m*)) )
      (set-activation-frame-argument! v* rank v)
      v* ) ) )
(define (ALLOCATE-FRAME size)
  (let ((size+1 (+ size 1)))
    (lambda ()
      (allocate-activation-frame size+1) ) ) )
```

6.3.10 Formes réductibles

Le prétraitement des formes réductibles, comportant une forme `lambda` explicite en position fonctionnelle, ajoute quatre nouveaux combinateurs. `CONS-ARGUMENT` permet de créer la liste des arguments excédentaires tandis que `ALLOCATE-DOTTED-FRAME` crée un bloc d'activation, tout comme `ALLOCATE-FRAME`, sauf que le champ supplémentaire est explicitement initialisé par `()` ce que l'on évite de faire dans `ALLOCATE-FRAME` pour des raisons de performance.

```
(define (meaning-dotted-closed-application n* n body e* r tail?)
  (let* ((m* (meaning-dotted* e* r (length e*) (length n*) #f))
         (r2 (r-extend* r (append n* (list n))))
         (m+ (meaning-sequence body r2 tail?)) )
    (if tail? (TR-FIX-LET m* m+)
        (FIX-LET m* m+) ) ) )
(define (meaning-dotted* e* r size arity tail?)
  (if (pair? e*)
      (meaning-some-dotted-arguments (car e*) (cdr e*)
                                     r size arity tail? )
      (meaning-no-dotted-argument r size arity tail?) ) )
(define (meaning-some-dotted-arguments e e* r size arity tail?)
  (let ((m (meaning e r #f))
        (m* (meaning-dotted* e* r size arity tail?))
        (rank (- size (+ (length e*) 1))) )
    (if (< rank arity) (STORE-ARGUMENT m m* rank)
        (CONS-ARGUMENT m m* arity) ) ) )
(define (meaning-no-dotted-argument r size arity tail?)
  (ALLOCATE-DOTTED-FRAME arity) )
(define (FIX-LET m* m+)
  (lambda ()
    (set! *env* (sr-extend* *env* (m*)))
    (let ((result (m+)))
      (set! *env* (environment-next *env*))
      result ) ) )
(define (TR-FIX-LET m* m+)
  (lambda ()
    (set! *env* (sr-extend* *env* (m*)))
    (m+) ) )
(define (CONS-ARGUMENT m m* arity)
  (lambda ()
    (let* ((v (m))
           (v* (m*)) )
      (set-activation-frame-argument!
       v* arity (cons v (activation-frame-argument v* arity)) )
      v* ) ) )
```

```
(define (ALLOCATE-DOTTED-FRAME arity)
  (let ((arity+1 (+ arity 1)))
    (lambda ()
      (let ((v* (allocate-activation-frame arity+1)))
        (set-activation-frame-argument! v* arity '())
        v* ) ) ) )
```

Le combinateur `FIX-LET` doit restaurer l'environnement courant, ce qui implique de devoir le mémoriser quelque part afin de le restaurer ultérieurement. Une solution élégante existe ici puisqu'apparaissant dans le chaînage des blocs d'activation, il suffit d'aller l'y chercher.

6.3.11 Appel aux primitives

Le dernier type, et non le moins important en nombre, est le cas des formes où l'on a une variable globale non modifiable en position fonctionnelle. Dans ce cas on appellera le bon invoqueur paramétré par l'arité afin de ne plus avoir à la vérifier.

```
(define (meaning-primitive-application e e* r tail?)
  (let* ((desc (get-description e))
         ;; desc = (function address . variables-list)
         (address (cadr desc))
         (size (length e*)) )
    (case size
      ((0) (CALL0 address))
      ((1)
       (let ((m1 (meaning (car e*) r #f)))
         (CALL1 address m1) ) )
      ((2)
       (let ((m1 (meaning (car e*) r #f))
             (m2 (meaning (cadr e*) r #f)) )
         (CALL2 address m1 m2) ) )
      ((3)
       (let ((m1 (meaning (car e*) r #f))
             (m2 (meaning (cadr e*) r #f))
             (m3 (meaning (caddr e*) r #f)) )
         (CALL3 address m1 m2 m3) ) )
      (else (meaning-regular-application e e* r tail?)) ) ) )
(define (CALL0 address)
  (lambda () (address)) )
(define (CALL3 address m1 m2 m3)
  (lambda () (let* ((v1 (m1))
                    (v2 (m2)) )
               (address v1 v2 (m3)) )) )
```

Notez la séquentialisation explicite des arguments de `CALL3` pour respecter l'ordre gauche-droite.

6.3.12 Lancement de l'interprète

Étant donné que les continuations ne sont plus explicites, il nous faut revoir les macros enrichissant l'environnement global. Leur stucture a peu changé et voici juste `defprimitive2` à titre d'exemple.

```
(define-syntax defprimitive2
  (syntax-rules ()
```

```
((defprimitive2 name value)
 (definitial name
   (letrec ((arity+1 (+ 2 1))
            (behavior
             (lambda (v* sr)
               (if (= (activation-frame-argument-length v*)
                      arity+1 )
                   (value (activation-frame-argument v* 0)
                          (activation-frame-argument v* 1) )
                   (wrong "Incorrect arity" 'name) ) ) ) )
     (description-extend! 'name `(function ,value a b))
           (make-closure behavior sr.init) ) ) ) ) )
```

On lancera l'interprète par :

```
(define (chapter63-interpreter)
  (define (toplevel)
    (set! *env* sr.init)
    (display ((meaning (read) r.init #t)))
    (toplevel) )
  (toplevel) )
```

6.3.13 La fonction `call/cc`

Puisque maintenant la fonction `call/cc` est magique, elle nécessite pour sa définition la fonction `call/cc` de la bibliothèque sur laquelle l'interprète s'appuie. On retrouve donc ici les définitions tautologiques des premiers interprètes.

```
(definitial call/cc
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (make-closure
     (lambda (v* sr)
       (if (= arity+1 (activation-frame-argument-length v*))
           (call/cc
            (lambda (k)
              (invoke
               (activation-frame-argument v* 0)
               (let ((frame (allocate-activation-frame (+ 1 1))))
                 (set-activation-frame-argument!
                  frame 0
                  (make-closure
                   (lambda (values r)
                     (if (= (activation-frame-argument-length values)
                            arity+1 )
                         (k (activation-frame-argument values 0))
                         (wrong "Incorrect arity" 'continuation) ) )
                   sr.init ) )
                 frame ) ) ) )
           (wrong "Incorrect arity" 'call/cc) ) )
     sr.init ) ) )
```

6.3.14 La fonction `apply`

Pour parachever notre tour d'horizon, voici la fonction `apply` qui est toujours difficile à écrire car elle dépend fortement du codage des fonctions et du protocole d'appel élu. La voici, dans tous ses détails et son entière complexité :

```
(definitial apply
  (let* ((arity 2)
         (arity+1 (+ arity 1)) )
    (make-closure
     (lambda (v* sr)
       (if (>= (activation-frame-argument-length v*) arity+1)
           (let* ((proc (activation-frame-argument v* 0))
                  (last-arg-index
                   (- (activation-frame-argument-length v*) 2) )
                  (last-arg
                   (activation-frame-argument v* last-arg-index) )
                  (size (+ last-arg-index (length last-arg)))
                  (frame (allocate-activation-frame size)) )
             (do ((i 1 (+ i 1)))
                 ((= i last-arg-index))
               (set-activation-frame-argument!
                frame (- i 1) (activation-frame-argument v* i) ) )
             (do ((i (- last-arg-index 1) (+ i 1))
                  (last-arg last-arg (cdr last-arg)) )
                 ((null? last-arg))
               (set-activation-frame-argument!
                frame i (car last-arg) ) )
             (invoke proc frame) )
           (wrong "Incorrect arity" 'apply) ) )
     sr.init ) ) )
```

Cette primitive vérifie qu'elle a au moins deux arguments, parcourt le bloc d'activation pour connaître le nombre exact d'arguments sur lesquels appliquer la fonction de premier argument : il faut pour cela calculer la longueur de la liste de dernier argument. Muni de ce nombre, on peut enfin allouer un bloc d'activation de la bonne taille dans lequel on recopie tous les arguments à leur bonne position, les premiers par recopie directe du bloc d'activation fourni à `apply`, les suivants par exploration de la liste des arguments superflus. L'exploration se termine lorsque la liste devient vide, ce qui est testé par `null?`. Il pourrait sembler plus robuste d'utiliser `atom?` mais cela rendrait correct le programme `(apply list '(a b . c))`, ce qui n'est pas conforme à la norme de Scheme.

On le voit : `apply` n'est pas une opération peu coûteuse puisqu'elle alloue un nouveau bloc d'activation et nécessite de parcourir la liste de dernier argument.

6.3.15 Conclusions sur l'interprète sans continuation

Ce nouvel interprète est environ deux à quatre fois plus rapide que le précédent et ce principalement du fait de la non-réification des continuations. En effet, représenter les continuations par des fermetures les confond avec les autres valeurs plus ordinaires. C'est faire fi d'une propriété importante des continuations qui est leur habituelle très faible durée de vie. C'est pour cela que l'allocation en pile des continuations est en général gagnante car elle a un faible coût. C'est aussi le cas de la désallocation, typiquement une seule instruction baissant le pointeur de sommet de pile. Que l'on ne s'y trompe pas : la masse de données allouées pour représenter les continuations est du même ordre de grandeur, que ce soit dans la pile ou dans le tas, mais sa gestion est de loin moins coûteuse en pile qu'en tas (voir aussi [App87] pour un avis divergent mais ne prenant pas en compte les effets de localité), encore que l'on puisse envisager de

spécialiser le tas en plusieurs zones dont l'une adaptée à la gestion des continuations comme décrit en [MB93].

Les suspensions qu'utilise le dernier interprète l'apparentent à la technique de *threaded code* [Bel73] usuelle à Forth [Hon93]. Cet interprète s'inspire fortement aussi de celui de Marc Feeley et Guy Lapalme [FL87].

Les combinateurs jouent très exactement le rôle de générateur de code. Ils ont été introduits dans cet interprète car ils fournissent une interface simple permettant de changer aisément la représentation des programmes prétraités. On peut tout à fait envisager de fabriquer des objets plutôt que des fermetures et l'on **[☞ Ex. 6.3]** verra dans un prochain chapitre l'usage que nous en ferons pour compiler.

6.4 Conclusions

À première vue ce chapitre et les trois interprètes qu'il propose peuvent apparaître comme une gigantesque régression annihilant successivement tous les progrès effectués lors des quatre premiers chapitres : on a quasiment supprimé la mémoire (sauf en ce qui concerne la gestion des blocs d'activation) et l'on a fait disparaître les continuations. En revanche, on a présenté la notion de prétraitement comme introduction à la compilation et à la séparation statique/dynamique ainsi que diverses améliorations visant à augmenter la vitesse d'exécution. En ce qui concerne ce dernier but, il est tout à fait atteint puisque l'on a pu augmenter cette vitesse d'environ deux ordres de grandeur par rapport à l'interprète dénotationnel.

L'interprète final implante toutes les formes spéciales de Scheme et représente ainsi l'essence de tout langage. Il suffit seulement de doter l'interprète d'un véritable gestionnaire de mémoire et des bibliothèques ad hoc pour le particulariser pour l'édition de texte, le dessin industriel, la recherche de primalité, le langage Scheme, le test de matériels, la réalité virtuelle, etc. Le « il suffit seulement » précédent n'est qu'une litote masquant l'effroyable complexité du choix d'un schéma de représentation des objets primitifs en mémoire dont le but principal est d'autoriser des tests de type rapides [Gud93] et un GC non moins efficace.

On peut bien sûr encore améliorer ces interprètes ou les étendre pour traiter de nouvelles formes spéciales. Rappelons que le but n'est que de procurer une suite d'interprètes incrémentiellement modifiés de manière à réduire la masse de détails dont ils sont composés et permettre la seule description des nouveaux buts dont ils sont l'illustration.

6.5 Exercices

Exercice 6.1 : [☞ p. 203] Modifiez le combinateur `CHECKED-GLOBAL-REF` afin que le message d'erreur en cas de variable non initialisée soit significatif.

Exercice 6.2 : Définissez élégamment, sans effort et pour le troisième interprète la primitive `list`.

Exercice 6.3 : Au lieu de prétraiter un programme, il pourrait être intéressant de visualiser en quoi il est prétraité. Ce qui suit est la visualisation de la factorielle :

```
? (disassemble '(lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
= (FIX-CLOSURE
    (ALTERNATIVE
      (CALL2 #<=> (SHALLOW-ARGUMENT-REF 0) (CONSTANT 0))
      (CONSTANT 1)
      (CALL2 #<*> (SHALLOW-ARGUMENT-REF 0)
                  (REGULAR-CALL
                  (CHECKED-GLOBAL-REF 10) ;←  fact
                  (STORE-ARGUMENT
                    (CALL2 #<-> (SHALLOW-ARGUMENT-REF 0)
                                (CONSTANT 1) )
                    (ALLOCATE-FRAME 1)
                    0 ) ) ) )
  1 )
```

Écrivez un tel visualisateur.

Exercice 6.4 : Modifiez le dernier interprète afin que les blocs d'activation soient alloués avant le calcul des arguments, arguments qui pourront ainsi être directement rangés à leur bonne place. **[☞ p. 181]**

Exercice 6.5 : Définissez une forme spéciale `redefine` qui insère dans **[☞ p. 183]** l'environnement global modifiable une variable déjà présente dans l'environnement global immuable. La valeur initiale de cette nouvelle globale est la valeur qu'elle possédait auparavant.

Exercice 6.6 : Améliorez le prétraitement des fonctions sans variable. **[☞ p. 202]**

6.6 Lectures recommandées

L'interprète final de ce chapitre est inspiré de [FL87] tandis que la méthode elle-même, visant à faire apparaître des combinateurs à partir de l'interprète dénotationnel, provient de [Cli84]. Enfin les amateurs d'interprètes vraiment rapides méditeront les thèses ou livres suivants : [Cha80] et [SJ93].

7
Compilation

Le chapitre précédent avait exposé un procédé de prétraitement transcrivant un programme écrit en Scheme vers un langage arborescent doté de quelque vingt-cinq instructions. Le présent chapitre exploite le résultat de ce prétraitement et le transforme en une suite d'octets, langage d'une machine ad hoc. Seront successivement analysées : la définition d'une machine virtuelle, la compilation vers son langage propre et l'implantation de diverses extensions de Scheme comme les échappements, les variables dynamiques et un mécanisme d'exception.

(SHALLOW-ARGUMENT-REF j)	(PREDEFINED i)
(DEEP-ARGUMENT-REF i j)	(SHALLOW-ARGUMENT-SET! j m)
(DEEP-ARGUMENT-SET! i j m)	(GLOBAL-REF i)
(CHECKED-GLOBAL-REF i)	(GLOBAL-SET! i m)
(CONSTANT v)	(ALTERNATIVE $m1$ $m2$ $m3$)
(SEQUENCE m $m+$)	(TR-FIX-LET $m*$ $m+$)
(FIX-LET $m*$ $m+$)	(CALL0 $address$)
(CALL1 $address$ $m1$)	(CALL2 $address$ $m1$ $m2$)
(CALL3 $address$ $m1$ $m2$ $m3$)	(FIX-CLOSURE $m+$ $arity$)
(NARY-CLOSURE $m+$ $arity$)	(TR-REGULAR-CALL m $m*$)
(REGULAR-CALL m $m*$)	(STORE-ARGUMENT m $m*$ $rank$)
(CONS-ARGUMENT m $m*$ $arity$)	(ALLOCATE-FRAME $size$)
(ALLOCATE-DOTTED-FRAME $arity$)	

TABLE 7.1 – Langage intermédiaire à 25 instructions.

Le tableau 7.1 présente les 25 instructions du langage intermédiaire. Dans ce tableau, m, $m1$, $m2$, $m3$, $m+$ et v sont des valeurs ; $m*$ retourne un bloc d'activation ; $rank$, $arity$, $size$, i et j sont des entiers naturels ; enfin $address$ représente une fonction prédéfinie (subr) qui prend des valeurs et en retourne une.

La compilation évoque souvent la production d'une suite d'instructions d'un assez bas niveau. Ce n'est pas le cas du résultat du prétraitement décrit dans le chapitre précédent qui fabrique l'équivalent d'un arbre structuré. Pour donner un exemple un peu plus parlant, considérez le programme suivant :

```
((lambda (fact) (fact 5 fact (lambda (x) x)))
 (lambda (n f k) (if (= n 0) (k 1)
                     (f (- n 1) f (lambda (r) (k (* n r)))) )) )
```

Sa transcription, après prétraitement, est la suivante :

```
(TR-FIX-LET
 (STORE-ARGUMENT
  (FIX-CLOSURE
   (ALTERNATIVE
    (CALL2 #<=> (SHALLOW-ARGUMENT-REF 0) (CONSTANT 0))
    (TR-REGULAR-CALL (SHALLOW-ARGUMENT-REF 2)
                     (STORE-ARGUMENT (CONSTANT 1)
                                     (ALLOCATE-FRAME 1)
                                     0 ) )
    (TR-REGULAR-CALL (SHALLOW-ARGUMENT-REF 1)
     (STORE-ARGUMENT (CALL2 #<-> (SHALLOW-ARGUMENT-REF 0)
                                 (CONSTANT 1) )
      (STORE-ARGUMENT (SHALLOW-ARGUMENT-REF 1)
       (STORE-ARGUMENT (FIX-CLOSURE
                        (TR-REGULAR-CALL (DEEP-ARGUMENT-REF 1 2)
                         (STORE-ARGUMENT (CALL2 #<*>
                                         (DEEP-ARGUMENT-REF 1 0)
                                         (SHALLOW-ARGUMENT-REF 0) )
                          (ALLOCATE-FRAME 1)
                          0 ) )
                        1 )
        (ALLOCATE-FRAME 3)
        2 )
       1 )
      0 ) ) )
 3 )
 (ALLOCATE-FRAME 1)
 0)
(TR-REGULAR-CALL (SHALLOW-ARGUMENT-REF 0)
 (STORE-ARGUMENT (CONSTANT 5)
  (STORE-ARGUMENT (SHALLOW-ARGUMENT-REF 0)
   (STORE-ARGUMENT (FIX-CLOSURE (SHALLOW-ARGUMENT-REF 0) 1)
     (ALLOCATE-FRAME 3)
    2 )
   1 )
  0 ) ) )
```

Ce n'est pas la plus lisible des formes mais elle est précise. Le but de ce chapitre est de montrer que cette forme est loin d'être ultime et que certaines transformations (linéarisation, codage en octets) peuvent encore y être apportées qui la transmueront en d'autres langages. Le langage des 25 instructions/générateurs (Tableau 7.1) va donc nous servir de langage intermédiaire, véritable tremplin vers d'autres mondes.

Nous considérerons la phase de prétraitement (le dernier interprète du chapitre précédent produisant ce fameux langage intermédiaire) comme la première passe d'un compilateur. Nous ne nous intéresserons donc qu'à ces 25 fonctions/générateurs que nous adapterons aux caractéristiques du langage cible final. Cette division du travail et surtout l'exécutabilité du langage intermédiaire nous a déjà permis de tester le prétraiteur aussi ne nous focaliserons-nous plus que sur cette seule seconde passe.

Nous étudierons, d'abord, la compilation vers une machine virtuelle, simple mais présentant toutes les caractéristiques d'une machine programmable en langage machine. Son jeu d'instructions sera représenté par des octets, des entiers de 0 à 255. Cette technique, dite de *bytecode*, apparue avant 1980 [Deu80, Row90], a souvent été utilisée pour exposer les rudiments de la compilation [Hen80]. Le code obtenu est

particulièrement compact, ce qui justifie son utilisation sur des machines à faible mémoire ou cache étroit ; c'est la technique utilisée par PC-Scheme [BJ86] et Caml Light [LW93].

La technique entière combine plusieurs aspects. Après prétraitement, un programme est compilé en un vecteur d'octets ; celui-ci est ensuite évalué par un interprète. Il y a donc à la fois compilation et interprétation mais seul l'interprète d'octets est nécessaire pour l'exécution.

7.1 Compilation vers des octets

Notre but est de faire émerger peu à peu une machine spécialisée dans l'interprétation d'octets. La définition de cette machine est fournie par les définitions des quelque vingt-cinq instructions du langage intermédiaire. Si certaines parmi celles-ci sont immédiates, d'autres vont requérir des traitements plus inventifs. Ce qui par contre va nous aider est que nous pouvons à la fois imaginer la machine et son jeu d'instructions, indispensable souplesse qui nous permettra, à l'occasion, d'adjoindre un nouveau registre ou de découvrir une pile.

Pour obtenir en final un vecteur d'octets, il faut linéariser le programme exprimé en langage intermédiaire. Il faut donc faire en sorte que les communications entre instructions soient limitées aux ressources communes de la machine, c'est-à-dire les registres et la pile. Pour l'instant notre machine ne dispose que d'un seul registre contenant l'environnement lexical courant `*env*`. Nous allons bientôt étoffer cette architecture quelque peu spartiate.

7.1.1 Introduction du registre `*val*`

Parmi les 25 instructions du langage intermédiaire, certaines produisent des valeurs comme, par exemple, l'instruction `SHALLOW-ARGUMENT-REF` ou `CONSTANT`, tandis que d'autres coordonnent des calculs comme `FIX-LET` ou `ALTERNATIVE`. Considérons, de plus près, le cas particulier de `GLOBAL-SET!` qui était ainsi définie :

```
(define (GLOBAL-SET! i m)
  (lambda () (global-update! i (m))) )
```

Rompre les communications requiert d'imaginer un nouveau registre, que l'on nommera `*val*`, où les producteurs de valeurs déposeront celles-ci afin que leurs consommateurs puissent l'y trouver. Un producteur comme `CONSTANT` se réécrira donc comme :

```
(define (CONSTANT value)
  (lambda ()
    (set! *val* value) ) )
```

Tandis que le consommateur `GLOBAL-SET!` deviendra :

```
(define (GLOBAL-SET! i m)
  (lambda ()
    (m)
    (global-update! i *val*) ) )
```

La forme `(m)` finira par produire une valeur dans le registre `*val*` qui sera alors transféré par `global-update!` à la bonne adresse. On notera que `global-update!`

ne perturbe pas le registre `*val*` (ce qu'il y a de plus efficace, c'est de n'engendrer aucune instruction), la valeur d'une forme d'affectation d'une variable globale est, par conséquent, la valeur se trouvant dans le registre `*val*`, c'est-à-dire la valeur affectée.

Le reste de la transformation se déduit facilement des deux exemples donnés plus haut. Par exemple, `SEQUENCE` se trouve automatiquement linéarisé en :

```
(define (SEQUENCE m m+)
  (lambda () (m) (m+)) )
```

Tandis que `FIX-LET` devient :

```
(define (FIX-LET m* m+)
  (lambda ()
    (m*)
    (set! *env* (sr-extend* *env* *val*))
    (m+)
    (set! *env* (activation-frame-next *env*)) ) )
```

7.1.2 Invention de la pile

Une partie de notre désir de linéarisation se trouve donc exaucée. Mais les choses sont un peu plus délicates en ce qui concerne, par exemple, `STORE-ARGUMENT` qui est devenu :

```
(define (STORE-ARGUMENT m m* rank)
  (lambda ()
    (m)
    (let ((v *val*))
      (m*)
      (set-activation-frame-argument! *val* rank v) ) ) )
```

Cette instruction utilise la forme `let` pour sauvegarder des valeurs et les restaurer si besoin est. La forme `let` sauve ici une valeur dans le registre anonyme `v` pendant le calcul de `(m*)`. On ne peut associer un vrai registre de la machine à `v` car on peut avoir besoin de plus d'un `v` simultanément et notamment dans le cas de multiples formes `STORE-ARGUMENT` imbriquées dans le calcul de `(m*)`. Il nous faut donc une zone pouvant sauvegarder un nombre quelconque de valeurs. Une pile ferait bien l'affaire surtout que les empilements/dépilements semblent, ici, bien équilibrés. Nous supposerons donc disposer d'une pile définie et gérée par les fonctions suivantes :

```
(define *stack* (make-vector 1000))
(define *stack-index* 0)
(define (stack-push v)
  (vector-set! *stack* *stack-index* v)
  (set! *stack-index* (+ *stack-index* 1)) )
(define (stack-pop)
  (set! *stack-index* (- *stack-index* 1))
  (vector-ref *stack* *stack-index*) )
```

Doté de cette nouvelle conquête technologique, l'instruction `STORE-ARGUMENT` peut être adaptée pour en user immodérément. Au lieu de sauvegarder en registre, on sauve dans la pile et c'est de là que l'on réexhume ces valeurs. Ce schéma ne convient que parce que l'on assurera que la pile que prend `(m*)` sera celle-là même que rendra `(m*)`. C'est donc un invariant que l'on devra respecter lors de l'écriture des instructions.

```
(define (STORE-ARGUMENT m m* rank)
  (lambda ()
```

```
    (m)
    (stack-push *val*)
    (m*)
    (let ((v (stack-pop)))
      (set-activation-frame-argument! *val* rank v) ) ) )
```

Le cas de `REGULAR-CALL` est apparenté sauf qu'il faut à la fois préserver la fonction à invoquer (lors du calcul de ses arguments) et l'environnement courant lors de l'invocation proprement dite. On avait :

```
(define (REGULAR-CALL m m*)
  (lambda ()
    (m)
    (let ((f *val*))
      (m*)
      (let ((sr *env*))
        (invoke f *val*)
        (set! *env* sr) ) ) ) )
```

On peut maintenant transformer cette définition, après l'invention d'un nouveau registre `*fun*`, en :

```
(define (REGULAR-CALL m m*)
  (lambda ()
    (m)
    (stack-push *val*)
    (m*)
    (set! *fun* (stack-pop))
    (stack-push *env*)
    (invoke *fun*)
    (set! *env* (stack-pop)) ) )
```

On remarquera, au passage, que nous avons redéfini le protocole d'appel fonctionnel pour ne plus prendre le bloc d'activation en argument puisqu'il est déjà dans le registre `*val*`. C'est ce que l'on peut voir dans la version courante de `FIX-CLOSURE` :

```
(define (FIX-CLOSURE m+ arity)
  (let ((arity+1 (+ arity 1)))
    (lambda ()
      (define (the-function sr)
        (if (= (activation-frame-argument-length *val*) arity+1)
            (begin (set! *env* (sr-extend* sr *val*))
                   (m+) )
            (wrong "Incorrect arity") ) )
      (set! *val* (make-closure the-function *env*)) ) ) )
```

L'ajout de registres peut aussi servir à linéariser l'appel aux primitives. On introduira les registres `*arg1*` et `*arg2*` (dont l'un d'entre eux n'est pas nécessairement différent de `*fun*` qui n'est jamais simultanément utilisé). Alors `CALL3` s'écrira :

```
(define (CALL3 address m1 m2 m3)
  (lambda ()
    (m1)
    (stack-push *val*)
    (m2)
    (stack-push *val*)
    (m3)
    (set! *arg2* (stack-pop))
    (set! *arg1* (stack-pop))
    (set! *val* (address *arg1* *arg2* *val*)) ) )
```

7.1.3 Individualisation des instructions

Actuellement les 25 instructions que nous redéfinissons engendrent des suspensions dont le corps est une séquence d'effets sur des registres. Faire apparaître des instructions, c'est transformer ces 25 instructions pour qu'elles engendrent des séquences de suspensions n'ayant qu'un unique effet : modifier un registre, empiler une valeur, etc. Cette inversion de point de vue permettra de donner un sens à la notion de compteur ordinal : un registre spécialisé désignant la prochaine instruction à exécuter. Le compteur ordinal autorisera aussi de décrire très précisément le protocole d'appel fonctionnel ainsi que, par la suite, le mystère de l'implantation de `call/cc`.

Linéarisation des affectations

Prenons le cas de `SHALLOW-ARGUMENT-SET!`, cette instruction était définie comme :

```
(define (SHALLOW-ARGUMENT-SET! j m)
  (lambda ()
    (m)
    (set-activation-frame-argument! *env* j *val*) ) )
```

La transformer en une séquence d'instructions conduit à la réécrire sous la forme des deux fonctions suivantes :

```
(define (SHALLOW-ARGUMENT-SET! j m)
  (append m (SET-SHALLOW-ARGUMENT! j)) )
(define (SET-SHALLOW-ARGUMENT! j)
  (list (lambda () (set-activation-frame-argument! *env* j *val*))) )
```

La première, de même nom qu'auparavant, compose les différents effets et renvoie la liste des instructions finales. La seconde `SET-SHALLOW-ARGUMENT!` est spécialisée dans l'écriture dans un bloc d'activation.

Linéarisation des invocations

Un bel exemple de linéarisation est fourni par `REGULAR-CALL`. Individualiser tous ses composants conduit à ajouter les instructions suivantes à la machine finale : `PUSH-VALUE`, `POP-FUNCTION`, `PRESERVE-ENV`, `FUNCTION-INVOKE` et `RESTORE-ENV`.

Voici ces nouvelles définitions :

```
(define (REGULAR-CALL m m*)
  (append m (PUSH-VALUE)
          m* (POP-FUNCTION) (PRESERVE-ENV)
             (FUNCTION-INVOKE) (RESTORE-ENV)
          ) )
(define (PUSH-VALUE)
  (list (lambda () (stack-push *val*))) )
(define (POP-FUNCTION)
  (list (lambda () (set! *fun* (stack-pop)))) )
(define (PRESERVE-ENV)
  (list (lambda () (stack-push *env*))) )
(define (FUNCTION-INVOKE)
  (list (lambda () (invoke *fun*))) )
(define (RESTORE-ENV)
  (list (lambda () (set! *env* (stack-pop)))) )
```

Le résultat de la compilation est maintenant, conformément à nos souhaits, une liste d'instructions élémentaires. Mais cette liste n'est plus directement exécutable, il nous faut donc fournir un moteur pour évaluer cette liste d'instructions ; ce sera la fonction `run`.

```
(define (run)
  (let ((instruction (car *pc*)))
    (set! *pc* (cdr *pc*))
    (instruction)
    (run) ) )
```

Le résultat de la compilation sera une liste d'instructions stockées dans la variable `*pc*` jouant le rôle d'un compteur ordinal. La fonction `run` simule un processeur qui lit une instruction, incrémente le compteur ordinal (ici toutes les instructions ont une même taille puisqu'elles sont toutes représentées par des fermetures de même signature `(lambda () ...)`), exécute l'instruction et recommence sempiternellement.

Linéarisation de l'alternative

L'alternative est toujours un problème pour la linéarisation car, s'il n'y a qu'une seule entrée, il y a deux sorties possibles. Et comment linéariser une patte d'oie ? Pour cela nous (ré)inventerons deux nouvelles instructions de saut affectant le compteur ordinal :`JUMP-FALSE` et `GOTO`. La dernière représente le saut inconditionnel [1] tandis que la première teste le contenu du registre `*val*` et ne saute que s'il contient Faux. L'une et l'autre de ces instructions de saut n'affectent que le registre `*pc*` à l'exclusion de tout autre.

```
(define (JUMP-FALSE i)
  (list (lambda () (if (not *val*) (set! *pc* (list-tail *pc* i))))) )
(define (GOTO i)
  (list (lambda () (set! *pc* (list-tail *pc* i)))) )
```

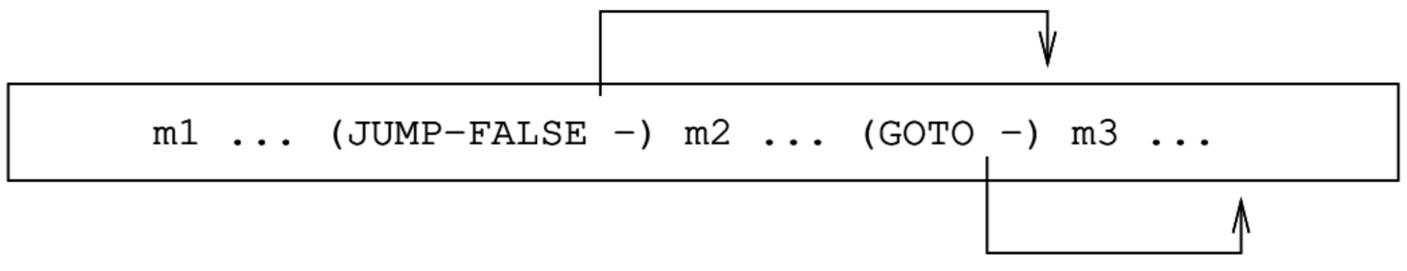


FIGURE 7.1 – Linéarisation de l'alternative

Voici donc avec ces deux nouvelles instructions comment linéariser l'alternative ce que l'on peut mieux visualiser à la figure 7.1.

```
(define (ALTERNATIVE m1 m2 m3)
  (append m1 (JUMP-FALSE (+ 1 (length m2)))
          m2 (GOTO (length m3))
          m3 ) )
```

La condition est calculée puis testée par `JUMP-FALSE`. Si la condition est vraie, on exécute les instructions qui suivent et à la fin de celles-ci on sautera les instructions correspondant à l'alternant. Si la condition était fausse, on sautera sur l'alternant que l'on exécutera. On le voit : on est parvenu au niveau d'un langage d'assemblage. On notera toutefois que nos sauts sont des sauts relatifs à la position courante.

1. Bien connu, sous le même nom, de beaucoup d'autres langages.

Linéarisation des abstractions

La dernière instruction difficile à linéariser est la création de fermetures car il y a à la fois à épisser le code de la fonction avec celui de qui la crée. Nous utiliserons encore une fois un saut pour ce faire (Figure 7.2). Voici la création de fermeture d'arité variable :

```
(define (NARY-CLOSURE m+ arity)
  (define the-function
    (append (ARITY>=? (+ arity 1)) (PACK-FRAME! arity) (EXTEND-ENV)
            m+ (RETURN) ) )
  (append (CREATE-CLOSURE 1) (GOTO (length the-function))
          the-function ) )
(define (CREATE-CLOSURE offset)
  (list (lambda () (set! *val* (make-closure (list-tail *pc* offset)
                                             *env* )))) )
(define (PACK-FRAME! arity)
  (list (lambda () (listify! *val* arity))) )
```

La nouvelle instruction `CREATE-CLOSURE` fabrique une fermeture dont le code se trouve à une instruction (un `GOTO`) de distance. Lorsque la fermeture a été créée (et mise dans le registre `*val*`) son créateur saute par dessus le code correspondant à son corps afin de poursuivre en séquence.

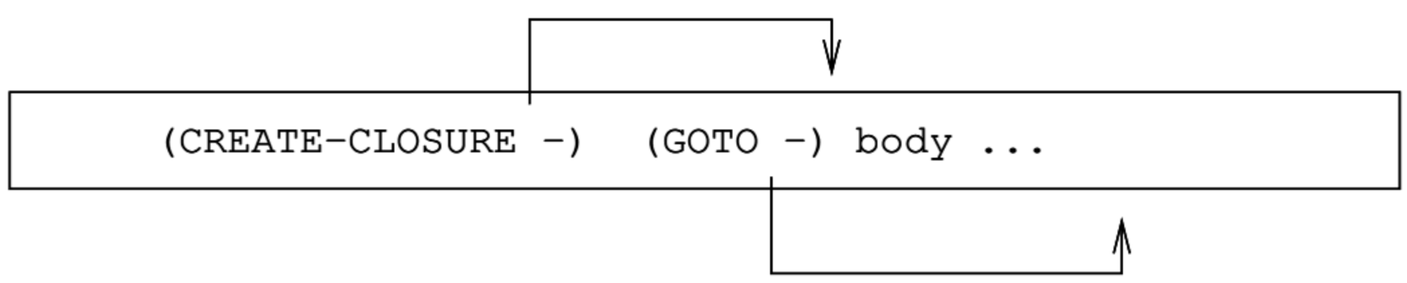


FIGURE 7.2 – Linéarisation de l'abstraction

7.1.4 Protocole d'appel fonctionnel

Les fonctions sont invoquées par les instructions `TR-REGULAR-CALL` ou `REGULAR-CALL` que nous avons détaillées il y a peu. [☞ p. 218] La fonction `invoke` condense le protocole d'appel fonctionnel.

```
(define (invoke f)
  (cond ((closure? f)
         (stack-push *pc*)
         (set! *env* (closure-closed-environment f))
         (set! *pc* (closure-code f)) )
        ... ) )
```

Pour invoquer une fonction, on sauvegarde le compteur ordinal désignant l'instruction qui suit l'instruction `(FUNCTION-INVOKE)` chez l'appelant, puis on désosse la fermeture pour remplir le registre d'environnement `*env*` avec son environnement de définition. On affecte enfin le compteur ordinal avec l'adresse de la première instruction de son corps. Notez que l'on ne préserve pas l'environnement courant puisqu'il est déjà traité par ailleurs selon que l'on invoque la fonction par `REGULAR-CALL` ou `TR-REGULAR-CALL`. La fonction `run` prend alors la main et exécute la première instruction du corps de la fonction invoquée qui vérifie son arité puis, en cas de succès, étend

l'environnement avec son bloc d'activation présent dans le registre `*val*`. Tout est maintenant en place pour que la fonction évalue son vrai corps. Une valeur est donc calculée puis placée dans `*val*`. Il faut maintenant la transmettre à l'appelant : c'est le rôle de l'instruction `RETURN` que de revenir à l'appelant en dépilant le compteur ordinal de retour présent au sommet de pile.

```
(define (RETURN)
  (list (lambda () (set! *pc* (stack-pop)))) )
```

Remarques

Les habitués des langages d'assemblage remarqueront que les sauts s'effectuent tous en avant ce qui permet d'éviter de gérer des étiquettes de saut et de calculer rétroactivement les distances à sauter. D'autre part, les sauts (ainsi que la fabrication des fermetures) sont tous relatifs au compteur ordinal ce qui produit du code indépendant de sa position en mémoire. C'est ce que l'on nomme du PIC (pour *PC-independent code*).

7.2 Langage et machine cible

Nous avons donc maintenant défini notre machine cible ainsi que le langage avec lequel on la programme. La machine comporte cinq registres `*env*`, `*val*`, `*fun*`, `*arg1*` et `*arg2*`, un compteur ordinal `*pc*` et une pile (Figure 7.3).

Les instructions sont au nombre de 34 et figurent dans la table 7.2. On a ajouté aux instructions précédemment vues, l'instruction `FINISH` qui permet d'achever les calculs (plus exactement de sortir de la fonction `run`) et, par exemple, de rendre la main au système d'exploitation ou à sa simulation en Lisp. **[☞ p. 213]**

Les 25 instructions/générateurs du langage intermédiaire peuvent être rangées en deux groupes : le groupe des instructions terminales et celui des instructions composites. Les 9 instructions terminales se retrouvent telles quelles (et sous le même nom) ; elles se signalent par une petite étoile en suffixe au tableau 7.2 parmi les 34 instructions de notre machine. En revanche, les 16 composites se sont vues explicitées en termes de 25 nouvelles instructions élémentaires. On aurait pu davantage individualiser certains effets et, par exemple, décomposer `CHECKED-GLOBAL-REF` en une séquence effectuant `GLOBAL-REF` puis vérifiant que `*val*` contient bien une valeur correcte, mais cela aurait nui à la vitesse de l'interprète que d'effectuer deux tours de `run` au lieu d'un pour obtenir la même chose. Inversement, on aurait pu agglomérer des instructions comme on peut le constater sur `CALL3` : `(POP-ARG2)` est toujours suivie de `(POP-ARG1)`, cette séquence particulière aurait pu être combinée en une unique instruction.

Quelques-unes des 16 instructions composites du langage intermédiaire ne sont point apparues ci-dessus. Voici leur définition brute et sans explication. La définition des instructions manquantes de la machine figurant dans le tableau 7.2 sera reprise par la suite.

```
(define (DEEP-ARGUMENT-SET! i j m)
  (append m (SET-DEEP-ARGUMENT! i j)) )
(define (GLOBAL-SET! i m)
  (append m (SET-GLOBAL! i)) )
```

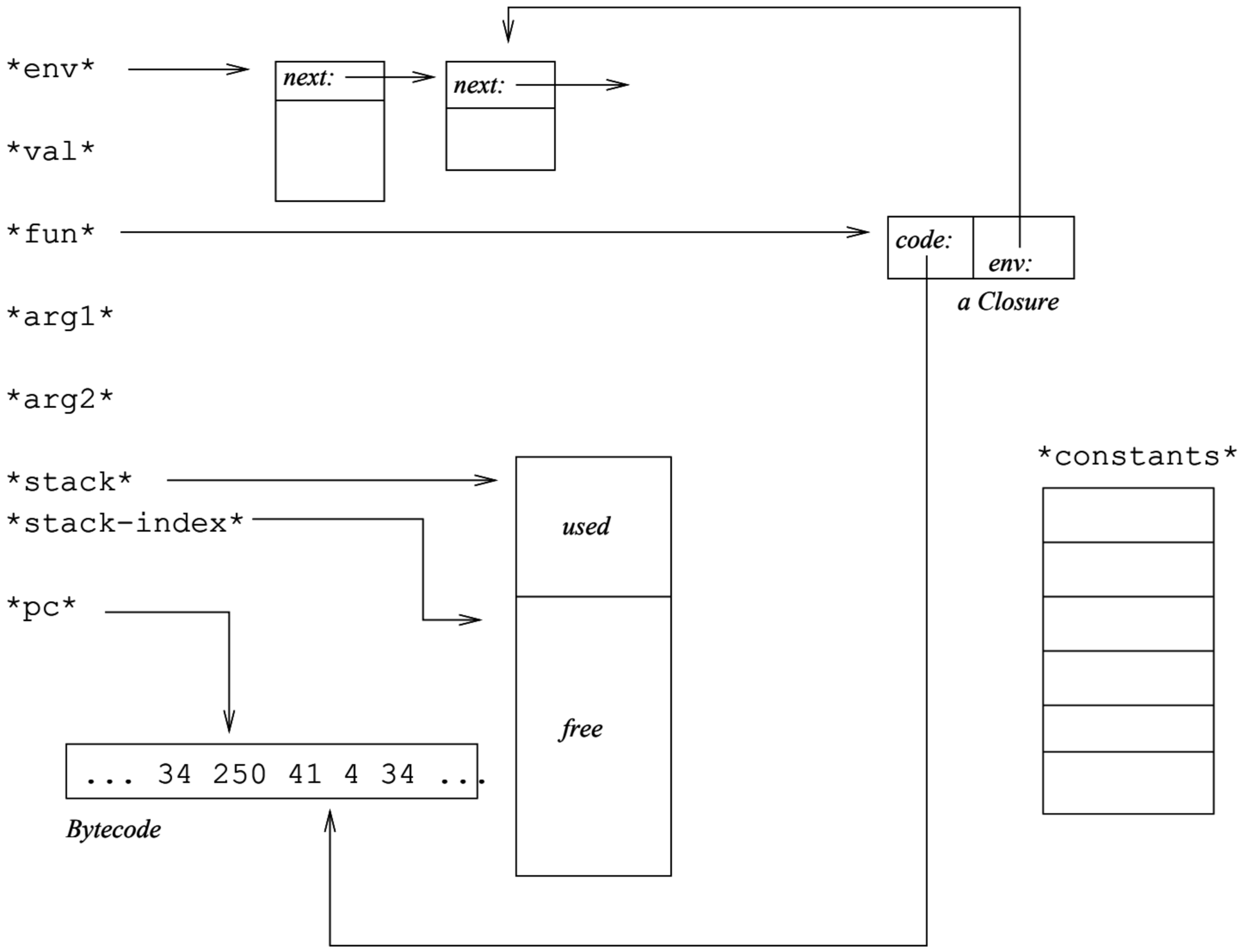


FIGURE 7.3 – Machine à octets

(SHALLOW-ARGUMENT-REF j)$_*$	(PREDEFINED i)$_*$
(DEEP-ARGUMENT-REF i j)$_*$	(SET-SHALLOW-ARGUMENT! j)
(SET-DEEP-ARGUMENT! i j)	(GLOBAL-REF i)$_*$
(CHECKED-GLOBAL-REF i)$_*$	(SET-GLOBAL! i)
(CONSTANT v)$_*$	(JUMP-FALSE $offset$)
(GOTO $offset$)	(EXTEND-ENV)
(UNLINK-ENV)	(CALL0 $address$)$_*$
(INVOKE1 $address$)	(PUSH-VALUE)
(POP-ARG1)	(INVOKE2 $address$)
(POP-ARG2)	(INVOKE3 $address$)
(CREATE-CLOSURE $offset$)	(ARITY=? $arity + 1$)
(RETURN)	(PACK-FRAME! $arity$)
(ARITY>=? $arity + 1$)	(POP-FUNCTION)
(FUNCTION-INVOKE)	(PRESERVE-ENV)
(RESTORE-ENV)	(POP-FRAME! $rank$)
(POP-CONS-FRAME! $arity$)	(ALLOCATE-FRAME $size$)$_*$
(ALLOCATE-DOTTED-FRAME $arity$)$_*$	(FINISH)

TABLE 7.2 – Instructions symboliques

```
(define (SEQUENCE m m+)
  (append m m+) )
(define (TR-FIX-LET m* m+)
  (append m* (EXTEND-ENV) m+) )
(define (FIX-LET m* m+)
  (append m* (EXTEND-ENV) m+ (UNLINK-ENV)) )
(define (CALL1 address m1)
  (append m1 (INVOKE1 address) ) )
(define (CALL2 address m1 m2)
  (append m1 (PUSH-VALUE) m2 (POP-ARG1) (INVOKE2 address)) )
(define (CALL3 address m1 m2 m3)
  (append m1 (PUSH-VALUE)
          m2 (PUSH-VALUE)
          m3 (POP-ARG2) (POP-ARG1) (INVOKE3 address) ) )
(define (FIX-CLOSURE m+ arity)
  (define the-function
    (append (ARITY=? (+ arity 1)) (EXTEND-ENV) m+ (RETURN)) )
  (append (CREATE-CLOSURE 1) (GOTO (length the-function))
          the-function ) )
(define (TR-REGULAR-CALL m m*)
  (append m (PUSH-VALUE) m* (POP-FUNCTION) (FUNCTION-INVOKE)) )
(define (STORE-ARGUMENT m m* rank)
  (append m (PUSH-VALUE) m* (POP-FRAME! rank)) )
(define (CONS-ARGUMENT m m* arity)
  (append m (PUSH-VALUE) m* (POP-CONS-FRAME! arity)) )
```

Les innombrables `append` qui parsèment ces définitions rendent le prétraitement particulièrement dispendieux et peu efficace. La bonne solution est de construire le code final en une seule passe, ce qui sera le cas lors de la compilation vers C. **[☞ p. 354]** Il faut pour cela linéariser la production de code, ce qui est indépendant de la linéarisation du code à laquelle ce chapitre se consacre. Pour prendre un exemple, `CALL2` devrait *(i)* engendrer le code de `m1` ; *(ii)* produire le code de `(PUSH-VALUE)` ; *(iii)* engendrer le code de `m2` ; *(iv)* produire le code de `(POP-ARG1)` ; enfin *(v)* produire le code de `(INVOKE2 address)`. S'acquitter de ces modifications est partout trivial sauf en ce qui concerne les `GOTO` ou `JUMP-FALSE` que l'on ne peut plus précalculer dans `ALTERNATIVE`, `FIX-CLOSURE` et `NARY-CLOSURE`. Il faut pour cela mettre en place un mécanisme de rustinage (pour *backpatching*). Pour `FIX-CLOSURE` par exemple, on *(i)* produira l'instruction `CREATE-CLOSURE` ; *(ii)* produira une instruction `GOTO` sans spécifier le décalage mais en notant la valeur du compteur ordinal courant ; *(iii)* on engendrera le code du corps de la fonction ; *(iv)* on notera le compteur ordinal courant pour en déduire le décalage du `GOTO` auparavant engendré ; enfin *(v)* on écrira ce décalage à la place qui lui avait été réservée dans l'instruction `GOTO`[2]. Le résultat net sera un processus de production de code beaucoup plus efficace que l'actuel que nous conserverons néanmoins pour sa simplicité.

2. Le problème se complique sur une machine à octets suivant que l'on a réservé 1 ou 2 octets pour le décalage qui peut s'avèrer, après détermination, être inférieur ou supérieur à 256. Quelques risques de suboptimalité sont donc à craindre.

7.2.1 Désassemblage

Nous avions montré en tête de chapitre la forme intermédiaire que revêtait le petit programme suivant : **[☞ p. 213]**

```
((lambda (fact) (fact 5 fact (lambda (x) x)))
 (lambda (n f k) (if (= n 0) (k 1)
                     (f (- n 1) f (lambda (r) (k (* n r)))) )) )
```

Il nous est maintenant possible d'exhiber la forme symbolique qu'il arbore dans le langage de notre machine. Les 78 instructions de la figure 7.4 font vraiment penser à du langage machine !

```
(CREATE-CLOSURE 2)
(GOTO 59)
(ARITY=? 4)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF 0)
(PUSH-VALUE)
(CONSTANT 0)
(POP-ARG1)
(INVOKE2 =)
(JUMP-FALSE 10)
(SHALLOW-ARGUMENT-REF 2)
(PUSH-VALUE)
(CONSTANT 1)
(PUSH-VALUE)
(ALLOCATE-FRAME 2)
(POP-FRAME! 0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(GOTO 39)
(SHALLOW-ARGUMENT-REF 1)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF 0)
(PUSH-VALUE)
(CONSTANT 1)
(POP-ARG1)
(INVOKE2 -)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF 1)
(PUSH-VALUE)
(CREATE-CLOSURE 2)
(GOTO 19)
(ARITY=? 2)
(EXTEND-ENV)
(DEEP-ARGUMENT-REF 1 2)
(PUSH-VALUE)
(DEEP-ARGUMENT-REF 1 0)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF 0)
(POP-ARG1)
(INVOKE2 *)
(PUSH-VALUE)
(ALLOCATE-FRAME 2)
(POP-FRAME! 0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME 4)
(POP-FRAME! 2)
(POP-FRAME! 1)
(POP-FRAME! 0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME 2)
(POP-FRAME! 0)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF 0)
(PUSH-VALUE)
(CONSTANT 5)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF 0)
(PUSH-VALUE)
(CREATE-CLOSURE 2)
(GOTO 4)
(ARITY=? 2)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF 0)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME 4)
(POP-FRAME! 2)
(POP-FRAME! 1)
(POP-FRAME! 0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
```

FIGURE 7.4 – Factorielle désassemblée

7.3 Codage des instructions

Cela fait des pages que nous parlons d'octets et nous n'en avons pas encore vu la queue d'un ! Pour les faire apparaître, il suffit de considérer les instructions du tableau 7.2 non plus comme des instructions mais comme des générateurs d'octets exécutés par une nouvelle fonction `run` encore plus adaptée. Ce changement de point de vue nous permettra de faire émerger des améliorations importantes pour la vitesse et la compacité du code engendré.

Le codage des instructions en octets doit faire l'objet de soins attentifs. Il faut à la fois élire un octet, lui associer son comportement au sein de la fonction `run` ainsi que diverses informations comme la longueur du codage (par exemple, pour une fonction de désassemblage). Pour toutes ces raisons, les instructions seront définies à l'aide d'une syntaxe particulière : `define-instruction`. Toutes les définitions d'instructions (qui apparaîtront çà et là) seront supposées regroupées au sein de la macro `define-instruction-set` que voici, accompagnée de quelques utilitaires :

```
(define-syntax define-instruction-set
  (syntax-rules (define-instruction)
    ((define-instruction-set
       (define-instruction (name . args) n . body) ... )
     (begin
       (define (run)
         (let ((instruction (fetch-byte)))
           (case instruction
             ((n) (run-clause args body)) ... ) ) )
       (run) )
     (define (instruction-size code pc)
       (let ((instruction (vector-ref code pc)))
         (case instruction
           ((n) (size-clause args)) ... ) ) )
     (define (instruction-decode code pc)
       (define (fetch-byte)
         (let ((byte (vector-ref code pc)))
           (set! pc (+ pc 1))
           byte ) )
       (let-syntax
           ((decode-clause
             (syntax-rules ()
               ((decode-clause iname ()) '(iname))
               ((decode-clause iname (a))
                (let ((a (fetch-byte))) (list 'iname a)) )
               ((decode-clause iname (a b))
                (let* ((a (fetch-byte))(b (fetch-byte)))
                  (list 'iname a b) ) ) )))
         (let ((instruction (fetch-byte)))
           (case instruction
             ((n) (decode-clause name args)) ... ) ) ) ) ) ) ) )
(define-syntax run-clause
  (syntax-rules ()
    ((run-clause () body) (begin . body))
    ((run-clause (a) body)
     (let ((a (fetch-byte))) . body) )
    ((run-clause (a b) body)
     (let* ((a (fetch-byte))(b (fetch-byte))) . body) ) ) )
```

```
(define-syntax size-clause
  (syntax-rules ()
    ((size-clause ())    1)
    ((size-clause (a))   2)
    ((size-clause (a b)) 3) ) )
```

La syntaxe `define-instruction-set` permet d'engendrer trois fonctions à la fois : la fonction `run` d'interprétation des octets, la fonction `instruction-size` qui calcule la taille du codage d'une instruction et la fonction `instruction-decode` qui permet de désassembler les octets composant une instruction particulière et dont la présence est, particulièrement en période de mise au point, la bienvenue. Contemplez, s'il vous plait, l'instruction :

```
(define-instruction (SHALLOW-ARGUMENT-REF j) 5
  (set! *val* (activation-frame-argument *env* j)) )
```

Cette définition participe à la définition de `run` en lui ajoutant une clause déclenché par l'octet `5` :

```
(define (run)
  (let ((instruction (fetch-byte)))
    (case instruction
       ...
       ((5) (let ((j (fetch-byte)))
              (set! *val* (activation-frame-argument *env* j)) ))
       ... ) )
  (run) )
```

La fonction `fetch-byte` lit l'(ou les) argument(s) nécessaire(s). Elle se définit simplement et a pour effet secondaire d'incrémenter, en passant, le compteur ordinal :

```
(define (fetch-byte)
  (let ((byte (vector-ref *code* *pc*)))
    (set! *pc* (+ *pc* 1))
    byte ) )
```

Le compteur ordinal est lui-même manipulé par la fonction `run` qui utilise de même `fetch-byte` pour lire la prochaine instruction. Notez, au passage, ce qui est d'ailleurs le cas d'un certain nombre de processeurs, que durant l'exécution d'une instruction le compteur ordinal désigne la prochaine instruction, à exécuter et non l'instruction courante.

Si l'on veut obtenir une grande célérité dans l'interprétation des octets, c'est-à-dire une très efficace fonction `run`, il faut veiller à la vitesse d'exécution de la forme `case` qui y apparaît. Sachant que l'instruction ne peut qu'être un octet, c'est-à-dire un nombre entre 0 et 255, la meilleure compilation de `case` est d'utiliser une table de sauts indexés par l'octet car alors le choix d'une clause s'effectue en temps constant. La recherche linéaire qui serait induite par l'expansion de `case` en une suite de `(if (eq? ...) ...)` serait létale. Peu de compilateurs pour Lisp (parmi lesquels Sqil [Sén91] et Bigloo [Ser93]) sont capables de cette performance.

La syntaxe `define-instruction` participe aussi à la fonction `instruction-size` et lui ajoute la clause comme quoi une instruction `SHALLOW-ARGUMENT-REF` a pour longueur 2 octets. L'instruction est désignée par son adresse (`pc`) dans le vecteur de code où elle figure. Il est certain qu'une forme `case`, présente ici pour stocker un vecteur de tailles d'instruction, est superfétatoire puisqu'aucune instruction n'a une taille variable.

```
(define (instruction-size code pc)
  (let ((instruction (vector-ref code pc)))
    (case instruction
      ...
      ((5) 2)
      ... ) ) )
```

Enfin, la définition d'une instruction participe à la fonction `instruction-decode` en lui ajoutant une clause spécialisée dans la reconnaissance de `SHALLOW-ARGUMENT-REF`. La fonction `instruction-decode` utilise sa propre définition de `fetch-byte` afin de ne pas perturber le compteur ordinal tout en accentuant sa ressemblance avec `run`.

```
(define (instruction-decode code pc)
  (define (fetch-byte)
    (let ((byte (vector-ref code pc)))
      (set! pc (+ pc 1))
      byte ) )
  (let ((instruction (fetch-byte)))
    (case instruction
      ...
      ((5) (let ((j (fetch-byte))) (list 'SHALLOW-ARGUMENT-REF j)))
      ... ) ) )
```

7.4 Instructions

Il y a 256 instructions possibles, ce qui nous laisse beaucoup d'espace puisque seules 34 instructions sont nécessaires. Nous allons profiter de cette manne pour réserver quelques octets de plus pour encoder les combinaisons les plus utiles. Par exemple, il est clair que la plupart des fonctions ont peu de variables. L'analyse de l'arité des fonctions [Cha80] figurant (au jour où j'écris ces lignes [3]) dans les programmes associés à ce livre, donne les résultats suivants. Seize fonctions sont d'arité variable ; les 1988 autres se répartissent comme indiqué au tableau 7.3.

arité	0	1	2	3	4	5	6	7	8
fréquence (en %)	35	30	18	9	4	1	0	0	0
cumul (en %)	35	66	84	93	97	99	99	99	100

TABLE 7.3 – Répartition des arités

L'examen de ce tableau montre que la plupart des fonctions ont moins de quatre variables. Généralisons ce résultat en admettant son universalité bien que l'arité zéro soit ici sur-représentée par rapport à la réalité du fait du chapitre 6. Nous chercherons donc à améliorer la vitesse d'exécution de fonctions ayant moins de quatre variables. En conséquence, toutes les instructions liées à l'arité se verront spécialisées pour les arités inférieures à quatre.

3. Les statistiques ont été produites par un programme écrit en Perl [WS90].

7.4.1 Variables locales

Parmi celles-là, prenons tout d'abord le cas de `SHALLOW-ARGUMENT-REF`. Cette instruction nécessite un argument j et charge le registre `*val*` avec l'argument j du premier bloc d'activation contenu dans le registre d'environnement `*env*`. Nous pouvons la spécialiser en dédiant cinq octets particuliers pour représenter les cas $j = 0, 1, 2, 3$. Nous pouvons aussi limiter notre machine en lui interdisant de tolérer les fonctions de plus de 256 variables [4], ce qui permet de coder l'index de l'argument à chercher sur un unique octet (c'est ce que vérifie la fonction `check-byte` [5]). Voici donc la fonction `SHALLOW-ARGUMENT-REF` vue comme un générateur de code-octets : elle retourne la liste des octets engendrés [6]. Cette liste contient un ou deux octets selon les cas.

```
(define (SHALLOW-ARGUMENT-REF j)
  (check-byte j)
  (case j
    ((0 1 2 3) (list (+ 1 j)))
    (else      (list 5 j)) ) )
(define (check-byte j)
  (unless (and (<= 0 j) (<= j 255))
    (static-wrong "Cannot pack this number within a byte" j) ) )
```

Et voici les cinq instructions physiques associées [7] :

```
(define-instruction (SHALLOW-ARGUMENT-REF0) 1
  (set! *val* (activation-frame-argument *env* 0)) )
(define-instruction (SHALLOW-ARGUMENT-REF1) 2
  (set! *val* (activation-frame-argument *env* 1)) )
(define-instruction (SHALLOW-ARGUMENT-REF2) 3
  (set! *val* (activation-frame-argument *env* 2)) )
(define-instruction (SHALLOW-ARGUMENT-REF3) 4
  (set! *val* (activation-frame-argument *env* 3)) )
(define-instruction (SHALLOW-ARGUMENT-REF j) 5
  (set! *val* (activation-frame-argument *env* j)) )
```

La modification de variables locales, qu'opère `SET-SHALLOW-ARGUMENT!`, subit le même traitement et conduit à ce qui suit (les définitions manquantes se déduisent aisément) :

```
(define (SET-SHALLOW-ARGUMENT! j)
  (case j
    ((0 1 2 3) (list (+ 21 j)))
    (else      (list 25 j)) ) )
(define-instruction (SET-SHALLOW-ARGUMENT!2) 23
  (set-activation-frame-argument! *env* 2 *val*) )
(define-instruction (SET-SHALLOW-ARGUMENT! j) 25
  (set-activation-frame-argument! *env* j *val*) )
```

---

4. COMMON LISP procure une constante `lambda-parameters-limit` dont la valeur est le nombre maximal de variables que peut avoir une fonction, sachant que ce nombre ne peut être inférieur à 50. Scheme ne dit rien sur ce point, ce qui peut s'interpréter de multiples manières. C'est d'ailleurs un problème intéressant que de représenter un entier vraiment quelconque. La plupart des systèmes de *bignums* les limitent à des entiers inférieurs à $(256)^{2^{32}}$, ce qui est encore loin de l'infini. Une solution existe qui consiste à préfixer la représentation d'un tel nombre par la représentation de la longueur de sa représentation. Ce schéma éminemment récursif s'arrête bien sûr à la représentation d'un petit entier.
5. Par la suite et pour alléger les fonctions présentées, nous ne mentionnerons plus `check-byte`.
6. Encore une fois, nous dilapidons notre capital par une profusion de `list` et `append`.
7. Il n'y a pas d'instruction de code `0` car il y a trop de zéros libres dans la nature.

En ce qui concerne les variables profondes, nous supposerons que tous les cas sont équiprobables [8] et les encoderons comme :

```
(define (DEEP-ARGUMENT-REF i j) (list 6 i j))
(define (SET-DEEP-ARGUMENT! i j) (list 26 i j))
(define-instruction (DEEP-ARGUMENT-REF i j) 6
  (set! *val* (deep-fetch *env* i j)) )
(define-instruction (SET-DEEP-ARGUMENT! i j) 26
  (deep-update! *env* i j *val*) )
```

7.4.2 Variables globales

Nous supposerons que toutes les variables globales modifiables sont équiprobables et qu'elles peuvent donc être directement encodées. Pour simplifier, nous supposerons aussi qu'il ne peut y avoir plus de 256 telles variables afin de pouvoir les encoder en un unique octet. Nous aurons donc :

```
(define (GLOBAL-REF i) (list 7 i))
(define (CHECKED-GLOBAL-REF i) (list 8 i))
(define (SET-GLOBAL! i) (list 27 i))
(define-instruction (GLOBAL-REF i) 7
  (set! *val* (global-fetch i)) )
(define-instruction (CHECKED-GLOBAL-REF i) 8
  (set! *val* (global-fetch i))
  (when (eq? *val* undefined-value)
    (signal-exception
     #t (list "Uninitialized global variable" i) ) ) )
(define-instruction (SET-GLOBAL! i) 27
  (global-update! i *val*) )
```

Le cas des variables globales prédéfinies et non modifiables est plus intéressant car on peut dédier quelques octets aux cas statistiquement les plus significatifs. Ici, nous choisirons d'accélérer la lecture des variables `T`, `F`, `NIL`, `CONS`, `CAR` et de quelques autres.

```
(define (PREDEFINED i)
  (check-byte i)
  (case i
    ;; 0=#t, 1=#f, 2=(), 3=cons, 4=car, 5=cdr, 6=pair?, 7=symbol?, 8=eq?
    ((0 1 2 3 4 5 6 7 8) (list (+ 10 i)))
    (else                (list 19 i)) ) )
(define-instruction (PREDEFINED0) 10   ;; #T
  (set! *val* #t) )
(define-instruction (PREDEFINED i) 19
  (set! *val* (predefined-fetch i)) )
```

Puisque nous avons commencé de traiter quelques constantes, traitons de même les citations. On admettra que la machine a un registre `*constants*` contenant un vecteur contenant lui-même toutes les citations faites par le programme. Une citation peut-être recherchée par la fonction `quotation-fetch`.

```
(define (quotation-fetch i)
```

8. C'est en fait faux, puisque les explications précédentes ont au moins montré que le paramètre j était en général inférieur à 4. Néanmoins, les variables profondes sont plus rares et cela peut justifier de ne pas chercher à améliorer leur accès.

```
(vector-ref *constants* i) )
```

Les citations sont collectées au sein de la variable `*quotations*` pendant la phase de compilation par le combinateur `CONSTANT`. Les citations faites seront sauvegardées lors de la compilation et déposées dans le registre `*constants*` à l'exécution. Une fois de plus, toutes les valeurs citées ne sont pas égales et certaines sont plus citées que d'autres, on dédiera quelques octets à celles-là. On réutilisera au passage certains des octets précédemment définis : `PREDEFINED0` et suivants. Enfin on supposera que l'on ne peut citer comme entiers immédiats que ceux compris entre 0 et 255 ; les autres entiers seront cités comme des constantes normales [9].

```
(define (CONSTANT value)
  (cond ((eq? value #t)     (list 10))
        ((eq? value #f)     (list 11))
        ((eq? value '())    (list 12))
        ((equal? value -1) (list 80))
        ((equal? value 0)   (list 81))
        ((equal? value 1)   (list 82))
        ((equal? value 2)   (list 83))
        ((equal? value 4)   (list 84))
        ((and (integer? value)   ; immediate value
              (<= 0 value)
              (< value 255) )
         (list 79 value) )
        (else (EXPLICIT-CONSTANT value)) ) )
(define (EXPLICIT-CONSTANT value)
  (set! *quotations* (append *quotations* (list value)))
  (list 9 (- (length *quotations*) 1)) )
(define-instruction (CONSTANT-1)  80
  (set! *val* -1) )
(define-instruction (CONSTANT0) 81
  (set! *val* 0) )
(define-instruction (SHORT-NUMBER value) 79
  (set! *val* value) )
```

7.4.3 Sauts

Si les restrictions que nous avons ci-dessus imposées étaient contraignantes, une sujétion plus grande encore serait celle limitant les sauts `GOTO` et `JUMP-FALSE` à 256 octets [10] puisque la taille du saut dépend de la taille du code compilé. Nous distinguerons deux cas suivant que le saut tient sur un octet ou deux. Nous aurons donc des instructions physiques différentes : `SHORT-GOTO` et `LONG-GOTO`. Ce sera un handicap de notre compilateur que de ne pas savoir bondir à plus de 65535 octets [11] :

```
(define (GOTO offset)
  (cond ((< offset 255) (list 30 offset))
        ((< offset (+ 255 (* 255 256)))
         (let ((offset1 (modulo offset 256))
               (offset2 (quotient offset 256)) )
```

---

9. C'est ennuyeux mais cela permet quand même d'implanter des grands nombres (*bignums*) grâce aux listes.

10. D'anciennes machines, comme celles à base de 8086, avaient cette sorte de limitation.

11. La même technique de dédoublement d'une instruction en `SHORT-` et `LONG-` peut s'appliquer à `GLOBAL-REF` et consorts pour ne plus limiter le nombre des variables globales modifiables à 256.

```
          (list 28 offset1 offset2) ) )
        (else (static-wrong "too long jump" offset)) ) )
(define (JUMP-FALSE offset)
  (cond ((< offset 255) (list 31 offset))
        ((< offset (+ 255 (* 255 256)))
         (let ((offset1 (modulo offset 256))
               (offset2 (quotient offset 256)) )
           (list 29 offset1 offset2) ) )
        (else (static-wrong "too long jump" offset)) ) )
(define-instruction (SHORT-GOTO offset) 30
  (set! *pc* (+ *pc* offset)) )
(define-instruction (SHORT-JUMP-FALSE offset) 31
  (if (not *val*) (set! *pc* (+ *pc* offset))) )
(define-instruction (LONG-GOTO offset1 offset2) 28
  (let ((offset (+ offset1 (* 256 offset2))) )
    (set! *pc* (+ *pc* offset)) ) )
```

7.4.4 Invocations

Nous avons à successivement analyser les invocations générales et les appels intégrés. La raison pour laquelle nous avions favorisé les fonctions de faible arité est toujours valable. Aussi pouvons-nous spécialiser, pour quelques octets de plus, les allocations de blocs d'activation pour des arités allant jusqu'à quatre variables.

```
(define (ALLOCATE-FRAME size)
  (case size
    ((0 1 2 3 4) (list (+ 50 size)))
    (else        (list 55 (+ size 1))) ) )
(define-instruction (ALLOCATE-FRAME1) 50
  (set! *val* (allocate-activation-frame 1)) )
(define-instruction (ALLOCATE-FRAME size+1) 55
  (set! *val* (allocate-activation-frame size+1)) )
```

Le rangement des valeurs des arguments dans les blocs d'activation peut aussi être amélioré pour les petites arités.

```
(define (POP-FRAME! rank)
  (case rank
    ((0 1 2 3) (list (+ 60 rank)))
    (else      (list 64 rank)) ) )
(define-instruction (POP-FRAME!0) 60
  (set-activation-frame-argument! *val* 0 (stack-pop)) )
(define-instruction (POP-FRAME! rank) 64
  (set-activation-frame-argument! *val* rank (stack-pop)) )
```

Les invocations intégrées sont très fréquentes, aussi méritent-elles qu'on leur dédie quelques octets. Par exemple, `INVOKE1`, qui est l'invoqueur particulier des fonctions prédéfinies unaires, s'écrit :

```
(define (INVOKE1 address)
  (case address
    ((car)     (list 90))
    ((cdr)     (list 91))
    ((pair?)   (list 92))
    ((symbol?) (list 93))
    ((display) (list 94))
    (else (static-wrong "Cannot integrate" address)) ) )
```

```
(define-instruction (CALL1-car) 90
  (set! *val* (car *val*)) )
(define-instruction (CALL1-cdr) 91
  (set! *val* (cdr *val*)) )
```

On fait bien sûr de même pour les fonctions prédéfinies d'arité 0, 2 et 3. L'apparition de `display` dans `INVOKE1` est liée à des raisons de mise au point ; c'est une fonction qui mérite plutôt d'apparaître dans une bibliothèque non primitive en raison de sa taille et du temps qu'elle requiert pour s'appliquer. Il vaudrait mieux dédier quelques octets à `cddr`, puis à `cdddr` et enfin à `cadr` dans l'ordre d'utilité.

La vérification d'arité peut, elle aussi, être spécialisée pour les petites arités. On distinguera ainsi :

```
(define (ARITY=? arity+1)
  (case arity+1
    ((1 2 3 4) (list (+ 70 arity+1)))
    (else        (list 75 arity+1)) ) )
(define-instruction (ARITY=?2) 72
  (unless (= (activation-frame-argument-length *val*) 2)
    (signal-exception
     #f (list "Incorrect arity for unary function") ) ) )
(define-instruction (ARITY=? arity+1) 75
  (unless (= (activation-frame-argument-length *val*) arity+1)
    (signal-exception #f (list "Incorrect arity")) ) )
```

Compte tenu de la faible proportion de fonctions d'arité multiple, on réservera le traitement précédent aux fonctions d'arité fixe. Les fonctions ici intégrées sont très simples et de calcul expéditif. On peut songer, puisque de nombreux octets sont encore libres, à intégrer des fonctions plus complexes comme, par exemple, `memq` ou `equal`. Le danger est qu'une invocation à `memq` peut ne jamais terminer (ou prendre un temps considérable) si la liste à laquelle elle s'applique est cyclique. MacScheme [85M85] limite à quelques milliers la longueur des listes en lesquelles chercher avec `memq`.

7.4.5 Divers

Il reste un assez grand nombre de petites instructions que l'on codera tout simplement sur un octet car elles n'ont pas d'argument. Les générateurs d'octets sont tous taillés sur le même gabarit, dont voici un exemple :

```
(define (RESTORE-ENV) (list 38))
```

Et voici leur définition en termes d'instructions :

```
(define-instruction (EXTEND-ENV) 32
  (set! *env* (sr-extend* *env* *val*)) )
(define-instruction (UNLINK-ENV) 33
  (set! *env* (activation-frame-next *env*)) )
(define-instruction (PUSH-VALUE) 34
  (stack-push *val*) )
(define-instruction (POP-ARG1) 35
  (set! *arg1* (stack-pop)) )
(define-instruction (POP-ARG2) 36
  (set! *arg2* (stack-pop)) )
(define-instruction (CREATE-CLOSURE offset) 40
  (set! *val* (make-closure (+ *pc* offset) *env*)) )
(define-instruction (RETURN) 43
```

```
  (set! *pc* (stack-pop)) )
(define-instruction (FUNCTION-GOTO) 46
  (invoke *fun* #t) )
(define-instruction (POP-FUNCTION) 39
  (set! *fun* (stack-pop)) )
(define-instruction (FUNCTION-INVOKE) 45
  (invoke *fun* #f) )
(define-instruction (PRESERVE-ENV) 37
  (preserve-environment) )
(define-instruction (RESTORE-ENV) 38
  (restore-environment) )
```

La préservation de l'environnement ainsi que sa restitution sont définies par les fonctions suivantes :

```
(define (preserve-environment)
  (stack-push *env*) )
(define (restore-environment)
  (set! *env* (stack-pop)) )
```

Deux méthodes existent pour invoquer les fonctions : `FUNCTION-GOTO` en position terminale et pour les autres : `FUNCTION-INVOKE`. En position terminale, la pile contient, en son sommet, l'adresse de retour. En position non terminale et après invocation de la fonction, la machine doit revenir à l'instruction qui suit `(FUNCTION-INVOKE)` pour poursuivre le travail. C'est pourquoi `FUNCTION-INVOKE` doit sauvegarder le compteur ordinal de retour. Lorsque l'appel est en position terminale, l'instruction `FUNCTION-GOTO` aurait pu être compilée en `FUNCTION-INVOKE` suivie de `RETURN` mais il ne sert à rien d'empiler l'adresse de `RETURN` dans la pile, il suffit de ne point l'y mettre ! Ainsi `FUNCTION-GOTO` invoque-t-il une fonction sans sauvegarder l'appelant puisqu'il a terminé son travail et que rien n'est à poursuivre. La définition de `invoke`, ici générique, achèvera de clarifier notre explication :

```
(define-generic (invoke (f) tail?)
  (signal-exception #f (list "Not a function" f)) )
(define-method (invoke (f closure) tail?)
  (unless tail? (stack-push *pc*))
  (set! *env* (closure-closed-environment f))
  (set! *pc* (closure-code f)) )
```

La fonction `invoke` est générique afin de nous permettre de l'étendre à de nouveaux types d'objets et, par exemple, aux primitives traditionnelles que nous représenterons sous forme de suspensions.

```
(define-method (invoke (f primitive) tail?)
  (unless tail? (stack-push *pc*))
  ((primitive-address f)) )
```

Là encore, suivant la valeur de la variable `tail?`, on préserve ou pas l'adresse de retour.

7.4.6 Lancement du compilateur-interprète

Réaliser une boucle d'interprétation, comme nous l'avons fait au cours des chapitres précédents, demande une coopération précise entre un compilateur et une machine d'exécution. La fonction `stand-alone-producer7d` prend un programme, initialise le compilateur et notamment son environnement de variables globales modifiables

prédéfinies et la liste des citations. Le résultat de la fonction `meaning` est maintenant une liste d'octets que l'on range dans un vecteur précédés par le code de `FINISH` et suivis de celui correspondant à `RETURN`. Le compteur ordinal initial est calculé pour désigner le premier octet qui suit le prologue (ici le deuxième octet, c'est-à-dire celui d'adresse 1). Pour des raisons que l'on reverra plus tard, **[☞ Ex. 6.1]** mais déjà esquissées, on prépare la liste des variables globales modifiables ainsi que la liste des citations. Enfin le produit final de la compilation est représenté par une fonction attendant pour s'évaluer qu'on lui fournisse une taille de pile.

```
(define (chapter7d-interpreter)
  (define (toplevel)
    (display ((stand-alone-producer7d (read)) 100))
    (toplevel) )
  (toplevel) )
(define (stand-alone-producer7d e)
  (set! g.current (original.g.current))
  (set! *quotations* '())
  (let* ((code (make-code-segment (meaning e r.init #t)))
         (start-pc (length (code-prologue)))
         (global-names (map car (reverse g.current)))
         (constants (apply vector *quotations*)) )
    (lambda (stack-size)
      (run-machine stack-size start-pc code
                   constants global-names ) ) ) )
(define (make-code-segment m)
  (apply vector (append (code-prologue) m (RETURN))) )
(define (code-prologue)
  (set! finish-pc 0)
  (FINISH) )
```

La fonction `run-machine` initialise puis lance la machine. Elle alloue un vecteur pour stocker les valeurs des variables globales modifiables, range les noms de ces mêmes variables globales modifiables, alloue une pile de travail et initialise tous les registres. Le seul problème est maintenant d'arrêter la machine ! Le programme a été compilé comme s'il était en position terminale, c'est-à-dire qu'il finira par exécuter un `RETURN`. Pour revenir, il faut donc que la pile contienne initialement une adresse sur laquelle ce `RETURN` sautera. C'est pourquoi, la pile contient initialement l'adresse de l'instruction `FINISH` qui est ainsi définie :

```
(define-instruction (FINISH) 20
  (*exit* *val*) )
```

Lorsque cette instruction est exécutée, elle stoppe la machine et retourne le contenu de son registre `*val*` grâce à un judicieux échappement placé par `run-machine` que voici :

```
(define (run-machine stack-size pc code constants global-names)
  (set! sg.current (make-vector (length global-names)
                                undefined-value ))
  (set! sg.current.names global-names)
  (set! *constants*   constants)
  (set! *code*        code)
  (set! *env*         sr.init)
  (set! *stack*       (make-vector stack-size))
  (set! *stack-index* 0)
  (set! *val*         'anything)
```

```
(set! *fun*          'anything)
(set! *arg1*         'anything)
(set! *arg2*         'anything)
(stack-push finish-pc)                    ; pc for FINISH
(set! *pc*           pc)
(call/cc (lambda (exit)
           (set! *exit* exit)
           (run) )) )
```

7.4.7 Point d'orgue

Voici pour achever cette longue section sur les instructions, la forme finale que revêtira le codage par octets (désassemblés pour demeurer lisibles) **[☞ p. 236]** du programme cité en tête de chapitre : **[☞ p. 224]**

7.5 Continuations

Nous avions montré que `call/cc` était un opérateur magique qui réifie le contexte d'évaluation en un objet invocable, nous allons maintenant démythifier son implantation. L'implantation qui suit est canonique. Des versions plus efficaces mais plus compliquées existent [CHO88, HDB90, MB93].

Le contexte d'évaluation est constitué de la pile et rien que de la pile. En effet, les registres `*fun*`, `*arg1*` et `*arg2*` n'ont que des rôles temporaires et, en aucun cas, ne peuvent être capturés (il ne peut y avoir d'appel à `call/cc` ou à quoique ce soit d'autre pendant qu'ils sont actifs [12]). Le registre `*val*` servira à transmettre les valeurs qui seront soumises aux continuations et n'est donc pas à sauvegarder. Le registre `*env*` n'est pas non plus à sauvegarder car `call/cc` n'étant pas une fonction que l'on intègre, l'environnement est déjà sauvegardé du fait de l'appel à `call/cc`. Il ne reste donc que la pile à préserver et pour ce faire, l'on usera des fonctions :

```
(define (save-stack)
  (let ((copy (make-vector *stack-index*)))
    (vector-copy! *stack* copy 0 *stack-index*)
    copy ) )
(define (restore-stack copy)
  (set! *stack-index* (vector-length copy))
  (vector-copy! copy *stack* 0 *stack-index*) )
(define (vector-copy! old new start end)
  (let copy ((i start))
    (when (< i end)
          (vector-set! new i (vector-ref old i))
          (copy (+ i 1)) ) ) )
```

Les continuations auront leur propre classe et seront munies d'un protocole d'appel particulier. Lorsqu'une continuation est invoquée, elle restaure la pile, renseigne le registre `*val*` avec la valeur reçue puis se branche, comme par un `RETURN`, à l'adresse

12. Ce peut être faux si l'on doit répondre à des sollicitations asynchrones (comme un signal d'UN⋆X). Dans ce cas, il vaut mieux basculer un drapeau que l'on testera explicitement de loin en loin mais régulièrement [Dev85].

```
(CREATE-CLOSURE 2)
(SHORT-GOTO 59)
(ARITY=?4)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF0)
(PUSH-VALUE)
(CONSTANT0)
(POP-ARG1)
(CALL2-=)
(SHORT-JUMP-FALSE 10)
(SHALLOW-ARGUMENT-REF2)
(PUSH-VALUE)
(CONSTANT1)
(PUSH-VALUE)
(ALLOCATE-FRAME2)
(POP-FRAME!0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(SHORT-GOTO 39)
(SHALLOW-ARGUMENT-REF1)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF0)
(PUSH-VALUE)
(CONSTANT1)
(POP-ARG1)
(CALL2-)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF1)
(PUSH-VALUE)
(CREATE-CLOSURE 2)
(SHORT-GOTO 19)
(ARITY=?2)
(EXTEND-ENV)
(DEEP-ARGUMENT-REF 1 2)
(PUSH-VALUE)
(DEEP-ARGUMENT-REF 1 0)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF0)
(POP-ARG1)
(CALL2-*)
(PUSH-VALUE)
(ALLOCATE-FRAME2)
(POP-FRAME!0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME4)
(POP-FRAME!2)
(POP-FRAME!1)
(POP-FRAME!0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME2)
(POP-FRAME!0)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF0)
(PUSH-VALUE)
(SHORT-NUMBER 5)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF0)
(PUSH-VALUE)
(CREATE-CLOSURE 2)
(SHORT-GOTO 4)
(ARITY=?2)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF0)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME4)
(POP-FRAME!2)
(POP-FRAME!1)
(POP-FRAME!0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
```

TABLE 7.4 – Programme désassemblé

contenue en sommet de pile. Comme `call/cc` n'a pu être appelée que par `(FUNCTION-INVOKE)`, l'instruction qui suivra sera nécessairement un `(RESTORE-ENV)` comme cela peut être vérifié dans la définition de `REGULAR-CALL`. **[☞ p. 218]**

```
(define-class continuation Object
  ( stack ) )
(define-method (invoke (f continuation) tail?)
  (if (= (+ 1 1) (activation-frame-argument-length *val*))
      (begin
        (restore-stack (continuation-stack f))
        (set! *val* (activation-frame-argument *val* 0))
        (set! *pc* (stack-pop)) )
      (signal-exception #f (list "Incorrect arity" 'continuation)) ) )
```

Pour que l'invocation d'une continuation réussisse, il faut que la pile ait une structure bien particulière bâtie par `call/cc`. Cette construction est assez délicate car la capture de continuation se doit de ne pas consommer, elle-même, de pile c'est-à-dire que, dans la forme `(call/cc` *f*`)`, la fonction *f* doit être appelée en position terminale [13]. C'est ce que réalise la définition qui suit. Elle alloue un bloc d'activation, le remplit avec la continuation réifiée et appelle son argument `f` avec.

```
(definitial call/cc
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (= arity+1 (activation-frame-argument-length *val*))
           (let ((f (activation-frame-argument *val* 0))
                 (frame (allocate-activation-frame (+ 1 1))) )
             (set-activation-frame-argument!
              frame 0 (make-continuation (save-stack)) )
             (set! *val* frame)
             (set! *fun* f)                    ; useful for debug
             (invoke f #t) )
           (signal-exception #t (list "Incorrect arity"
                                      'call/cc )) ) ) ) ) )
```

En conclusion, l'implantation canonique de `call/cc` coûte une copie de pile à chaque réification et une autre copie à chaque invocation de continuation. De meilleures stratégies existent mais sont plus complexes à mettre en œuvre. L'idée majeure de la plupart de ces stratégies est que lorsqu'une continuation est réifiée, elle a beaucoup de chances de l'être encore, en tout ou en partie, [Dan87] ; il faut donc favoriser les schémas permettant de partager ces captures.

7.6 Échappements

Même si le coût de `call/cc` peut être amoindri, il ne serait pas loyal de ne pas détailler comment implanter de simples échappements. Nous choisirons d'implanter la forme spéciale nommée `bind-exit` **[☞ p. 95]** présente en Dylan et analogue au `let/cc` de EuLisp, au `block`/`return-from` de Common Lisp ou à l'`escape` de Vlisp. Contrairement à l'esprit de Scheme, nous avons opté pour une nouvelle forme spéciale plutôt que pour une fonction pour principalement deux raisons : la première est

13. Ce n'est cependant pas obligatoire en Scheme.

qu'introduire une nouvelle forme spéciale est un peu plus simple car on n'a pas à se plier à la structure de la pile telle qu'elle est dictée par le protocole d'appel fonctionnel, d'autre part cela nous permettra de montrer le compilateur entier sous forme de coupe dans ses multiples étages. Enfin, s'il fallait encore une dernière raison, c'est parce que fonction ou forme spéciale sont, ici, équivalentes en puissance puisque l'on peut programmer l'une par l'autre.

La forme `bind-exit` a la syntaxe suivante :

`(bind-exit (variable) formes...)` *Dylan*

La *variable* est liée à la continuation de la forme `bind-exit` puis le corps de la forme est évalué. La continuation capturée ne peut être utilisée que pendant la durée de cette dernière évaluation. Disposer de `bind-exit` permet de définir la fonction `call/ep` (pour *call-with-exit-procedure*) et inversement :

```
(define (call/ep f)
  (bind-exit (k) (f k)) )
(bind-exit (k) body) ≡ (call/ep (lambda (k) body))
```

Les échappements seront représentés par des objets de la classe `escape`. Ils possèdent un champ unique qui désigne la hauteur de la pile à l'instant où débute l'évaluation de la forme `bind-exit`.

```
(define-class escape Object
  ( stack-index ) )
```

Pour définir une nouvelle forme spéciale, on doit tout d'abord augmenter la fonction `meaning`, l'analyseur lexical des formes à compiler, d'une nouvelle clause reconnaissant ces formes. On adjoindra donc à `meaning` la clause :

```
... ((bind-exit) (meaning-bind-exit (caadr e) (cddr e) r tail?)) ...
```

Puis l'on définira la fonction de prétraitement de telles formes :

```
(define (meaning-bind-exit n e+ r tail?)
  (let* ((r2 (r-extend* r (list n)))
         (m+ (meaning-sequence e+ r2 #t)) )
    (ESCAPER m+) ) )
```

Le prétraitement consiste à faire du corps de la forme `bind-exit` une séquence à prétraiter dans un environnement lexical étendu par la variable locale qu'introduit `bind-exit`. La génération de code est l'apanage de la fonction `ESCAPER` (toujours en majuscules). Pour elle, nous inventerons deux nouvelles instructions : `PUSH-ESCAPER` et `POP-ESCAPER`.

```
(define (ESCAPER m+)
  (append (PUSH-ESCAPER (+ 1 (length m+)))
          m+ (RETURN) (POP-ESCAPER) ) )
(define (POP-ESCAPER) (list 250))
(define escape-tag (list '*ESCAPE*))
(define (PUSH-ESCAPER offset) (list 251 offset))
(define-instruction (POP-ESCAPER) 250
  (let* ((tag (stack-pop))
         (escape (stack-pop)) )
    (restore-environment) ) )
(define-instruction (PUSH-ESCAPER offset) 251
  (preserve-environment)
  (let* ((escape (make-escape (+ *stack-index* 3)))
         (frame (allocate-activation-frame 1)) )
    (set-activation-frame-argument! frame 0 escape)
```

```
(set! *env* (sr-extend* *env* frame))
(stack-push escape)
(stack-push escape-tag)
(stack-push (+ *pc* offset)) ) )
```

L'instruction `PUSH-ESCAPER` (Figure 7.5) *(i)* sauve l'environnement courant dans la pile ; *(ii)* alloue un échappement valide désignant la troisième case au-dessus de l'actuel sommet de pile ; *(iii)* alloue un bloc d'activation pour enrichir l'environnement lexical courant (dont l'unique champ contient l'échappement) ; *(iv)* empile enfin cet échappement puis un drapeau bizarre que s'octroie l'implantation, suivi d'une adresse désignant l'instruction `(POP-ESCAPER)` qui suit.

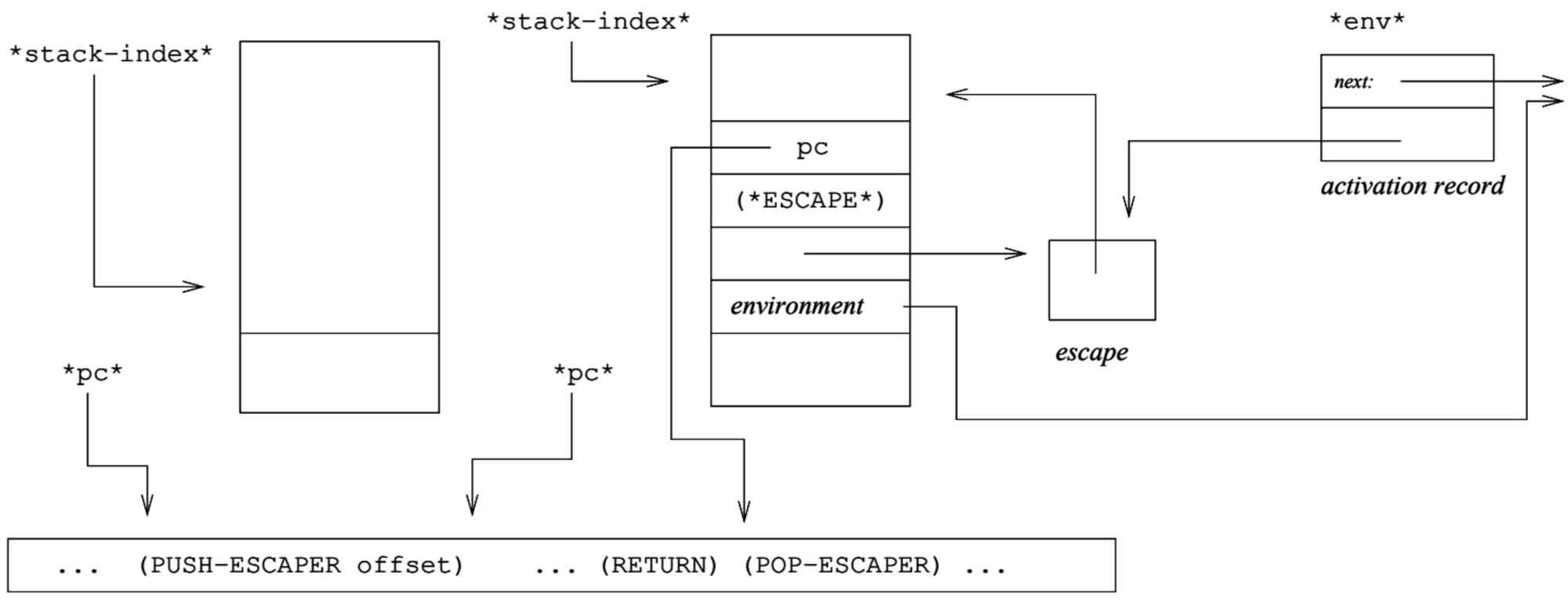


FIGURE 7.5 – Pile avant et après `PUSH-ESCAPER`

La méthode d'invocation des échappements est compliquée du fait qu'il faut détecter si l'échappement est valide ou pas. Un échappement est valide si la hauteur de pile est supérieure à la hauteur préservée dans l'échappement, s'il y a bien un échappement à cet endroit dans la pile et si c'est bien celui auquel on s'intéresse. Le tout est vérifié en temps constant par la fonction `escape-valid?`.

```
(define-method (invoke (f escape) tail?)
  (if (= (+ 1 1) (activation-frame-argument-length *val*))
      (if (escape-valid? f)
          (begin (set! *stack-index* (escape-stack-index f))
                 (set! *val* (activation-frame-argument *val* 0))
                 (set! *pc* (stack-pop)) )
          (signal-exception #f (list "Escape out of extent" f)) )
      (signal-exception #f (list "Incorrect arity" 'escape)) ) )
(define (escape-valid? f)
  (let ((index (escape-stack-index f)))
    (and (>= *stack-index* index)
         (eq? f (vector-ref *stack* (- index 3)))
         (eq? escape-tag (vector-ref *stack* (- index 2))) ) ) )
```

Lorsqu'un échappement est invoqué, il vérifie d'abord qu'il est bien valide puis il ramène le niveau de la pile à celui qui prévalait au début de l'évaluation du corps de la forme `bind-exit` ; il renseigne le registre `*val*` avec la valeur fournie puis effectue l'équivalent d'un `(RETURN)` affectant le compteur ordinal. L'instruction suivante sera donc `(POP-ESCAPER)` qui dépile l'échappement, restaure l'environnement sauvegardé et sort donc de la forme `bind-exit`.

Si aucun échappement n'est appelé durant le corps de la forme `bind-exit`, le même scénario s'applique : le `(RETURN)` qui précède `(POP-ESCAPER)` enlève l'adresse de la pile et permet de se brancher à cette même instruction dans la même configuration de pile que lors de l'appel à l'échappement.

L'entrée dans la forme `bind-exit` coûte deux allocations d'objets et quatre positions en pile. L'invocation d'un échappement coûte un test de validité et quelques mouvements de registres. L'échappement a été explicitement alloué car il se peut que la variable liée par la forme `bind-exit` soit capturée par une quelconque fermeture, ce qui lui confère implicitement une durée de vie illimitée. Dans le cas contraire, on peut ne pas allouer l'échappement et simplement conserver le pointeur de pile dans un registre. L'implantation que nous venons d'exposer est, on le conçoit donc, assez conservatrice et peu efficace au regard de ce que l'on peut obtenir par une bonne compilation. Elle demeure cependant et sans commune mesure largement plus efficace que l'implantation canonique de `call/cc`.

La forme spéciale `bind-exit` (ainsi que les variables dynamiques de la prochaine section) permettent de programmer facilement les analogues de `catch`/`throw`. **[☞ p. 72]** Si, par contre, le langage procurait une forme `unwind-protect`, alors la vitesse de `bind-exit` serait à revoir à la baisse car tout échappement devrait évaluer les nettoyeurs associés aux formes `unwind-protect` qui seraient prématurément achevées ; ces nettoyeurs seraient déterminés par une scrutation de la pile entre le point de départ et celui d'arrivée.

7.7 Variables dynamiques

On l'a souvent déjà écrit, les variables dynamiques correspondent à un concept important. Il en a été présenté une implantation en liaison profonde. On supposera, comme auparavant, **[☞ p. 159]** disposer de deux nouvelles formes spéciales pour créer et référencer des variables dynamiques. Voici leur syntaxe :

```
(dynamic-let (variable value) body...)
(dynamic variable)
```

La forme `dynamic-let` établit une liaison entre la variable et sa valeur pendant la durée d'évaluation de son corps. On peut obtenir la valeur de cette variable grâce à la forme `dynamic`. Il serait bien sûr erroné de demander la valeur d'une variable qui n'est pas liée.

On augmente donc l'analyseur syntaxique `meaning` avec deux nouvelles clauses :

```
... ((dynamic)      (meaning-dynamic-reference (cadr e) r tail?))
    ((dynamic-let) (meaning-dynamic-let (car (cadr e))
                                        (cadr (cadr e))
                                        (cddr e) r tail? )) ...
```

On y associe les deux prétraiteurs nécessaires :

```
(define (meaning-dynamic-let n e e+ r tail?)
  (let ((index (get-dynamic-variable-index n))
        (m (meaning e r #f))
        (m+ (meaning-sequence e+ r #f)) )
    (append m (DYNAMIC-PUSH index) m+ (DYNAMIC-POP)) ) )
(define (meaning-dynamic-reference n r tail?)
```

```
  (let ((index (get-dynamic-variable-index n)))
    (DYNAMIC-REF index) ) )
```

Ils emploient trois nouveaux générateurs :

```
(define (DYNAMIC-PUSH index) (list 242 index))
(define (DYNAMIC-POP)        (list 241))
(define (DYNAMIC-REF index)  (list 240 index))
```

Ils correspondent à trois nouvelles instructions de notre machine virtuelle :

```
(define-instruction (DYNAMIC-PUSH index) 242
  (push-dynamic-binding index *val*) )
(define-instruction (DYNAMIC-POP) 241
  (pop-dynamic-binding) )
(define-instruction (DYNAMIC-REF index) 240
  (set! *val* (find-dynamic-value index)) )
```

Comment représenter l'environnement des variables dynamiques ? La première idée qui vient à l'esprit est d'inventer un nouveau registre `*dynenv*` qui pointe en permanence sur une liste d'associations couplant variables dynamiques et valeurs. Cette liste n'est pas une vraie liste mais une suite chaînée de zones dans la pile. Malencontreusement on a augmenté l'état de la machine : le contenu du registre `*dynenv*` doit lui aussi être sauvegardé par `PRESERVE-ENV` et, bien sûr, être restauré par `RESTORE-ENV`. Comme ces deux instructions sont assez courantes, ajouter le registre `*dynenv*` serait assez coûteux même si l'on ne s'en servait jamais. Un grand principe est que seuls ses utilisateurs doivent payer un service ; corollairement, ceux qui ne se servent pas d'un service ne doivent en aucun cas être pénalisés. L'adjonction d'un registre est donc une mauvaise idée.

Plutôt que de maintenir un registre, donnons seulement aux utilisateurs de variables dynamiques le moyen de rétablir l'information s'il en est besoin. Le coût sera donc plus important mais au moins cette opération ne perturbera-t-elle pas ceux pour lesquels elle n'a pas d'utilité. L'environnement sera implanté sous la forme d'une suite de zones chaînées dans la pile mais chacune de ces zones sera précédée d'une étiquette spéciale l'identifiant à coup sûr (Figure 7.6). Le résultat de la fonction `search-dynenv-index` permet donc de rétablir l'information qui aurait pu se trouver dans un registre.

```
(define dynenv-tag (list '*dynenv*))
(define (search-dynenv-index)
  (let search ((i (- *stack-index* 1)))
    (if (< i 0) i
        (if (eq? (vector-ref *stack* i) dynenv-tag)
            (- i 1)
            (search (- i 1)) ) ) ) )
(define (pop-dynamic-binding)
  (stack-pop)
  (stack-pop)
  (stack-pop)
  (stack-pop) )
(define (push-dynamic-binding index value)
  (stack-push (search-dynenv-index))
  (stack-push value)
  (stack-push index)
  (stack-push dynenv-tag) )
```

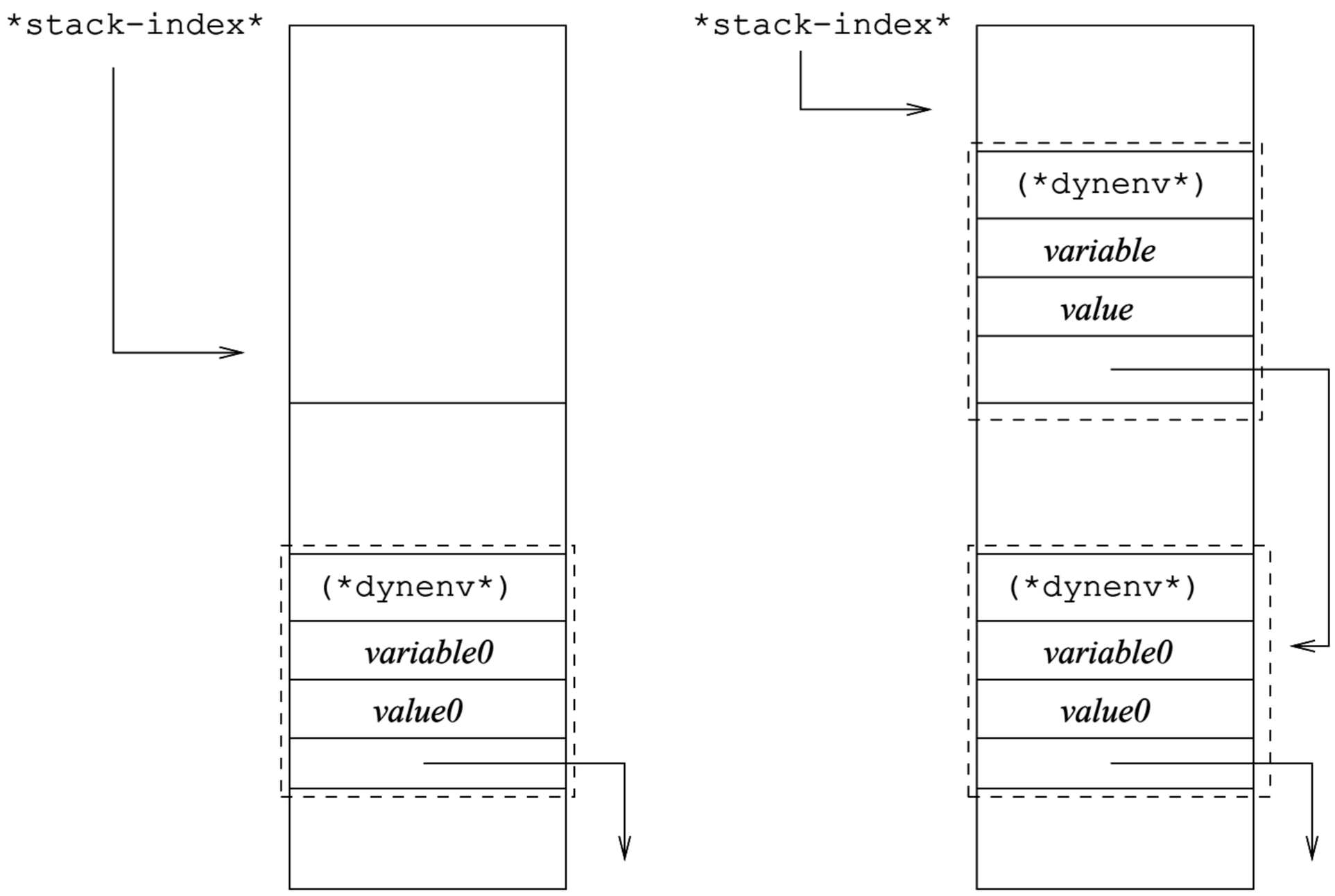


FIGURE 7.6 – Pile avant et après `DYNAMIC-PUSH`

Le point délicat qui reste à éclaircir est : Comment référencer les variables dynamiques ? Une référence telle que `(dynamic foo)` doit être compilée en des octets, ce qui exclut d'utiliser une référence directe au symbole `foo`. Exactement comme nous avons numéroté les variables globales modifiables au fur et à mesure qu'elles se présentaient, nous numéroterons les variables dynamiques grâce à `get-dynamic-variable-index` :

```
(define *dynamic-variables* '())
(define (get-dynamic-variable-index n)
  (let ((where (memq n *dynamic-variables*)))
    (if where (length where)
        (begin
          (set! *dynamic-variables* (cons n *dynamic-variables*))
          (length *dynamic-variables*) ) ) ) )
```

À toute variable dynamique est conféré un index qui permet de la retrouver dans l'environnement dynamique présent dans la pile. Notez que la variable `*dynamic-variables*` appartient au monde de la compilation et n'est pas nécessaire pour le bon fonctionnement de la machine. Elle peut cependant être utile pour établir un message comportant le nom de la variable en cas d'erreur.

Vous aurez noté que les variables dynamiques avaient été introduites par le biais de deux formes spéciales. Les procurer, plus fidèlement à l'esprit de Scheme, sous forme de fonctions n'aurait pas été équivalent. Puisque les fonctions sont invoquées avec des arguments calculés, les noms des variables dynamiques seraient alors calculés, ce qui n'est pas le cas de nos formes spéciales. La contrepartie serait évidente : il ne serait plus possible de numéroter les variables dynamiques puisque tout symbole,

voire toute valeur, pourrait servir. [☞ **p. 47**] Il nous faudrait alors envisager une autre compilation en faisant, par exemple, explicitement référence aux symboles qui sont des structures de taille respectable pouvant poser des problèmes de comparaison, surtout si on leur ajoute des concepts de paquetage ou d'internement.

7.8 Exceptions

Il n'y a pas de vrai langage qui ne définisse un mécanisme de rattrapage d'erreur. C'est un aspect très important qui permet de construire des applications robustes et autonomes. Il n'existe pas de mécanisme de rattrapage d'erreur dans la spécification de Scheme, aussi allons-nous en introduire un afin de montrer sans atours superflus ce qu'il en est.

Le concept d'erreur recouvre plusieurs réalités différentes. En premier lieu, on trouve rangées sous cette rubrique les situations non prévues, non souhaitées, mais testables, dans lesquelles une fonction est appliquée à de mauvais arguments. Pour des raisons de type, `(car 33)`, ou de domaine, `(quotient 111 0)`, le système sous-jacent peut achopper. Mais, si ces situations peuvent être explicitement testées à l'aide de prédicats appropriés, d'autres événements peuvent survenir comme de tenter d'ouvrir un fichier inexistant. Rattraper ces erreurs nécessite :

1. une mécanique de réification de l'erreur en une structure de données compréhensible par les programmes de l'utilisateur ;
2. un appel à la fonction que l'utilisateur a désignée comme étant celle qui doit s'occuper des erreurs.

Ce mécanisme une fois incorporé à un langage, les utilisateurs ont souhaité pouvoir eux-mêmes l'utiliser. Les erreurs ont alors pris le nom d'*exceptions* et d'elles naquit la *programmation par exception* où l'utilisateur ne programme que le cas correct et laisse le traitement des sorties (éventuellement multiples) aux bons soins du rattrapeur d'exceptions.

Plusieurs modèles d'exception existent ; ils fondent pratiquement tous la recherche du rattrapeur d'exceptions sur la notion de durée de vie dynamique. Lorsque l'on désire protéger un calcul des effets des exceptions, on associe à ce calcul et pendant toute la durée de ce calcul, une fonction de rattrapage d'exceptions. Si le calcul s'achève sans exception, alors le rattrapeur d'erreurs n'a plus lieu d'être et est inactivé. Si par contre, une exception survient, il faut invoquer le rattrapeur d'exceptions. Si l'utilisateur invoque le système d'exceptions, il fournira l'objet servant d'exception ; si par contre la situation est découverte par le système, alors il sera de son ressort que de construire l'exception.

Parmi les modèles d'exception, il existe des modèles avec ou sans reprise. Lorsque certaines exceptions sont signalées et, en particulier, celles que peut signaler le programmeur (`cerror` en COMMON LISP ou EULISP), il est quelquefois possible de reprendre le calcul à l'endroit même où il avait été interrompu. Dans de nombreux autres cas, ce n'est pas possible et la seule opération permise sur le contrôle, est de s'échapper vers un contexte sain.

Le modèle d'exception de COMMON LISP est plus que complet mais sa taille l'oppose au dessein de ce livre qui est d'en montrer l'essence. Le modèle de ML est un modèle sans reprise : lorsqu'une exception est signalée, la pile est rabaissée au niveau qu'elle avait lorsque le rattrapeur d'exceptions actif a été spécifié. Il est possible de relancer l'exception afin de bénéficier du rattrapeur précédent. Ce modèle ne convient pas car il perd l'environnement des variables dynamiques qui étaient présentes au moment de l'exception. Voici donc le modèle proposé, décrit d'abord informellement puis implanté, en espérant que les deux coïncident ! Il s'inspire fortement de celui d'EULISP.

La forme spéciale `monitor` permet d'associer une fonction de rattrapage d'exceptions au calcul correspondant à son corps. Sa syntaxe est donc :

```
(monitor rattrapeur formes... )
```

On ajoutera donc la forme spéciale `monitor` à l'analyseur syntaxique `meaning` :

```
... ((monitor) (meaning-monitor (cadr e) (cddr e) r tail?)) ...
```

Le rattrapeur est évalué et devient le rattrapeur d'exceptions courant. Les formes du corps de `monitor` sont alors évaluées comme si elles figuraient dans une forme `begin`. Si aucune exception n'est signalée pendant le calcul, la forme `monitor` retourne la valeur de la dernière forme de son corps et réactive le rattrapeur d'exceptions qui avait été masqué par la forme `monitor`.

Si une exception est signalée, alors on recherche le rattrapeur courant, c'est-à-dire celui qui a été associé par la forme `monitor` dynamiquement la plus proche (en terme d'implantation, on cherche le rattrapeur qui est le plus haut dans la pile d'évaluation). Sans changer la pile ni son contenu, on invoque le rattrapeur avec deux arguments qui sont un booléen (indiquant si l'exception est continuable) et l'objet représentant l'exception. Comme il faut prévoir que les rattrapeurs peuvent eux-mêmes être erronés, le rattrapeur courant s'exécute sous le contrôle du rattrapeur qu'il masquait. Le calcul mené par le rattrapeur peut soit s'échapper, soit retourner une valeur. Si l'exception était continuable, cette valeur devient la valeur attendue à l'endroit où l'exception avait été signalée. Si l'exception n'était pas continuable, une exception (non continuable) est signalée. Lorsque l'on sort du rattrapeur par échappement, le calcul est de nouveau contrôlé par le plus proche rattrapeur. Enfin, un rattrapeur de base existe (tout en bas de la pile) qui, si jamais il est invoqué, arrête le programme en cours et retourne au système d'exploitation ou ce qui en tient lieu (Figure 7.7).

Le prétraitement associé à la forme `monitor` emploie deux nouveaux générateurs qui gèrent la hiérarchie des rattrapeurs.

```
(define (meaning-monitor e e+ r tail?)
  (let ((m (meaning e r #f))
        (m+ (meaning-sequence e+ r #f)) )
    (append m (PUSH-HANDLER) m+ (POP-HANDLER)) ) )
(define (PUSH-HANDLER) (list 246))
(define (POP-HANDLER)  (list 247))
```

Nous y ajouterons, pour traiter le cas des exceptions non continuables et pourtant continuées, le générateur `NON-CONT-ERR`.

```
(define (NON-CONT-ERR) (list 245))
```

Les instructions `PUSH-HANDLER` et `POP-HANDLER` renvoient, pour leur définition, aux fonctions appropriées.

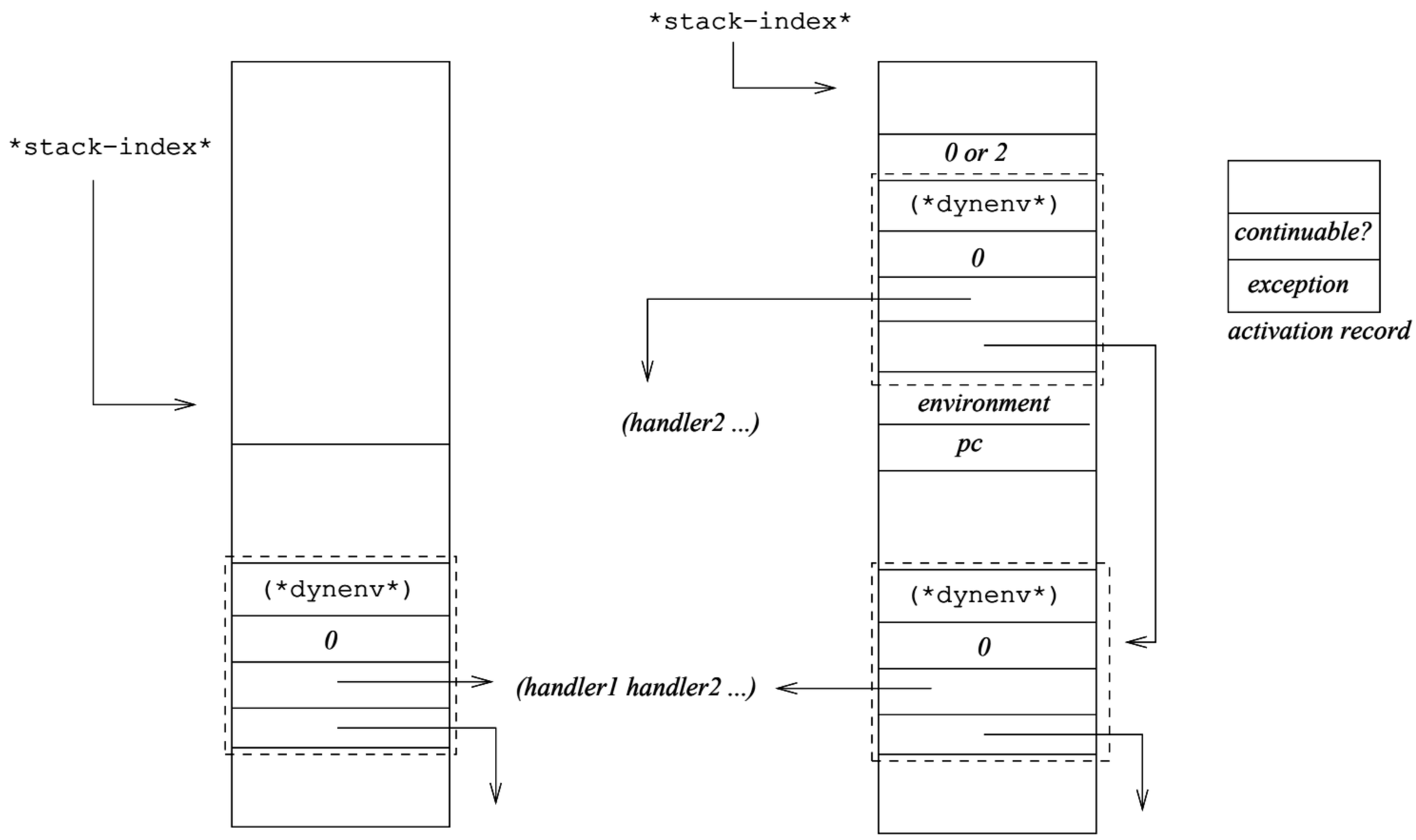


FIGURE 7.7 – Signalisation d'une exception

```
(define-instruction (PUSH-HANDLER) 246
  (push-exception-handler) )
(define-instruction (POP-HANDLER) 247
  (pop-exception-handler) )
```

Nous sommes maintenant à pied d'œuvre. Faire en sorte de déterminer quel est le rattrapeur le plus proche du sommet de pile est simple et s'apparente au mécanisme des liaisons dynamiques. Nous utiliserons donc ce mécanisme qui permet en outre de n'avoir pas à créer un nouveau registre contenant la liste des rattrapeurs actifs. C'est pourquoi on utilisera l'index 0 pour y stocker les rattrapeurs, car on aura pu constater que l'index 0 ne pouvait être attribué par la fonction `get-dynamic-variable-index`. Le point délicat est qu'un rattrapeur doit être invoqué sous le contrôle du rattrapeur précédent. Nous résolvons ce problème en associant à l'index 0, non pas le rattrapeur, mais la liste de tous les rattrapeurs, à charge lorsqu'une exception est signalée, d'empiler la liste correcte des rattrapeurs.

```
(define (search-exception-handlers)
  (find-dynamic-value 0) )
(define (push-exception-handler)
  (let ((handlers (search-exception-handlers)))
    (push-dynamic-binding 0 (cons *val* handlers)) ) )
(define (pop-exception-handler)
  (pop-dynamic-binding) )
```

Lorsqu'aucune exception n'est signalée, les rattrapeurs sont empilés puis dépilés sans perturber le régime d'évaluation. Les exceptions sont signalées par la fonction `signal-exception` que l'on avait déjà pu auparavant apercevoir, remplaçant l'antique `wrong` de naguère.

```
(define (signal-exception continuable? exception)
  (let ((handlers (search-exception-handlers))
```

```
    (v* (allocate-activation-frame (+ 2 1))) )
(set-activation-frame-argument! v* 0 continuable?)
(set-activation-frame-argument! v* 1 exception)
(set! *val* v*)
(stack-push *pc*)
(preserve-environment)
(push-dynamic-binding 0 (if (null? (cdr handlers)) handlers
                            (cdr handlers) ))
(if continuable?
    (stack-push 2)    ; pc for (POP-HANDLER) (RESTORE-ENV) (RETURN)
    (stack-push 0) )  ; pc for (NON-CONT-ERR)
(invoke (car handlers) #t) ) )
```

La fonction `signal-exception` est appelée avec un booléen (lui indiquant si elle doit appeler le rattrapeur de façon continuable ou non) et une valeur représentant l'exception. Toute valeur est possible ici et les exceptions prédéfinies ont ici été encodées par des listes dont le premier terme est une chaîne de caractères décrivant l'anomalie. COMMON LISP et EULISP réifient les exceptions en de vrais objets dont la classe est signifiante. La première chose est d'aller chercher la liste des rattrapeurs actifs puis d'allouer et de remplir un bloc d'activation pour l'appel au premier d'entre eux. La partie délicate est que l'on doit préparer la pile qui se trouve dans un état inconnu car on ignore où l'exception est survenue. On commence par sauvegarder le compteur ordinal et l'environnement (pour une éventuelle reprise), puis la nouvelle liste des rattrapeurs privée du premier d'entre eux. Comme il est nécessaire qu'il y en ait au moins un dans la liste, on s'arrangera pour que le programme débute avec un premier (et ultime) rattrapeur que l'on n'enlèvera jamais. Il faut bien sûr que ce rattrapeur ne commette aucune erreur, ce qui est le cas puisqu'il émet un message d'erreur et rend la main au système d'exploitation. Comment distinguer une erreur continuable d'une qui ne l'est pas ? Une façon simple est de supposer que la mémoire de code contient à des adresses fixes, les instructions adéquates. On avait déjà placé l'instruction `(FINISH)`, on y ajoutera quelques autres.

```
(define (code-prologue)
  (set! finish-pc 1)
  (append (NON-CONT-ERR) (FINISH) (POP-HANDLER)
          (RESTORE-ENV) (RETURN) ) )
```

Rendre le traitement de l'exception continuable ou pas est désormais simple et se représente par l'adresse de retour que l'on empile et qui sera trouvée par le `(RETURN)` qui clôt le rattrapeur. L'adresse `0` mène à `(NON-CONT-ERR)` qui signale une nouvelle exception, tandis que l'adresse `2` permet de dépiler ce que `signal-exception` avait empilé et de revenir d'où l'on était parti.

```
(define-instruction (NON-CONT-ERR) 245
  (signal-exception #f (list "Non continuable exception continued")) )
```

Il ne reste plus qu'à montrer comment lancer la machine malgré ces contraintes supplémentaires. La fonction `run-machine` sera juste enrichie de la pose implicite du rattrapeur ultime. Ce dernier pouvant être, par exemple, programmé pour s'arrêter après avoir imprimé son état final. Cela nous donne sans plus de détails :

```
(define (run-machine pc code constants global-names dynamics)
  (define base-error-handler-primitive
    (make-primitive base-error-handler) )
  (set! sg.current (make-vector (length global-names)
```

```
                                    undefined-value ))
  (set! sg.current.names    global-names)
  (set! *constants*         constants)
  (set! *dynamic-variables* dynamics)
  (set! *code*              code)
  (set! *env*               sr.init)
  (set! *stack-index*       0)
  (set! *val*               'anything)
  (set! *fun*               'anything)
  (set! *arg1*              'anything)
  (set! *arg2*              'anything)
  (push-dynamic-binding 0 (list base-error-handler-primitive))
  (stack-push finish-pc)                    ; pc for FINISH
  (set! *pc*                pc)
  (call/cc (lambda (exit)
             (set! *exit* exit)
             (run) )) )
(define (base-error-handler)
  (show-registers "Panic error: content of registers:")
  (wrong "Abort") )
```

La fonction `signal-exception` peut être rendue accessible aux programmes ; elle permet de signaler (fonctions `error`/`cerror` en COMMON LISP ou EULISP) ses propres exceptions et de les rattraper. Son coût devrait toutefois la limiter aux situations exceptionnelles.

Cette section a décrit un mécanisme de gestion d'exceptions. C'est un modèle avec reprise possible et qui invoque le rattrapeur dans l'environnement dynamique de la survenue de l'exception, ce qui donne plus de possibilités quant à une détection fine (et sans affectation) du contexte de l'anomalie. Il est par exemple possible d'écrire cette factorielle inusuelle :

```
(monitor (lambda (c e) ((dynamic foo) 1)
   (let fact ((n 5))
     (if (= n 0) (/ 11 0)
         (* n (bind-exit (k)
                (dynamic-let (foo k)
                   (fact (- n 1)) ) )) ) ) ) )
```

Pour ne pas perdre trop d'informations sur le contexte de l'anomalie, il est donc bon de ne pas s'échapper trop tôt.

L'emploi d'une machine virtuelle à code représenté par des octets simplifie certainement l'implantation du mécanisme de gestion des exceptions car celles-ci ne peuvent survenir que pendant le traitement des instructions qui sont bien individualisées et telles que les ressources de la machine sont toujours cohérentes entre deux instructions. Ce n'est pas le cas d'une vraie implantation qui peut recevoir des signaux asynchrones en des états imprévus.

On pourrait s'interroger sur le fait que certaines des exceptions prédéfinies sont continuables tandis que d'autres ne le sont pas. Une machine à octets est si régulière que toutes les exceptions peuvent être rendues continuables car elles sont toujours associables à un compteur ordinal bien défini. Une fois encore, ce n'est pas le cas d'une vraie implantation où soit la notion de compteur ordinal peut être floue, soit une anomalie, telle qu'une division par zéro, n'est détectable que bien après l'opération fautive. Il est probable que seules les exceptions signalées par l'utilisateur peuvent

être portablement continuables.

À ce titre, il est pertinent de s'interroger sur la nécessité de signaler les erreurs de prétraitement par `signal-exception` et de se demander en outre si ces erreurs sont continuables ou non. Comme le rattrapeur est alors sous la responsabilité du compilateur, on peut envisager que celui-ci utilise des exceptions continuables pour permettre la correction à la volée de certaines anomalies.

7.9 Compilation séparée

La compilation, telle qu'elle est pratiquée avec des langages comme C ou Pascal, passe par la compilation de fichiers. Cette section aborde ce thème et présente un tel compilateur assorti d'un lanceur d'exécutables autonomes. Y seront aussi présentés quelques problèmes d'édition de liens.

7.9.1 Compiler un fichier

La compilation de fichiers ne pose guère de problèmes. La fonction `compile-file` qui suit répond facilement à la question. Elle initialise les variables de compilation `g.current` qui représente l'environnement global modifiable, `*quotations*` qui collecte les citations et `*dynamic-variables*` qui rassemble les variables dynamiques. Elle compile le contenu du fichier considéré comme un gigantesque `begin` et sauve le résultat, c'est-à-dire les nouvelles valeurs des trois précédentes variables ainsi que le code dans le fichier résultat.

```
(define (read-file filename)
  (call-with-input-file filename
    (lambda (in)
      (let gather ((e (read in))
                   (content '()) )
        (if (eof-object? e)
            (reverse content)
            (gather (read in) (cons e content)) ) ) ) ) )
(define (compile-file filename)
  (set! g.current '())
  (set! *quotations* '())
  (set! *dynamic-variables* '())
  (let* ((complete-filename (string-append filename ".scm"))
         (e                 `(begin . ,(read-file complete-filename)))
         (code              (make-code-segment (meaning e r.init #t)))
         (global-names      (map car (reverse g.current)))
         (constants         (apply vector *quotations*))
         (dynamics          *dynamic-variables*)
         (ofilename         (string-append filename ".so")) )
    (write-result-file ofilename
                       (list ";;; Bytecode object file for "
                             complete-filename )
                       dynamics global-names constants code
                       (length (code-prologue)) ) ) )
(define (write-result-file ofilename comments dynamics global-names
                           constants code entry )
  (call-with-output-file ofilename
```

```
(lambda (out)
  (for-each (lambda (comment) (display comment out))
            comments )(newline out)(newline out)
  (display ";;; Dynamic variables" out)(newline out)
  (write dynamics out)(newline out)(newline out)
  (display ";;; Global modifiable variables" out)(newline out)
  (write global-names out)(newline out)(newline out)
  (display ";;; Quotations" out)(newline out)
  (write constants out)(newline out)(newline out)
  (display ";;; Bytecode" out)(newline out)(newline out)
  (write code out)(newline out)(newline out)
  (display ";;; Entry point" out)(newline out)
  (write entry out) (newline out) ) ) )
```

Pour ne pas s'encombrer des problèmes liés au système de fichiers et à la structure de leur nom, nous supposerons, comme en Scheme, que ceux-ci peuvent être désignés par de simples chaînes de caractères. Le compilateur prend une telle chaîne qui correspond au radical du nom du fichier, c'est-à-dire son nom privé de son extension habituelle, ici `".scm"`. Le résultat de la compilation sera stocké dans un fichier de même radical et d'extension `".so"`. Le résultat est formé du code produit, de la liste des noms des variables globales modifiables, de la liste des variables dynamiques, de la liste des citations et du compteur ordinal de la première instruction à exécuter dans le code. Toujours pour simplifier, nous constituerons le fichier de sortie par de simples écritures effectuées par `write`. On y adjoindra quelques commentaires pour aider la lecture qu'un humain pourrait faire du fichier résultat.

Si le fichier à compiler est :

si/example.scm

```
(set! fact
      ((lambda (fact) (lambda (n)
                        (if (< n 0)
                            "Toctoc la tete!"
                            (fact n fact (lambda (x) x)) ) ))
       (lambda (n f k)
         (if (= n 0)
             (k 1)
             (f (- n 1) f (lambda (r) (k (* n r)))) ) ) ) )
```

Alors le résultat de la compilation sera :

si/example.so

```
;;; Bytecode object file for si/example.scm
;;; Dynamic variables
()
;;; Global modifiable variables
(FACT)
;;; Quotations
#("Toctoc la tete!")
;;; Bytecode
#(245 20 247 38 43 40 30 59 74 32 1 34 81 35 106 31 10 3 34 82 34
  51 60 39 46 30 39 2 34 1 34 82 35 105 34 2 34 40 30 19 72 32 6 1
  2 34 6 1 0 34 1 35 109 34 51 60 39 46 43 34 53 62 61 60 39 46 43
```

```
   34 51 60 32 40 30 38 72 32 1 34 81 35 107 31 4 9 0 30 24 6 1 0 34
   1 34 6 1 0 34 40 30 4 72 32 1 43 34 53 62 61 60 39 46 43 33 27 0
   43)
;;; Entry point
5
```

7.9.2 Fabriquer une application

Compiler des fichiers est bel et bon mais doit néanmoins permettre leur exécution ultérieure. Le second utilitaire correspond à ce que l'on nomme souvent un *éditeur de liens* (le `ld` d'UN⋆X) et rassemble des fichiers compilés en un seul et unique fichier exécutable. La fonction `build-application` prend le nom du fichier à engendrer puis les noms des fichiers à relier ensemble.

```
(define (build-application application-name ofilename . ofilenames)
  (set! sg.current.names    '())
  (set! *dynamic-variables* '())
  (set! sg.current          (vector))
  (set! *constants*         (vector))
  (set! *code*              (vector))
  (let install ((filenames (cons ofilename ofilenames))
                (entry-points '()) )
    (if (pair? filenames)
        (let ((ep (install-object-file! (car filenames))))
          (install (cdr filenames) (cons ep entry-points)) )
        (write-result-file application-name
                           (cons ";;; Bytecode application containing "
                                 (cons ofilename ofilenames) )
                           *dynamic-variables*
                           sg.current.names
                           *constants*
                           *code*
                           entry-points ) ) ) )
```

Le cœur du travail est effectué par `install-object-file!` qui installe un fichier compilé au sein des cinq variables gouvernant la machine, à savoir `sg.current.names` qui nomme les variables globales modifiables, `*dynamic-variables*` qui nomme les variables dynamiques, `sg.current` qui contient les valeurs des variables globales modifiables, `*constants*` qui contient l'ensemble des citations et `*code*` qui est le vecteur des instructions. Après installation de tous ces fichiers, il suffit d'écrire le fichier exécutable résultant sous une forme semblable à celle des fichiers compilés, excepté le point d'entrée qui devient une liste de points d'entrée.

La fonction `install-object-file!` installe un fichier compilé et retourne l'adresse de sa première instruction. Placer le code du fichier à installer dans le vecteur `*code*` est simple, ce qui l'est moins est de faire en sorte que les fichiers partagent ce qu'ils ont en commun et protègent ce qu'ils ont en propre. Ce sera l'objet des fonctions dont le nom commence par `relocate` et qui apparaissent dans la définition suivante :

```
(define (install-object-file! filename)
  (let ((ofilename (string-append filename ".so")))
    (if (probe-file ofilename)
        (call-with-input-file ofilename
```

```
(lambda (in)
  (let* ((dynamics     (read in))
         (global-names (read in))
         (constants    (read in))
         (code         (read in))
         (entry        (read in)) )
    (close-input-port in)
    (relocate-globals! code global-names)
    (relocate-constants! code constants)
    (relocate-dynamics! code dynamics)
    (+ entry (install-code! code)) ) ) )
      (signal #f (list "No such file" ofilename)) ) ) )
(define (install-code! code)
  (let ((start (vector-length *code*)))
    (set! *code* (vector-append *code* code))
    start ) )
```

Les citations, par exemple, sont particulières à chaque fichier et ne doivent pas être confondues. Elles sont, dans chaque fichier, numérotées à partir de zéro et donc partagent les mêmes numéros. Les citations seront concaténées dans la variable nommée `*constants*` et les numéros y référant seront mis à jour. Ces numéros apparaissent dans le vecteur des instructions comme argument de l'instruction `CONSTANT`, c'est donc là qu'il faut les mettre à jour et, pour cela, on scrute séquentiellement de part en part le vecteur de code.

```
(define CONSTANT-code 9)
(define (relocate-constants! code constants)
(define n (vector-length *constants*))
(let ((code-size (vector-length code)))
  (let scan ((pc 0))
    (when (< pc code-size)
      (let ((instr (vector-ref code pc)))
        (when (= instr CONSTANT-code)
          (let* ((i (vector-ref code (+ pc 1)))
                 (quotation (vector-ref constants i)) )
            (vector-set! code (+ pc 1) (+ n i)) ) )
        (scan (+ pc (instruction-size code pc))) ) ) ) )
(set! *constants* (vector-append *constants* constants)) )
```

En ce qui concerne les variables globales, il faut au contraire faire en sorte de les partager. Deux fichiers parlant de la variable `foo` doivent partager cette variable. Les variables sont, elles, numérotées par rapport à une liste locale de noms. Les seules places où apparaissent ces numéros sont comme arguments des instructions `GLOBAL-REF`, `CHECKED-GLOBAL-REF` et `SET-GLOBAL!`. Chaque numéro trouvé est associé à son nom externe, qui lui est associé au numéro qu'aura la variable dans l'exécutable ; ce dernier enfin remplace ce premier.

```
(define CHECKED-GLOBAL-REF-code 8)
(define GLOBAL-REF-code 7)
(define SET-GLOBAL!-code 27)
(define (relocate-globals! code global-names)
  (define (get-index name)
    (let ((where (memq name sg.current.names)))
      (if where (- (length where) 1)
          (begin (set! sg.current.names (cons name sg.current.names))
                 (get-index name) ) ) ) )
```

```
(let ((code-size (vector-length code)))
  (let scan ((pc 0))
    (when (< pc code-size)
      (let ((instr (vector-ref code pc)))
        (when (or (= instr CHECKED-GLOBAL-REF-code)
                  (= instr GLOBAL-REF-code)
                  (= instr SET-GLOBAL!-code) )
          (let* ((i (vector-ref code (+ pc 1)))
                 (name (list-ref global-names i)) )
            (vector-set! code (+ pc 1) (get-index name)) ) )
        (scan (+ pc (instruction-size code pc))) ) ) ) )
  (let ((v (make-vector (length sg.current.names) undefined-value)))
    (vector-copy! sg.current v 0 (vector-length sg.current))
    (set! sg.current v) ) )
```

Le processus est similaire pour les variables dynamiques :

```
(define DYNAMIC-REF-code 240)
(define DYNAMIC-PUSH-code 242)
(define (relocate-dynamics! code dynamics)
  (for-each get-dynamic-variable-index dynamics)
  (let ((dynamics (reverse! dynamics))
        (code-size (vector-length code)) )
    (let scan ((pc 0))
      (when (< pc code-size)
        (let ((instr (vector-ref code pc)))
          (when (or (= instr DYNAMIC-REF-code)
                    (= instr DYNAMIC-PUSH-code) )
            (let* ((i (vector-ref code (+ pc 1)))
                   (name (list-ref dynamics (- i 1))) )
              (vector-set! code (+ pc 1)
                           (get-dynamic-variable-index name) ) ) )
          (scan (+ pc (instruction-size code pc))) ) ) ) ) )
```

On notera dans ces fonctions l'emploi de la fonction `instruction-size`. On remarquera aussi que scruter par trois fois le vecteur de code est inutile et pourrait être factorisé en une seule passe.

La forme adoptée pour les fichiers compilés permet d'imaginer asssez simplement de nouveaux modes d'agrégation de fichiers. La liste des variables globales modifiables constitue en quelque sorte l'interface du fichier considéré comme un module. Ce module exporte toutes ses variables globales modifiables sous le nom qu'elles avaient dans le fichier. On peut simplement renommer ces variables ou restreindre leur ensemble. On peut même inventer un langage d'édition de liens spécifiant comment agglomérer des modules et gérer les noms des variables qu'ils utilisent. Esquissons un peu plus ce que pourrait être ce langage inspiré de celui proposé pour les modules d'EuLisp [QP91a]. En guise d'exemple, la directive suivante définit ce qu'importe un module nommé `mod` :

```
(ordered-union
   (only (fact) (expose "fact"))
   (union (except-pattern ("fib*") (expose "fib"))
          (rename ((call/cc call-with-current-continuation))
            (expose "scheme") )
          (expose "numeric") ) )
```

Supposons que la notation `foo@module` désigne la variable de nom `foo` dans le module `module`. Supposons en outre que le module `fact` définisse les variables `fact`

et `fact100` (contenant la valeur précalculée de `(fact 100)`, valeur souvent demandée), que le module `fib` définisse les variables `fib`, `fib20` et `Fibonacci`. Le module `numeric` procure des fonctions comme `fact` et `fib` tandis que `scheme` procure toutes les fonctions du R4RS. Le module produit par cette directive est ainsi formé : il contient la variable `fact@fact` (la variable `fact20@fact`, quoiqu'exposée par la directive `(expose "fact")`, est exclue du fait de la directive restreignante `only`), la seule variable `Fibonacci@fib` (les autres variables du même module ont des noms exclus par la clause `except-pattern`). La variable `call/cc@scheme` renomme la variable `call-with-current-continuation@scheme`. Comme en outre l'union créant `mod` est spécifiée comme ordonnée, le module `fact` sera évalué avant les autres (`fib`, `numeric` et `scheme`), dont l'ordre n'est pas spécifié mais doit être compatible avec leur définition : le module `numeric` utilise fort probablement le module `scheme` et doit donc être évalué après.

7.9.3 Exécuter une application

La fonction d'un exécutable est d'être exécuté. Comme on a tout préparé, le processus devient simple mais long et scrupuleux en ce qui concerne l'initialisation raisonnée de tous les registres.

```
(define (run-application stack-size filename)
  (if (probe-file filename)
      (call-with-input-file filename
    (lambda (in)
      (let* ((dynamics     (read in))
             (global-names (read in))
             (constants    (read in))
             (code         (read in))
             (entry-points (read in)) )
        (close-input-port in)
        (set! sg.current.names    global-names)
        (set! *dynamic-variables* dynamics)
        (set! sg.current (make-vector (length sg.current.names)
                                      undefined-value ))
        (set! *constants*         constants)
        (set! *code*              (vector))
        (install-code! code)
        (set! *env*               sr.init)
        (set! *stack*             (make-vector stack-size))
        (set! *stack-index*       0)
        (set! *val*               'anything)
        (set! *fun*               'anything)
        (set! *arg1*              'anything)
        (set! *arg2*              'anything)
        (push-dynamic-binding
         0 (list (make-primitive (lambda ()
                                   (show-exception)
                                   (*exit* 'aborted) ))) )
        (stack-push 1)                          ; pc for FINISH
        (if (pair? entry-points)
            (for-each stack-push entry-points)
            (stack-push entry-points) ) )
    (set! *pc* (stack-pop))
```

```
        (call/cc (lambda (exit)
                   (set! *exit* exit)
                   (run) )) ) )
    (static-wrong "No such file" filename) ) )
```

On commence par lire le fichier contenant l'exécutable et on ferme le port de sortie devenu inutile aussitôt. On initialise les variables globales de la machine (comme le faisait autrefois la fonction `run-machine`). Après avoir réservé de la place pour le rattrapeur d'exceptions de base, on empile tous les points d'entrée des fichiers compilés participant à l'exécutable. Pour être un peu plus général, on permet aussi de considérer un simple fichier compilé comme un exécutable à lui tout seul. Le rattrapeur par défaut termine l'exécution si invoqué, il capture pour cela la continuation d'appel de `run-application` (le système d'exploitation) et la place dans la variable `*exit*`. La fonction `(show-exception)` est sensée imprimer un message d'erreur significatif[14] à partir de l'exception présente dans le registre `*val*`. Il ne reste plus qu'à simuler un `(RETURN)` pour évaluer le premier fichier composant l'exécutable qui lui-même enchaînera sur le second et ainsi de suite.

On notera que la fonction `run-application` prend en argument la taille de la pile d'évaluation à utiliser. Tenir dans les limites de la pile est un point d'implantation subtil mais de grande importance. Si l'on devait, à chaque `stack-push`, tester que l'on ne déborde pas de la pile serait par trop coûteux ; on peut parfois utiliser le système d'exploitation ou les propriétés des mémoires segmentées pour éliminer ce test et notamment accroître la taille de la pile en cas de dépassement. Ici notre simulation est particulièrement inefficace car on ne dira jamais assez que la forme `(vector-ref` v i`)` est assez coûteuse puisqu'elle doit vérifier que v est un vecteur, i un entier, positif et inférieur à la taille du vecteur.

La fonction `run-application` n'a pas besoin de grand chose pour exécuter un fichier compilé. Elle n'a en gros besoin que de `run` et des fonctions relatives aux vecteurs, listes et autres classes comme `primitive`, `continuation` etc. Elle a aussi besoin de la fonction `read` pour lire les citations. Elle s'affranchit donc complètement du compilateur ayant produit l'application mais ne procure aucune aide à la mise au point du programme : c'était le but !

7.10 Conclusions

Pour obtenir une véritable implantation de Scheme, il faudrait ajouter les nombreuses fonctions manquantes ainsi que les types de données correspondants : cela ne présente guère de difficultés autres que d'être soigneux pour bien détecter les données incorrectes de type ou de domaine erroné.

L'implantation décrite plus haut est assez générale et peut bien sûr être encore améliorée. Son intérêt est d'être dérivée assez simplement de l'interprète efficace précédent, interprète que l'on a reconfiguré en compilateur. Il existe en fait un lien profond entre interprète et compilateur (exploré en [Nei84]) : le premier exécute un programme tandis que le second le transcrit en quelque chose qui sera exécuté. C'est

---

14. Par exemple, `"bus error, core not dumped"`.

donc une simple différence de bibliothèque (d'exécution ou de génération) qui est en cause. Nous avons usé de ce lien pour dériver le compilateur de l'interprète.

En revanche, nous avons hérité des seules informations qui figuraient dans le langage intermédiaire et elles sont insuffisantes pour assurer une bonne compilation. La principale analyse qui nous fait défaut est que l'on ne sait rien sur l'emploi des variables locales. On ignore si ces variables sont modifiables, closes, multiplement closes, modifiées lorsque closes, etc. Toutes ces propriétés (statiques) permettent, dans de nombreux cas, de se dispenser des blocs d'activation pour n'utiliser que la pile d'où un gain important en termes d'allocation et donc en vitesse. Les blocs d'activation ne sont utiles que pour conserver les valeurs de variables closes ; il n'est donc pas nécessaire d'allouer de tels blocs lorsque les variables ne sont pas closes. Puisque leurs valeurs sont déjà dans la pile, autant les y laisser et les y chercher, ce qui n'améliorera pas peu la vitesse du tout.

7.11 Exercices

Exercice 7.1 : Modifiez le compilateur décrit dans ce chapitre pour utiliser un registre `*dynenv*` pour désigner l'environnement dynamique. **[☞ p. 240]**

Exercice 7.2 : Définissez une fonction `load` chargeant à l'exécution un programme compilé et l'exécutant. Si par exemple `fact.scm` est un fichier compilé en `fact.so`, on pourra compiler et exécuter le fichier suivant :

```
(begin (load "fact")
       (fact 5) )
```

Exercice 7.3 : Écrivez une fonction `global-value` prenant un nom de variable globale et retournant sa valeur.

Exercice 7.4 : Modifiez les instructions liées aux variables dynamiques afin de les implanter par liaison superficielle. **[☞ p. 22]**

Exercice 7.5 : **[☞ p. 252]** Écrivez un renommeur de variables exportées. Si `fact.so` est un fichier compilé, alors :

```
(build-application-renaming-variables
  "nfact.so" "fact.so" '((fact factorial)) )
```

crée un nouveau module `nfact.so` où la variable `fact` a été renommée en `factorial`.

Exercice 7.6 : **[☞ p. 186]** Modifiez l'instruction `CHECKED-GLOBAL-REF` pour se modifier en `GLOBAL-REF` dès que la variable lue est initialisée.

Projet 7.7 : Gnu Emacs Lisp [LLSt93], xscheme [Bet91] ont, entre autres implantations de Lisp ou Scheme, des compilateurs de code-octets. Adaptez le compilateur du présent chapitre aux interprètes de code-octet implantés par ces machines virtuelles.

7.12 Lectures recommandées

Peu d'ouvrages exposent des rudiments de compilation (d'où celui-ci), on consultera cependant [All78], [Hen80] et pour une compilation d'ordre encore plus supérieur [Dil88]. Contiennent aussi des compilateurs intéressants et commentés [Ste78, AG85, FWH92].

8

Évaluation et réflexivité

Une caractéristique unique de Lisp est la disponibilité d'un mécanisme d'évaluation : `eval`. Quoique ce livre n'ait fait que parler d'évaluation, jamais encore n'avait été abordé le problème de sa mise à la disposition des programmeurs. L'évaluation pose de nombreux problèmes de spécification, d'intégrité aussi bien que linguistiques : c'est ce que condensent nos amis d'outre-Atlantique avec le laconique « `eval` *is evil* ». Encapsuler son génie sous une forme utilisable est un premier pas vers la réflexivité programmatique que ce chapitre présentera en une seconde partie.

Depuis maintenant plus de deux cents pages, nous présentons de multiples interprètes détaillant les tréfonds du mécanisme d'évaluation. Pour la plupart d'entre eux, rendre accessible aux programmeurs le mécanisme d'évaluation est trivial et ne nécessite que très peu de code. C'est ce qu'ont fait les implanteurs depuis des lustres. Toutefois l'existence d'un tel mécanisme, si elle va théoriquement de soi, **[☞ p. 2]** n'en a pas moins été un des buts de la création de Lisp. L'explicitation de la fonction `eval` apparaît en effet dès le tout début des années soixante dans les écrits fondateurs [McC60, MAE+62].

Le mécanisme d'évaluation explicite est un mécanisme fondamental qui autorise de nombreux effets et notamment un puissant système de macros, l'immersion de l'environnement de programmation au sein du langage et un caractère réflexif prononcé. Ce mécanisme ne va pas sans méfait et notamment les macros, la non-séparation entre environnement et langage et la quasi cancéreuse invasion réflexive. Comme toute puissante magie, l'évaluation explicite ne peut utilement et sans danger être mise entre toutes les mains !

Quel peut être le contrat que doit satisfaire le mécanisme d'évaluation ? Manifestement, l'on aimerait observer que :

$$\texttt{(eval '}\pi\texttt{)} \quad \equiv \quad \pi \tag{1}$$

Mais nous allons bientôt démontrer l'ambiguïté de cette formulation.

8.1 Programmes et valeurs

Tentons matériellement d'incorporer `eval` au sein du premier interprète de ce livre. **[☞ p. 3]** Cet interprète était rédigé en Scheme pur sans restriction particu-

lière. Supposons tout d'abord que `eval` soit une forme spéciale, alors elle apparaîtra en `evaluate` qui deviendra :

```
(define (evaluate e env)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e)(string? e)(char? e)(boolean? e)) e)
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote)  (cadr e))
        ((if)     (if (not (eq? (evaluate (cadr e) env)
                                the-false-value ))
                      (evaluate (caddr e) env)
                      (evaluate (cadddr e) env) ))
        ((begin)  (eprogn (cdr e) env))
        ((set!)   (update! (cadr e) env (evaluate (caddr e) env)))
        ((lambda) (make-function (cadr e) (cddr e) env))
        ((eval)   (evaluate (evaluate (cadr e) env) env)) ; ** Modified **
        (else     (invoke (evaluate (car e) env)
                          (evlis (cdr e) env) )) ) ) )
```

La forme spéciale `eval` commence par évaluer la forme qui correspond à son premier argument (tout comme le ferait une fonction) puis évalue la valeur résultante dans l'environnement courant. Cette éminemment triviale description masque des abîmes interrogatifs.

Comme maintes fois souligné, la fonction `evaluate` contient un premier étage d'analyse syntaxique et un second d'évaluation. C'est ce que nous avions séparé en les fonctions `meaning` et `run` dans les tous derniers interprètes. Le premier argument de `evaluate` correspond à un programme tandis que son résultat appartient au domaine des valeurs. Il y a donc un problème, disons de « typage », avec la forme `(evaluate (evaluate ...) ...)`, où l'`evaluate` englobant est appliqué à une valeur et non à un programme. Les programmes sont-ils des valeurs ; les valeurs sont-elles des programmes ?

Les langages de programmation sont habituellement décrits par une grammaire précisant leur forme syntaxique. La grammaire de Scheme apparaît en [CR91b] et elle spécifie que certains agencements de parenthèses et de lettres sont des programmes syntaxiquement légaux. La grammaire spécifie aussi la syntaxe des données et l'on peut vérifier que tout programme satisfait à la syntaxe des données ; c'est pourquoi la fonction `read` est universelle en ce sens qu'elle peut aussi bien servir à lire des programmes que des données. Néanmoins, il n'est nullement obligatoire que les programmes soient lus par `read` pour être évalués, même si c'est plus simple. Un évaluateur à la Smalltalk [GR83] lit ses programmes dans des fenêtres avec un lecteur spécial qui mémorise les positions (en termes de lignes et de colonnes) des dits programmes de manière à faire ressortir les portions syntaxiquement incorrectes. Mais puisque l'usage prévaut, il est bien sûr clair que tout programme respectant la grammaire de Scheme est (ou peut être associé à) une valeur légale selon cette même grammaire.

Il est non moins clair qu'inversement de nombreuses valeurs correspondent à des programmes mais que d'encore plus nombreuses autres n'en sont pas comme, par exemple, `(quote . 1)`. Il reste malheureusement une frange de valeurs pour lesquelles le statut n'est pas aussi limpide :

— Une valeur ressemblant grosso modo à un programme mis à part quelques sco-

ries locales (par exemple `(if #t 1 (quote . 2))`) est-elle un programme ? Cette expression ne pose aucun problème à la définition opérationnelle ci-dessus donnée pour `eval`, puisque `(quote . 2)` n'est pas évaluée, mais ne répond pas à la grammaire des programmes de Scheme.

— Une valeur telle que celle construite par l'expression `` `(',car '(a b)) `` et comportant en position fonctionnelle une forme citant la valeur de la fonction `car` est-elle un programme ? Cette valeur n'est pas un programme conforme syntaxiquement au R^{4}RS car il n'a pas de représentation externe : il ne peut être typographié avec un clavier normal ! Mais encore une fois, il ne pose aucun problème pour l'`eval` ci-avant énoncé.

— La structure en termes de doublets d'une valeur telle que celle construite par l'expression suivante (avec les conventions syntaxiques de COMMON LISP où `#1=` est un nom pour l'expression qui suit, tandis que `#1#` représente l'expression de nom `1`) est contemplable à la figure 8.1.

```
(let ((n 4))
  #1=(if (= n 1) 1
         (* n ((lambda (n) #1#) (- n 1))) ) )
```

Cette valeur, comportant un cycle (on dit que ce programme est syntaxiquement récursif), est donc un programme syntaxiquement illégal aux yeux de Scheme. Une fois de plus, elle ne pose pas de problème à la forme `eval` précédemment vue.

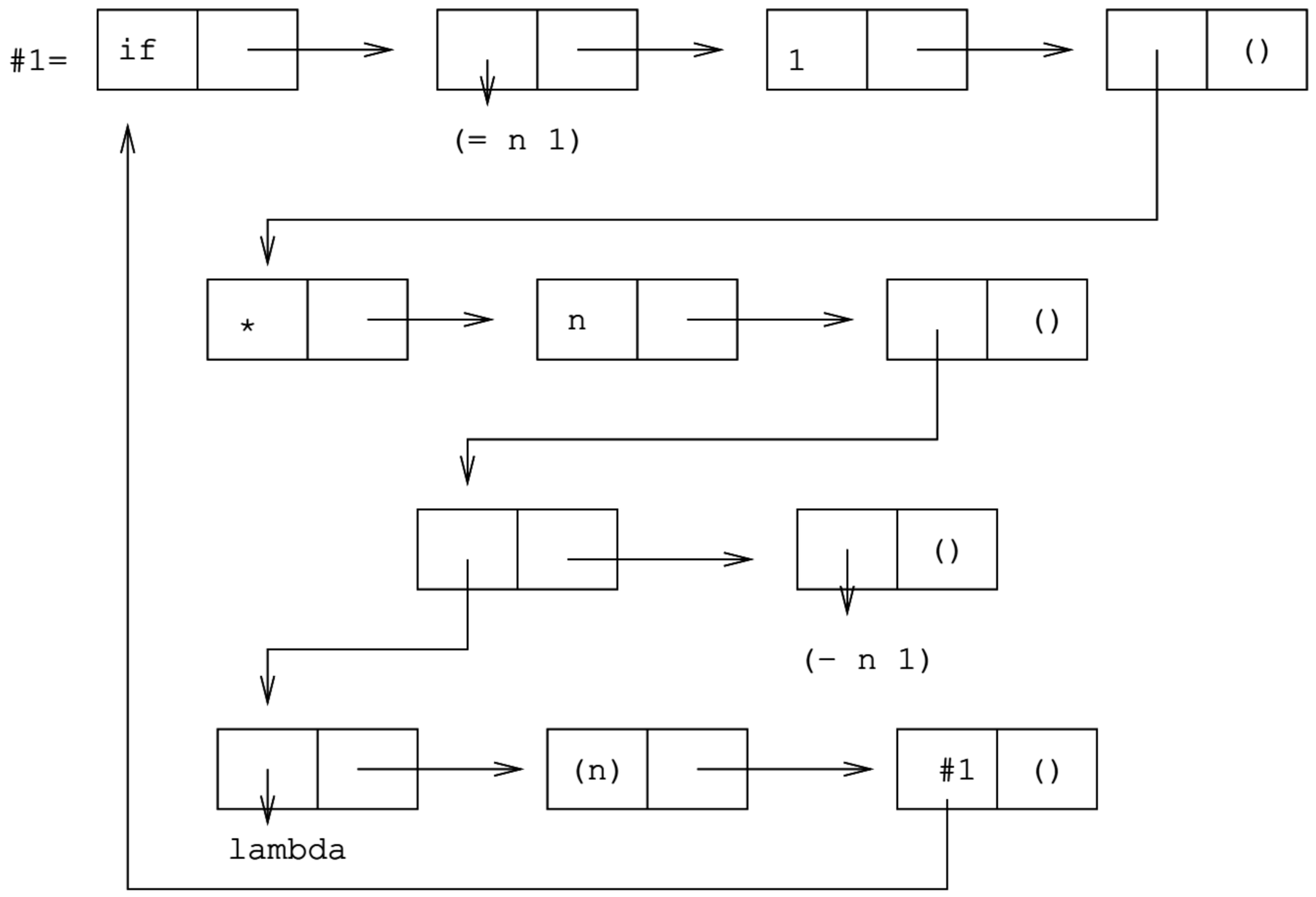


FIGURE 8.1 – Factorielle syntaxiquement récursive

Toutes ces valeurs, fortement tératologiques à l'aune de la théorie des grammaires, montrent bien que la notion de programme est délicate en présence d'`eval`. Le même problème se pose avec les macros qui effectuent des calculs sur les représentations des programmes. On retrouve aussi ici les distinctions entre erreur statique et er-

reur dynamique que nous avions auparavant discutées. **[☞ p. 186]** Tracer une ligne est difficile mais nous nous y risquerons ! L'hypothèse la plus laxiste est de tout permettre : une anomalie syntaxique (comme `(quote . 3)`) n'est décelée que lorsqu'il est obligatoire d'évaluer ce programme. L'hypothèse la plus contraignante est de n'accepter que des programmes représentés par des graphes acycliques finis et dont tous les nœuds syntaxiques sont corrects. Cela élimine la factorielle syntaxiquement récursive (quoique syntaxiquement correcte) donnée en exemple un peu plus haut. C'est cette ligne dure, celle de Scheme, que nous retiendrons puisqu'il est montré en [Que92a] que les programmes syntaxiquement récursifs peuvent être réécrits sous forme purement arborescente sans cycle. En ce qui concerne les citations, l'on autorisera toute valeur finie dont les atomes (les feuilles) ont une représentation externe. Cela exclut les citations comportant des fermetures, des continuations ou des flux ainsi que les structures cycliques [1].

Nous caractériserons opérationnellement les valeurs que nous admettons comme programmes légaux grâce aux prédicats suivants qui testent syntaxe et non-cyclicité [2] :

```
(define (program? e)
  (define (program? e m)
    (if (atom? e)
        (or (symbol? e) (number? e) (string? e)
            (char? e) (boolean? e) )
        (if (memq e m) #f
            (let ((m (cons e m)))
(define (all-programs? e+ m+)
  (if (memq e+ m+) #f
      (let ((m+ (cons e+ m+)))
        (and (pair? e+)
             (program? (car e+) m)
             (or (null? (cdr e+))
                 (all-programs? (cdr e+) m+) ) ) ) ) ) )
(case (car e)
  ((quote) (and (pair? (cdr e))
                (quotation? (cadr e))
                (null? (cddr e)) ))
  ((if) (and (pair? (cdr e))
             (program? (cadr e) m)
             (pair? (cddr e))
             (program? (caddr e) m)
             (pair? (cdddr e))
             (program? (cadddr e) m)
             (null? (cddddr e)) ))
((begin) (all-programs? (cdr e) '()))
((set!) (and (pair? (cdr e))
             (symbol? (cadr e))
             (pair? (cddr e))
             (program? (caddr e) m)
             (null? (cdddr e)) ))
((lambda) (and (pair? (cdr e))
               (variables-list? (cadr e))
               (all-programs? (cddr e) '()) ))
```

---

1. Les permettre n'est pas très coûteux et peut servir, par exemple, dans l'implantation de MEROONET.
2. Notez que la valeur `(let ((e '(x))) `(lambda ,e ,e))` est un programme légal.

```
                (else (all-programs? e '())) ) ) ) ) )
  (program? e '()) )
(define (variables-list? v*)
  (define (variables-list? v* already-seen)
    (or (null? v*)
        (and (symbol? v*) (not (memq v* already-seen)))
        (and (pair? v*)
             (symbol? (car v*))
             (not (memq (car v*) already-seen))
             (variables-list? (cdr v*)
                              (cons (car v*) already-seen) ) ) ) )
  (variables-list? v* '()) )
(define (quotation? e)
  (define (quotation? e m)
    (if (memq e m) #f
        (let ((m (cons e m)))
          (or (null? e)(symbol? e)(number? e)
              (string? e)(char? e)(boolean? e)
              (and (vector? e)
                   (let loop ((i 0))
                     (or (>= i (vector-length e))
                         (and (quotation? (vector-ref e i) m)
                              (loop (+ i 1)) ) ) ) )
              (and (pair? e)
                   (quotation? (car e) m)
                   (quotation? (cdr e) m) ) ) ) ) )
  (quotation? e '()) )
```

Muni du prédicat `program?`, nous pouvons améliorer la formulation de la forme spéciale `eval` en :

```
... ((eval) (let ((v (evaluate (cadr e) env)))
              (if (program? v)
                  (evaluate v env)
                  (wrong "Illegal program" v) ) ))  ...
```

La forme est cependant encore maladroite car les prédicats précédents ne testent pas qu'une valeur *est* un programme ou une citation correcte mais seulement qu'une valeur *est une représentation légale* d'un programme ou d'une citation. C'est un gage d'intelligibilité d'une valeur dans un rôle particulier. La valeur n'est ni le programme, ni la citation : ces dernières sont ce que les besoins de l'interprète exigent qu'ils soient. Pour affiner ces notions, intéressons-nous maintenant à l'interprète pré-dénotationnel du chapitre 4, dans lequel **[☞ p. 105]** mémoires et continuations étaient explicites et au sein duquel les données du Scheme interprété étaient représentées par des fermetures simulant des objets. Voici cet interprète auquel on a ajouté la forme spéciale `eval` :

```
(define (evaluate e r s k)
  (if (atom? e)
      (if (symbol? e) (evaluate-variable e r s k)
          (evaluate-quote e r s k) )
      (case (car e)
        ((quote)  (evaluate-quote (cadr e) r s k))
        ((if)     (evaluate-if (cadr e) (caddr e) (cadddr e) r s k))
        ((begin)  (evaluate-begin (cdr e) r s k))
        ((set!)   (evaluate-set! (cadr e) (caddr e) r s k))
        ((lambda) (evaluate-lambda (cadr e) (cddr e) r s k))
```

```
        ((eval)   (evaluate-eval (cadr e) r s k))        ; ** Modified **
        (else     (evaluate-application (car e) (cdr e) r s k)) ) ) )
(define (evaluate-eval e r s k)
  (evaluate e r s
    (lambda (v ss)
      (let ((ee (transcode-back v ss)))
        (if (program? ee)
            (evaluate ee r ss k)
            (wrong "Illegal program" ee) ) ) ) ) )
```

Dans cet interprète, la valeur obtenue par évaluation du premier argument de la forme `eval` est d'abord décodée sous sa forme externe (par `transcode-back`) puis sa nature de programme est vérifiée pour être enfin évaluée. Les variables `e` et `ee` ont pour domaine les descriptions de programmes tandis que `v` nomme des valeurs. C'est le rôle de `transcode-back` que de prendre une valeur et de la mettre en forme de telle manière qu'elle puisse être considérée comme une description de programme. Ce mécanisme ressemble tout à fait au truc qui permet d'obtenir l'effet d'`eval` dans un Scheme où il n'y a que `load` de disponible : écrivez la valeur à évaluer dans un fichier (usez de `display` pour `transcode-back`), puis évaluez ce fichier (par `load` au lieu de `evaluate`). Dans ce cas, le programme est « compilé » en un nom de fichier[3].

Un dernier exemple fourni par l'interprète compilant rapide du chapitre 6 révèlera encore quelques points. **[☞ p. 175]** Là, valeurs et descriptions de programmes partagent le même support ; il ne sera donc pas nécessaire d'introduire de conversion entre eux. Par contre, le but de l'interprète rapide était de supprimer de l'évaluation tout calcul statique pouvant être auparavant effectué.

```
(define (meaning e r tail?)
  (if (atom? e)
      (if (symbol? e) (meaning-reference e r tail?)
                      (meaning-quotation e r tail?) )
      (case (car e)
        ((quote)  (meaning-quotation (cadr e) r tail?))
        ((lambda) (meaning-abstraction (cadr e) (cddr e) r tail?))
        ((if)     (meaning-alternative (cadr e) (caddr e) (cadddr e)
                                       r tail? ))
        ((begin)  (meaning-sequence (cdr e) r tail?))
        ((set!)   (meaning-assignment (cadr e) (caddr e) r tail?))
        ((eval)   (meaning-eval (cadr e) r tail?))        ; ** Modified **
        (else     (meaning-application (car e) (cdr e)
                                       r tail? )) ) ) )
(define (meaning-eval e r tail?)
  (let ((m (meaning e r #f)))
    (lambda ()
      (let ((v (m)))
        (if (program? v)
            (let ((mm (meaning v r tail?)))
              (mm) )
            (wrong "Illegal program" v) ) ) ) ) )
```

Il devient maintenant clair que, pour cet interprète compilant, qui convertit une description de programme en un arbre de suspensions, la compilation de la valeur `v` ne peut être rendue statique. Pour la première fois, une suspension sera engendrée qui au lieu de ne contenir que des calculs simples comme des allocations, des alternatives

3. En fait, le codage d'`eval` par `load` ne fournit qu'un `eval` *at toplevel* que l'on abordera plus loin.

ou des séquences, des accès en lecture ou écriture au sein de structures de données variées, comporte, *horresco referens*, un appel à `meaning` ! Cet appel implique donc la nécessité de conserver, à l'exécution, le code de la fonction `meaning` et consœurs. C'est particulièrement net en ce qui concerne le compilateur d'octets pour lequel les modifications ne sont pas peu importantes.

La confusion entre valeurs et programmes, que nous avons tenté de lever, est du même ordre que la confusion entre citation et valeur. Ce n'est pas sans raison que `eval` et `quote` sont souvent dits avoir des rôles inverses : `quote` a pour argument une description de valeur à synthétiser tandis que `eval` part d'une valeur et la convertit en un programme à évaluer. L'une et l'autre opèrent des conversions entre signifiant et signifié, qu'il importe de ne point confondre.

8.2 `eval` comme forme spéciale

Lorsque `eval` est procurée comme une forme spéciale, on s'approche assez de l'idéal prêché en (1), c'est-à-dire rendre `(eval (quote π))` équivalent à π. C'est tout à fait ce que l'on souhaite si l'on adopte une pensée équationnelle [FH89, Mul92]. Dans une théorie bien construite, on peut substituer deux égaux entre eux *dans tout contexte*, ce qui implique plus précisément que (1) est une forme atténuée de (2), où $C[\,]$ représente un contexte :

$$\forall C[\,],\quad C[\texttt{(eval '}\pi\texttt{)}] \quad\equiv\quad C[\pi] \tag{2}$$

Ainsi non seulement l'évaluation d'une forme close (sans variable libre) comme `(eval '((lambda (x y) x) 1 2))` retourne `1` mais on peut aussi bénéficier de l'environnement lexical courant, comme le montrent les formes qui suivent :

```
((lambda (x) (eval 'x))
 3 )                                → 3
((lambda (x y) (eval y))
 4 'x )                             → 4
((lambda (x y z) (eval y))
 5 (list 'eval 'z) 'x )             → 5
```

Cet effet n'est rendu possible que parce que l'environnement lexical courant est disponible. On peut même constater sur l'interprète compilant rapide qu'il faut non seulement procurer le compilateur `meaning` en état de marche à l'exécution mais aussi capturer l'environnement lexical courant afin de pouvoir reprendre la compilation dans cet environnement même. C'est pourquoi la suspension engendrée enclôt non seulement `meaning` mais aussi `r` (et accessoirement `tail?`), qui toutes doivent être présentes à l'exécution.

L'effet sera encore plus pertinent si l'on montre ce que devient le compilateur d'octets discuté au chapitre 7. **[☞ p. 213]** La fonction d'analyse `meaning-eval` appelle le générateur d'octets `EVAL/CE` (pour *eval in current environment*). Ce générateur reçoit l'argument à évaluer ainsi que l'environnement lexical de compilation.

```
(define (meaning-eval e r tail?)
  (let ((m (meaning e r #f)))
    (EVAL/CE m r) ) )
(define (EVAL/CE m r)
  (append (PRESERVE-ENV) (CONSTANT r) (PUSH-VALUE)
          m (COMPILE-RUN) (RESTORE-ENV) ) )
```

Le générateur d'octets `EVAL/CE` préserve systématiquement l'environnement d'exécution parce que l'on ne tient pas compte du paramètre `tail?`. Une nouvelle instruction `(COMPILE-RUN)` condense en elle tout le compilateur-évaluateur. Cette instruction prend l'expression à évaluer dans le registre `*val*` et l'environnement de compilation en sommet de pile, délègue à `compile-on-the-fly` le soin de compiler le programme et d'installer son code en mémoire pour l'exécuter comme si c'était le corps d'une fonction.

```
(define (COMPILE-RUN) (list 255))
(define-instruction (COMPILE-RUN) 255
  (let ((v *val*)
        (r (stack-pop)) )
    (if (program? v)
        (compile-and-run v r #f)
        (signal-exception #t (list "Illegal program" v)) ) ) )
(define (compile-and-run v r tail?)
  (unless tail? (stack-push *pc*))
  (set! *pc* (compile-on-the-fly v r)) )
(define (compile-on-the-fly v r)
  (set! g.current '())
  (for-each g.current-extend! sg.current.names)
  (set! *quotations* (vector->list *constants*))
  (set! *dynamic-variables* *dynamic-variables*)
  (let ((code (apply vector (append (meaning v r #f) (RETURN)))))
    (set! sg.current.names (map car (reverse g.current)))
    (let ((v (make-vector (length sg.current.names)
                          undefined-value )))
      (vector-copy! sg.current v 0 (vector-length sg.current))
      (set! sg.current v) )
    (set! *constants* (apply vector *quotations*))
    (set! *dynamic-variables* *dynamic-variables*)
    (install-code! code) ) )
```

La compilation à la volée nécessite de remettre à jour les variables globales de compilation : `g.current`, l'environnement des variables globales modifiables, `*quotations*`, les citations et `*dynamic-variables*`, les variables dynamiques [4]. Le code compilé est suivi d'un `(RETURN)` afin de pouvoir retourner vers son appelant. Après la compilation, on enrichit en retour l'environnement d'exécution afin d'y adjoindre les nouvelles citations, variables globales modifiables ou dynamiques créées à cette occasion. Il ne reste plus qu'à installer le nouveau segment de code puis à l'exécuter.

On remarquera que cette installation est analogue à une édition de liens dynamique, ce qui montre qu'`eval`, pour un langage comme C, revient à écrire une chaîne de caractères dans un fichier, à compiler ce fichier par appel à `cc` puis à charger ce fichier par édition de liens dynamique et finalement à l'exécuter. Le code ainsi installé ne peut être éliminé (dans l'état actuel de la machine) et donc, à terme, obstruera la mémoire. La récupération de l'espace mémoire consommé par du code compilé est toujours délicate.

Le coût d'introduction de cette nouvelle forme spéciale est donc celui d'une nouvelle instruction `(COMPILE-RUN)` qui s'accompagne de la présence de la fonction `mea-`

4. Au contraire des autres variables, `*dynamic-variables*` est partagée entre le compilateur et la machine d'exécution, aussi les affectations idempotentes qui portent sur elle sont-elles inutiles mais elles indiquent ce qu'il faudrait changer si elles n'étaient point partagées.

`ning` et d'utilitaires (soit un énorme accroissement de la taille des petites applications) ainsi que de citations supplémentaires pour mémoriser l'environnement de compilation nécessaire à chaque occurence de `eval`. Il est heureux que l'on n'ait besoin de sauvegarder ceux-ci qu'en ces seuls endroits. Le coût marginal est donc limité au nombre de formes `eval`.

En matière de mise au point, insérer une forme `eval` permet de programmer une boucle d'interaction locale autorisant de consulter ou modifier les variables locales sous leur propre nom. La disponibilité d'un débogueur, activable à quelque stade de programme que l'on soit par simple interruption (lancée depuis le clavier, par exemple), requerrait le texte entier du programme ainsi que la préservation de tous les environnements de compilation.

8.2.1 Création de variables globales

En vertu du contrat explicité en (2), l'expression `(eval '(set! foo 33))`, doit être équivalente à `(set! foo 33)` dont on a **[☞ p. 105]** vu différentes sémantiques au chapitre 4 qui sont analysées plus avant au paragraphe 6.1.9. **[☞ p. 194]** Si la simple mention d'un nom de variable suffit pour qu'elle existe, `eval` doit alors faire de même. S'il faut par contre déclarer les variables avant usage, `eval` doit alors s'y conformer. On observera que la fonction `compile-on-the-fly` étendait, en retour d'évaluation, l'environnement global afin de prendre en compte les variables globales modifiables nouvellement apparues.

Le but de cette remarque est de faire prendre conscience que les équations (1) ou (2) stipulent que non seulement les valeurs de π et `(eval '`π`)`, mais aussi leurs effets induits (variables globales créées, modifications de l'environnement global ou du contexte d'évaluation), doivent être les mêmes. Cette notion de « même » est prise en compte par l'équation (2) qui doit rester valable quel que soit le contexte, ce qui signifie que π et `(eval '`π`)` doivent être indiscernables : il n'existe donc pas de programme que l'on puisse écrire et qui permette de les distinguer[5].

8.3 `eval` comme fonction

La forme spéciale `eval` évalue son argument tout comme le ferait une fonction, on peut donc légitimement se demander si l'on ne pourrait pas procurer ce mécanisme d'évaluation sous forme d'une fonction plutôt que par une forme spéciale. Reprenons donc les différents interprètes et montrons comment introduire cette fonction.

Avec l'interprète naïf du début de ce livre, on écrira : **[☞ p. 3]**

```
(defprimitive eval
  (lambda (v)
    (if (program? v)
        (evaluate v env.global)
        (wrong "Illegal program" v) ))
  1 )
```

5. C'est en fait assez théorique car si l'on permet que soit consultée l'heure, les programmes peuvent mesurer leur temps d'exécution et en déduire peut-être s'il y a eu `eval` ou non.

Tout de suite apparaît la différence majeure qui est que l'on a perdu l'environnement lexical courant. Comme il faut cependant en donner un à l'évaluateur `evaluate`, on fournira le seul qui soit visible et sensé : l'environnement global ! On a donc ici une fonction jouant le rôle d'un évaluateur global équivalent à celui qu'utilise la boucle fondamentale d'interaction. On la nommera `eval/at` pour *eval at top-level* tandis que l'on nommera `eval/ce`, pour la distinguer quand besoin s'en fera sentir, la précédente forme spéciale. Employer `eval/at` n'est pas complètement perdant car il y a un gain qui est que l'on n'a plus besoin de mémoriser les environnements lexicaux présents aux appels à `eval/ce` : on économise donc quelques citations.

Pour fixer les idées et reprendre de précédents exemples, qu'`eval/at` soit une fonction conduit à :

```
(set! x 2)(set! z 1)
((lambda (x) (eval/at 'x))
 3 )                                  → 2
((lambda (x y) (eval/at y))
 4 'x )                               → 2
((lambda (x y z) (eval/at y))
 5 (list 'eval/at 'z) 'x )            → 1
```

Si `eval/at` est une régression par rapport à `eval/ce`, on peut (presque) simuler cette première avec cette dernière en posant :

```
(define (eval/at x) (eval/ce x))
```

Malheureusement il y a une variable de trop dans l'environnement capturé : la variable locale nommée `x` qui masque la variable globale de même nom. **[☞ Ex. 8.3]** Ce problème laisse penser qu'une manipulation plus fine de l'environnement pourrait constituer une solution. Cette définition nous suggère cependant ce que peut être la définition d'`eval/at` dans l'interprète compilateur d'octets. La définition emploie encore la fonction `compile-and-run` mais sans lui demander d'empiler l'adresse de retour car l'appelant de `eval/at` y a déjà veillé. La compilation est toutefois ici faite avec `r.init` puisque l'on n'a plus accès à l'environnement lexical courant.

```
(definitial eval
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (= arity+1 (activation-frame-argument-length *val*))
           (let ((v (activation-frame-argument *val* 0)))
             (if (program? v)
                 (compile-and-run v r.init #t)
                 (signal-exception #t (list "Illegal program" v)) ) )
           (signal-exception
            #t (list "Incorrect arity" 'eval) ) ) ) ) ) )
```

Évaluer comme si l'on était au niveau fondamental d'interaction est à prendre au pied de la lettre en ce qui concerne l'environnement lexical mais n'a aucune raison d'être, implantatoirement parlant, si l'on considère par exemple l'environnement dynamique. Par exemple, que `eval` soit une forme spéciale ou une fonction, on doit observer :

```
(dynamic-let (a 22)
  (eval '(dynamic a)) )    → 22
```

En ce qui concerne le contrat initial d'`eval` à savoir l'équation (1), elle demeure

à condition de ne la considérer que dans le contexte vide, c'est-à-dire justement au niveau fondamental. Par contre, `eval/at` ne satisfait pas à (2) comme l'ont prouvé les exemples plus haut. Pour mieux préciser le comportement d'`eval/at`, on recourera à (3) où v est une variable non capturable :

$$C[\texttt{(eval/at '}\pi\texttt{)}] \equiv \texttt{(let ((}v\texttt{ (lambda () }\pi\texttt{)))}\ C[\texttt{(}v\texttt{)}]) \qquad (3)$$

La fonction `eval/at` est la variante la plus commune des `eval` que l'on peut rencontrer dans les systèmes Lisp.

8.4 Le coût d'`eval`

Le coût lié à l'utilisation d'`eval` est difficile à apprécier *in toto*. Supposons tout d'abord que l'application autonome à bâtir n'est autre que le fameux `(display "Hello world")`. Si la forme spéciale `eval/ce` n'apparaît pas dans l'application (et c'est une simple vérification statique), nul coût ! Si elle apparaît, il faut alors au moins ajouter le compilateur à l'application et sa taille change d'ordre de grandeur : disons, grossièrement, de 50 à 500 Koctets. Si l'évaluation dynamique est procurée sous forme de fonction et s'il n'est pas prouvé que la fonction ne sera pas inutile, on se retrouve alors dans le même cas. Pour prouver que `eval/at` est inutile, il faut vérifier que nul n'en use. Le langage peut aider pour cette preuve : en Scheme par exemple, il suffit de prouver que la variable globale `eval/at` n'est pas mentionnée. En COMMON LISP, ce n'est en général pas possible puisqu'il faut prouver de plus (et entre autres) que personne ne peut engendrer la chaîne de caractères `"eval/at"`, la convertir en un symbole (par `find-symbol`), et extraire du symbole (par `symbol-function`) la fonction éponyme. De fait la seule mention de `symbol-function` avec un argument non prévisible coûte.

Si d'aventure un appel à `eval` apparaît, peut-on limiter l'ensemble des valeurs sur lesquelles `eval` est appliquée ? Dans la majeure partie des cas, cette analyse n'est guère possible et l'on peut donc légitimement penser que toute valeur est possible. On ne peut alors plus faire l'économie d'une seule fonction de l'environnement global puisque tout, *tout* est possiblement appelable ! Engendrer une application doit alors contenir tout ce que le langage définit sans exception et la taille de l'exécutable, dans des langages riches comme COMMON LISP, enfle encore d'un ordre de grandeur.

Nous ne sommes pas arrivés au bout de nos peines. Même si l'on est dans un cas idéal avec un système de modules qui limite la visibilité et la modifiabilité de l'environnement global [QP91b], la présence d'`eval` tue de nombreuses possibilités de compilation améliorée. Examinez la définition suivante :

```
(define (fib n)
  (if (< n 2) (eval (read))
      (+ (fib (- n 1)) (fib (- n 2))) ) )
```

Indépendamment des effets que peut avoir `eval` sur les autres variables globales, `eval` peut aussi modifier la variable globale `fib`, ce qui oblige, lors du second appel récursif, à chercher la valeur de `fib` via l'adresse associée à la variable globale et non à utiliser un aveugle `GOTO`. Il ne saurait donc y avoir d'amélioration liée à l'invariabilité de variables globales puisque `eval` peut tout modifier. En conclusion la présence d'un seul `eval` dans un module rend pratiquement tout modifiable dans ce module.

Pour toutes ces raisons, `eval` est considérée comme une caractéristique coûteuse.

Toutefois, au sein d'une grosse application (de quelques Moctets) et en phase de développement où tout est en état plus ou moins instable, l'adjonction d'`eval` est infime et bien commode. Il est vrai que si l'on a une calculette à programmer, il est plus simple, en durée de programmation (et c'est le programmeur qui coûte cher par les temps qui courent), d'appeler `eval` pour calculer des expressions arithmétiques plutôt que d'implanter un interprète d'un petit langage ad hoc. Lisp est un remarquable langage d'extension et disposer d'`eval` en bibliothèque est un atout important.

8.5 `eval` interprété

Ce livre fait un usage immodéré d'interprètes, aussi peut-on se demander si, à défaut d'`eval` dans un système, l'utilisateur ne pourrait pas y suppléer avec ses seuls moyens ? Après tout, étant donné le nombre d'interprètes déjà vus, il n'a que l'embarras du choix pour en élire un, récupérer son code par `ftp` anonyme (ou le taper lui-même) et l'incorporer à son application. La réponse est à la fois oui et non.

Si l'on désire écrire, en Scheme pur, un évaluateur, ce ne peut être que sous la forme d'une fonction puisqu'il n'est pas possible de définir ses propres formes spéciales. Deux problèmes se posent.

8.5.1 Interchangeabilité des représentations

Le premier est d'organiser les interactions entre le système sous-jacent et l'interprète que l'on souhaite écrire. Ceci impose de ne point imposer de nouveaux types de données et de se conformer aux types préxistant. Les doublets que gèrent l'interprète doivent être les doublets de l'implantation ; les booléens doivent être les mêmes, les fonctions doivent respecter des protocoles d'appel semblables. L'évaluation explicite, figurant dans l'exemple qui suit, doit obligatoirement retourner une fonction invoquable par le système sous-jacent.

```
((eval '(lambda (x) (cons x x)))
 33 )     → (33 . 33)
```

Inversement, l'interprète doit aussi savoir invoquer les fonctions du système sous-jacent :

```
(((eval '(lambda (f)
           (lambda (x) (f x)) ))
  list )
 44 )     → (44)
```

Une représentation commune possible pour les fonctions est d'utiliser les fonctions d'arité multiples `(lambda values ...)` de manière à s'adapter aux deux modes d'invocation possibles. Les modifications à apporter, par exemple, au premier des **[☞ p. 3]** interprètes de ce livre, l'interprète naïf du premier chapitre, sont simples. Et le reste s'en déduit aisément.

```
(define (invoke fn args)
  (if (procedure? fn)
      (apply fn args)
      (wrong "Not a function" fn) ) )
(define (make-function variables body env)
  (lambda values
```

```
    (eprogn body (extend env variables values)) ) )
```

Un point délicat est de faire en sorte que le rattrapage des erreurs (par exemple, les erreurs d'arités) soit le même dans cet `eval` et dans le système sous-jacent mais nous ne nous y attarderons pas.

8.5.2 Environnement global

Le second problème concerne la manipulation de l'environnement global. Les évaluateurs de ce livre avaient tous leur propre définition de leur environnement global qu'ils bâtissaient à l'aide des macros `definitial`, `defprimitive`... Ici, il s'agit de coopérer avec le système sous-jacent et donc de faire en sorte que l'environnement global vu depuis l'interprète ou le système soit le même. Encore en d'autres termes, l'expression qui suit doit être, sans erreur, évaluable :

```
(begin (set! foo 128)
       (eval '(set! bar (+ foo foo)))
       bar )   → 256
```

Compilateur d'applications autonomes

Si l'on est dans un compilateur de fichiers comme Scheme->C [Bar89] ou Bigloo [Ser94], le programme est connu et ne saurait (par construction) accéder aux variables qu'il ne mentionne pas. Les variables qu'`eval` crée ne peuvent donc être manipulées que par `eval`. Il suffit donc de connecter l'environnement global sous-jacent à l'interprète et, pour cela, définir deux fonctions que nous nommerons `global-value` et `set-global-value!`. Leur corps est régulièrement formé d'après le nom de toutes les variables globales du programme (et elles sont statiquement connues). On peut même imaginer que ces fonctions soient automatiquement synthétisées[6] : **[☞ Ex. 7.3]**

```
(define (global-value name)
  (case name
    ((car) car)
    ...
    ((foo) foo)
    (else (wrong "No such global variable" name)) ) )
(define (set-global-value! name value)
  (case name
    ((foo) (set! foo value))
    ...
    (else (wrong "No such global mutable variable" name)) ) )
```

L'interprète peut créer toutes les variables globales qu'il souhaite ; aucune n'est atteignable directement par le système sous-jacent et l'interprète est le seul à pouvoir les manipuler.

Système interactif

Si par contre on est dans un système doté d'une boucle d'interaction, alors à la fois le système et l'interprète peuvent créer de nouvelles variables. L'exemple donné plus

6. Dans la fonction `set-global-value!`, la variable `car` n'apparaît pas afin de préserver son immuabilité.

haut peut être éclaté en une séquence d'interactions où l'on voit que l'on peut demander au système sous-jacent la valeur d'une variable créée par l'interprète explicite :

```
? (begin (set! foo 128)
         (eval '(set! bar (+ foo foo)))
        2001 )
= 2001
? bar
= 256
```

Plusieurs solutions existent pour résoudre ce problème de coopération.

Passage par symboles

La plus traditionnelle de ces méthodes passe par l'emploi de symboles, ce qui a miné pendant de longues années les bases sémantiques de Lisp par l'entretien de confusions regrettables entre symboles et variables, attisées par le fait que les variables sont représentées par des symboles et que, comme on va le voir, les symboles peuvent aussi servir à implanter des variables.

Les symboles sont des structures de données nécessitant un unique champ permettant d'associer à un symbole son nom : une chaîne de caractères. Les symboles sont créés explicitement en Scheme par la fonction `string->symbol` et sont tels que deux symboles de même nom ne peuvent simultanément exister et être différents. On assure souvent cela par une table de hachage associant aux chaînes de caractères leur symbole attitré. La fonction `string->symbol` commence d'abord par regarder si un symbole de nom approprié existe déjà et le retourne en cas de succès et sinon en fabrique un. Pour accélérer la recherche des symboles à partir de leur nom, des champs supplémentaires (cachés) chaînant les symboles entre eux sont souvent ajoutés.

On ajoute fréquemment aux symboles une liste de propriétés, généralement gérée comme une Pliste, encore que l'on rencontre aussi des A-listes ou des tables de hachage. L'encombrement et les vitesses d'accès aux propriétés dépendent de l'implantation choisie. Certains systèmes Lisp, comme Le-Lisp, ont encore ajouté des champs aux symboles, sortes d'antémémoire pour des propriétés assez répandues comme, par exemple, la façon d'imprimer bellement.

La structure de symbole n'étant plus à quelques champs près, pourquoi ne pas y ajouter un champ pour y stocker la valeur globale de la variable éponyme ? Et aussi, dans le cas d'un Lisp_2, la valeur de la fonction globale éponyme. C'est ce qui a été fait depuis des temps immémoriaux sous les noms de Cval et Fval [Cha80, GP88]. Des fonctions existent pour lire et écrire ces champs : en COMMON LISP, `symbol-value` et `(setf symbol-value)` ainsi que `symbol-function` et `(setf symbol-function)`.

La technique est particulièrement attractive car excessivement sûre. Tout symbole susceptible de servir de représentation pour une variable globale a nécessairement été fabriqué auparavant par `symbol->string` (en général à la lecture par `read`) et donc une adresse pour contenir sa valeur globale existe déjà sous la forme d'un champ du symbole. Certes le schéma est dispendieux puisque tout symbole n'est pas nécessairement le support de la représentation d'une variable globale de même nom ; en revanche il ne peut exister de variable globale non associée à une adresse apte à contenir sa valeur globale.

Il suffit donc de fournir deux fonctions, `global-value` et `set-global-value!`, capables d'obtenir et d'affecter la valeur de variables globales à partir de leurs noms. L'interprète explicite manipulera l'environnement global par ces fonctions et le problème de l'extension de cet environnement global ne se posera jamais puisqu'il est résolu par un mécanisme sous-jacent, magique, lié au concept de symbole.

Indiquons comment enrichir l'interprète du premier chapitre pour tenir compte de ces fonctions magiques. L'environnement, valeur de la variable `env`, ne va plus contenir que les seules variables locales à l'exclusion des variables globales qui seront gérées directement. Comme on a en plus adopté une représentation commune des fonctions entre interprète et système, on peut entièrement laisser tomber la définition de l'environnement global de l'interprète puisque ce dernier peut utiliser directement l'environnement global du système. On aura ainsi :

```
(define (lookup id env)
  (if (pair? env)
      (if (eq? (caar env) id)
          (cdar env)
          (lookup id (cdr env)) )
      (global-value id) ) )
(define (update! id env value)
  (if (pair? env)
      (if (eq? (caar env) id)
          (begin (set-cdr! (car env) value)
                 value )
          (update! id (cdr env) value) )
      (set-global-value! id value) ) )
```

Cette solution semble élégante mais a un coût impressionant puisque toute variable globale est accessible par son nom. Une application autonome contenant un appel à `global-value` ne peut éliminer quoique ce soit de la bibliothèque de fonctions initiales puisque a priori tout nom est calculable et donc tout est nécessaire. Ce n'est pas gênant dans un environnement de programmation puisque c'est une de ses propriétés que de tout mettre à la disposition des programmeurs, c'est en revanche un problème pour une petite application autonome. Pire, la fonction `set-global-value!` peut changer la valeur de n'importe quelle variable globale et donc briser toute optimisation de compilation puisque plus rien n'est a priori immuable.

Les deux fonctions `global-value` et `set-global-value!` peuvent être considérées comme une version spécialisée de `eval` ; on la trouvait d'ailleurs sous le nom de `symeval` dans les anciens systèmes Lisp. On peut aussi les voir comme des fonctions réflexives réifiant l'accès à l'environnement global. Écrire « `foo` » pour connaître la valeur de la variable globale `foo`, c'est écrire (et faire implicitement évaluer) `(global-value 'foo)`.

Environnement de première classe

Une autre technique serait de définir une fonction `eval` à deux variables : l'expression à calculer et l'environnement dans lequel la calculer. Cette signature apparente cet `eval` binaire à l'évaluateur naïf du premier chapitre mais ce qui fait défaut pour adopter cette solution est qu'il doit être possible de fournir des valeurs comme second argument et des environnements, alors qu'aucune opération n'est disponible pour en

obtenir. Il faut donc autoriser la réification des environnements ainsi qu'éventuellement d'autres opérations comme l'extraction/modification de valeur de variables. On pourra grâce à ces opérateurs connecter les environnements manipulés par l'évaluateur explicite à ceux de l'implantation. Cette technique soulève de nombreux problèmes que nous allons maintenant aborder.

8.6 Réification d'environnements

L'interprète dénotationnel avait bien montré que l'évaluation en Scheme reposait sur un triplet (environnement, continuation, mémoire). Rejetant encore une fois la mémoire, on voit que les continuations sont manipulables via `call/cc` (ou `bind-exit`) qui les réifie. Ce n'était pas le cas des environnements lexicaux qui ne pouvaient être directement atteints. Les réifier en des structures de données manipulables est l'objet de cette section.

Ce qui caractérise et différencie les implantations d'environnements de première classe [RA82, MR91], ce sont les propriétés qu'ils possèdent et les opérations qu'ils proposent. Les trois opérations fondamentales sont : rechercher la valeur d'une variable, modifier la valeur d'une variable ou étendre un environnement. Réifier un environnement, c'est donner un moyen de capturer des liaisons mais sans l'a priori des abstractions qui ne les capturent que de façon opaque et non explicitement manipulable.

8.6.1 La forme spéciale `export`

Nous allons introduire une nouvelle forme spéciale, que l'on nommera `export`, à qui l'on mentionnera les noms des variables à capturer et qui retournera un environnement enclosant les liaisons des variables spécifiées. L'environnement réifié pourra être utilisé comme second argument de la fonction d'évaluation binaire que l'on distinguera des précédentes en la nommant cette fois-ci `eval/b`. Donnons quelques exemples :

```
(let ((r (let ((x 3) (export x)))))
  (eval/b 'x r) )                          → 3
(let ((f+r (let ((x 3))
             (cons (lambda () x) (export x)) )))
  (eval/b '(set! x (+ 1 x)) (cdr f+r))
  ((car f+r)) )                            → 4
(let ((r (export car)))
  (let ((car cdr))
    (eval/b '(car (cons 1 2)) r) ) )   → 1
```

Dans le premier exemple, l'environnement de première classe créé capture la variable `x` qui peut donc être utilisée à loisir par `eval/b`. Le second exemple montre que l'on peut aussi modifier la variable `x` et que c'est bien la liaison qui a été capturée puisque cette modification est bien perçue de qui l'enclôt par les voies naturelles de l'abstraction. Le troisième exemple montre que l'on peut aussi capturer des liaisons globales.

La forme spéciale `export` permet de jongler avec les environnements, d'en capturer des bouts et des les utiliser ailleurs. Son implantation n'est pas triviale, nous allons

donc la décrire. Comme toutes les formes spéciales, on ajoutera une clause d'analyse syntaxique à la fonction `meaning` :

```
... ((export) (meaning-export (cdr e) r tail?)) ...
```

Un environnement réifié doit non seulement capturer la liste chaînée des blocs d'activation mais doit aussi conserver les noms et les adresses des variables dont on a demandé la capture. Cela rappelle l'implantation de `eval/ce`, qui requérait que soit conservée la même information. On représentera un environnement réifié par un objet à deux champs : la liste des blocs d'activation et une liste de couples nom-adresse donnant la « carte » permettant de déchiffrer les blocs d'activation (Figure 8.2).

```
(define-class reified-environment Object
  ( sr r ) )
```

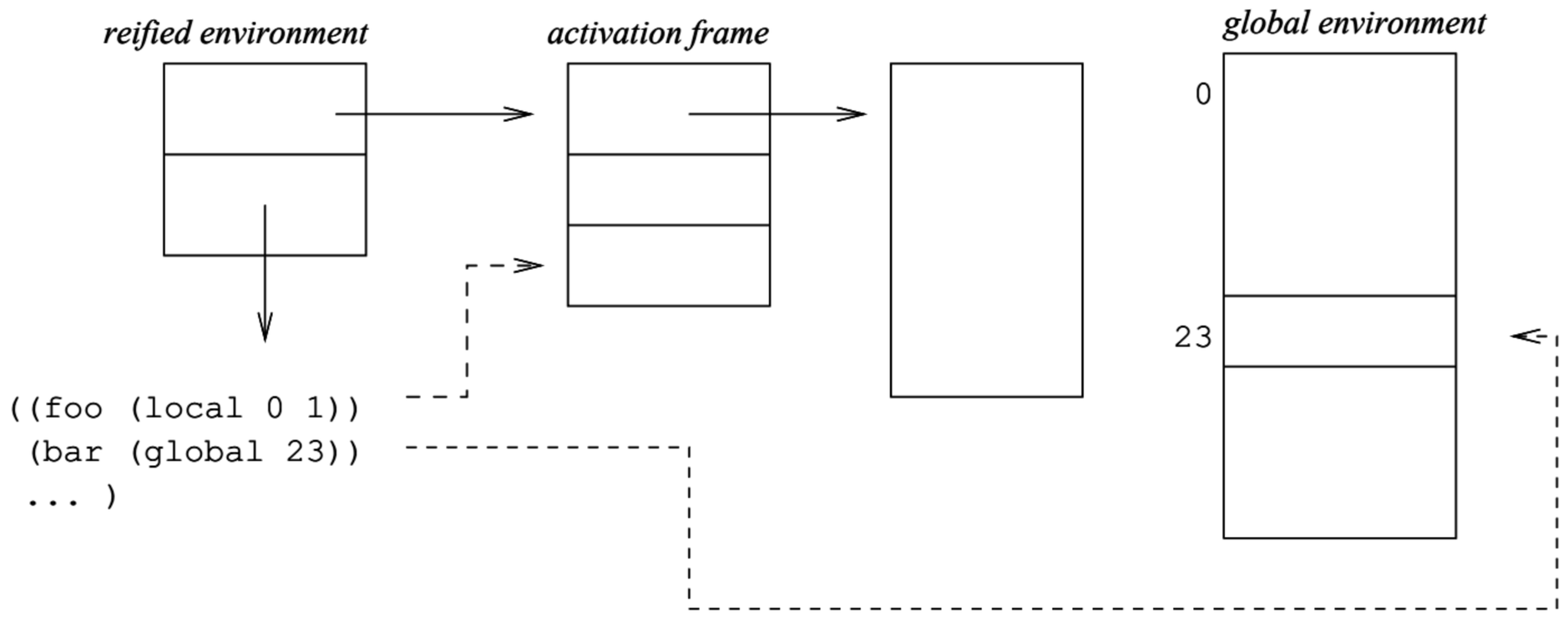


FIGURE 8.2 – Environnement réifié

Sans entrer de suite dans les détails, on compilera ainsi une forme `export` :

```
(define (meaning-export n* r tail?)
  (unless (every? symbol? n*)
          (static-wrong "Incorrect variables" n*) )
  (append (CONSTANT (extract-addresses n* r))
          (CREATE-1ST-CLASS-ENV) ) )
(define (CREATE-1ST-CLASS-ENV) (list 254))
(define-instruction (CREATE-1ST-CLASS-ENV) 254
  (create-first-class-environment *val* *env*) )
(define (create-first-class-environment r sr)
  (set! *val* (make-reified-environment sr r)) )
```

La liste des couples nom-adresses sera représentée par une citation dont se saisira (dans le registre `*val*`) la nouvelle instruction `CREATE-1ST-CLASS-ENV` pour fabriquer l'objet de nos vœux.

Pour simplifier l'évaluation, nous allons changer la représentation des environnements statiques de compilation (les valeurs des variables `r`). Au lieu de la représentation en côtelettes, nous allons adopter une représentation plus explicite sous la forme d'une liste d'association nom-adresse comme celle qui apparaît à la figure 8.2. Il n'y a que peu de choses à changer mais en simplifiant `compute-kind`, on complique `r-extend*` qui doit, chaque fois que l'on étend l'environnement, enterrer un peu plus les variables profondes.

```
(define (compute-kind r n)
```

```
  (or (let ((var (assq n r)))
        (and (pair? var) (cadr var)) )
      (global-variable? g.current n)
      (global-variable? g.init n)
      (adjoin-global-variable! n) ) )
(define r.init '())
(define (r-extend* r n*)
  (let ((old-r (bury-r r 1)))
    (let scan ((n* n*)(i 0))
      (cond ((pair? n*) (cons (list (car n*) `(local 0 . ,i))
                              (scan (cdr n*) (+ i 1)) ))
            ((null? n*) old-r)
            (else (cons (list n* `(local 0 . ,i)) old-r)) ) ) ) )
(define (bury-r r offset)
  (map (lambda (d)
         (let ((name (car d))
               (type (car (cadr d))) )
           (case type
             ((local checked-local)
              (let* ((addr (cadr d))
                     (i (cadr addr))
                     (j (cddr addr)) )
                `(,name (,type ,(+ i offset) . ,j) . ,(cddr d)) ) )
             (else d) ) ) )
       r ) )
```

Plutôt que de réifier un environnement avec les seules variables mentionnées, on fera en sorte que `(export)` soit équivalente à la forme `(export` *variables*`...)`, où l'on aurait spécifié toutes les variables des abstractions ambiantes. Avec la forme `(export)` (souvent appelée aussi `(the-environment)`), `eval/b` peut entièrement simuler la forme spéciale `eval/ce`, puisque :

`(eval/ce '`π`)` $\equiv$ `(eval/b '`π `(export))`

Pour réaliser cette amélioration, il nous suffira de capturer tout l'environnement disponible (qui maintenant a la bonne structure) et écrire :

```
(define (extract-addresses n* r)
  (if (null? n*) r
      (let scan ((n* n*))
        (if (pair? n*)
            (cons (list (car n*) (compute-kind r (car n*)))
                  (scan (cdr n*)) )
            '() ) ) ) )
```

8.6.2 La fonction `eval/b`

Quelques réminiscences d'`eval/ce` perdurent au sein d'`eval/b`; elles sont semblables sauf en ce qui concerne l'acquisition des paramètres d'évaluation et le protocole de retour d'évaluation. On comparera utilement ce qui suit à ce qui fut. La fonction `eval/b` vérifie la nature de ses arguments, puis délègue à la fonction `compile-on-the-fly` précédemment vue le soin de compiler à la volée l'expression dans l'environnement fourni et d'installer puis d'exécuter le code produit quelque part en mémoire. L'environnement courant n'a pas besoin d'être sauvegardé puisque la sauvegarde est déjà assurée par le protocole d'appel fonctionnel.

```
(definitial eval/b
  (let* ((arity 2)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (= arity+1 (activation-frame-argument-length *val*))
           (let ((exp (activation-frame-argument *val* 0))
                 (env (activation-frame-argument *val* 1)) )
             (if (program? exp)
                 (if (reified-environment? env)
                     (compile-and-evaluate exp env)
                     (signal-exception
                      #t (list "Not an environment" env) ) )
                 (signal-exception
                  #t (list "Illegal program" exp) ) ) )
           (signal-exception
            #t (list "Incorrect arity" 'eval/b) ) ) ) ) ) )
(define (compile-and-evaluate v env)
  (let ((r (reified-environment-r env))
        (sr (reified-environment-sr env)) )
    (set! *env* sr)
    (set! *pc* (compile-on-the-fly v r)) ) )
```

8.6.3 Enrichissement d'environnements

Les environnements sont souvent étendus ; il paraît donc logique de procurer cette opération. On fournira donc une fonction `enrich` prenant un environnement et une suite de noms de variables et ajoutant celles-ci à celui-là. La valeur retournée sera un *nouvel* environnement enrichi ; `enrich` est un modificateur fonctionnel qui ne perturbe pas ses arguments. Donnons un exemple d'emploi : simulons un `letrec` par enrichissement explicite d'environnement. Une liaison globale est capturée[7] ; deux variables locales l'enrichissent (`odd?` et `even?`) ; deux définitions mutuellement récursives y sont évaluées ; un calcul enfin y est pratiqué.

```
((lambda (e)
   (set! e (enrich (export *) 'even? 'odd?))
   (eval/b '(set! even? (lambda (n)
                          (if (= n 0) #t (odd? (- n 1))) )) e)
   (eval/b '(set! odd? (lambda (n)
                         (if (= n 0) #f (even? (- n 1))) )) e)
   (eval/b '(even? 4) e) )
 'ee )                                  → #t
```

La figure 8.3 détaille le résultat de la construction. Un nouveau bloc d'activation est alloué pour contenir les nouvelles variables ; l'environnement réifié associe à ces nouveaux noms les adresses idoines. La seule difficulté est que ces nouvelles variables, quoiqu'existantes, n'ont pas de valeur. On retrouve là les discussions propres au `letrec` ou aux variables non initialisées. **[☞ p. 56]**

On introduira donc un nouveau type d'adresse locale `checked-local` qui est un peu l'analogue de `checked-global`. Notre définition de `enrich` nous apporte la possibilité de variables locales non initialisées, ce qui n'était pas possible avec `lambda`. Un

7. On voit ici une erreur de conception : il n'y a pas moyen de créer un environnement vide avec `export` ; on capture donc n'importe quoi, ici la multiplication `*`, pour créer un petit environnement.

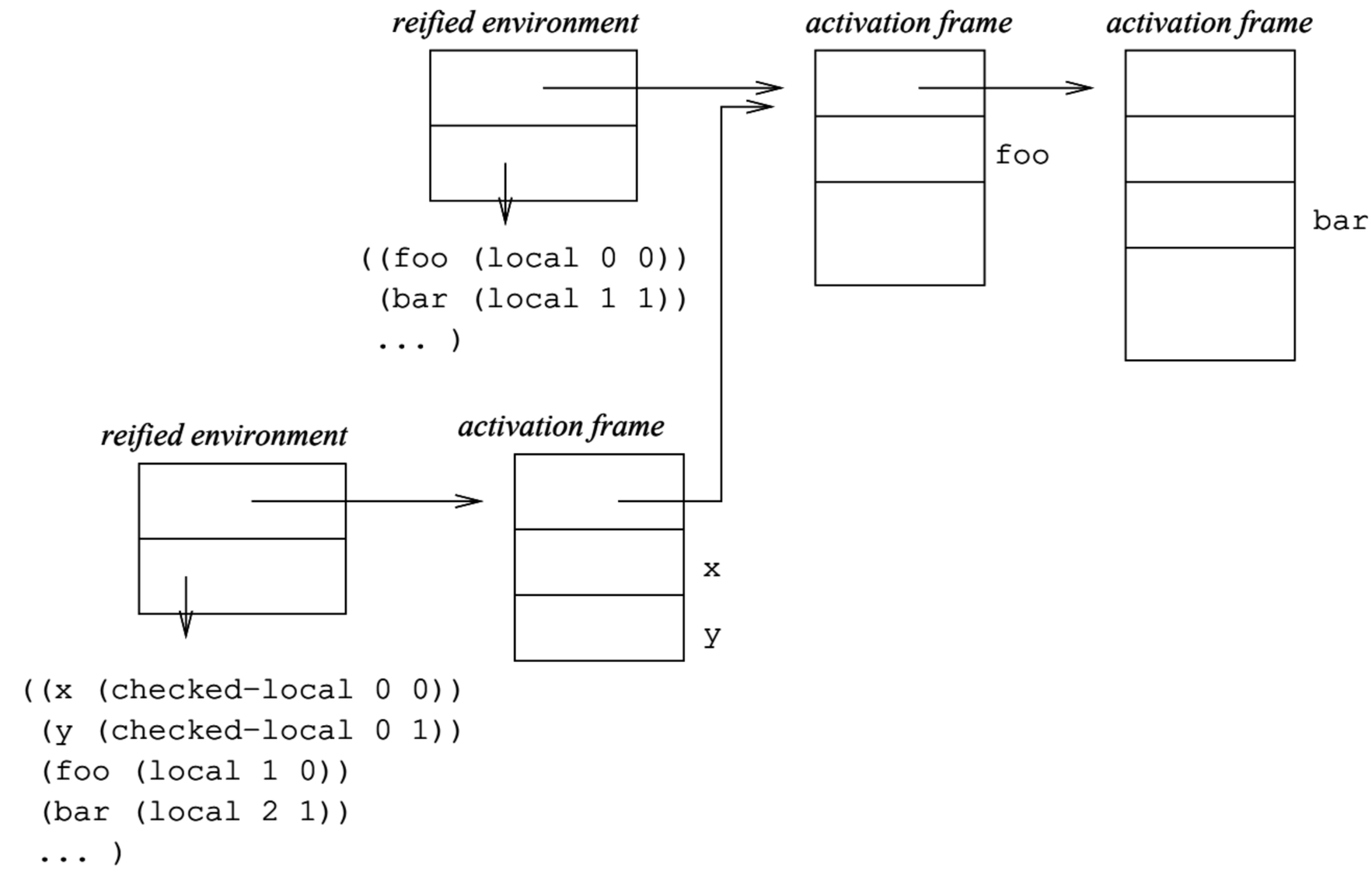


FIGURE 8.3 – Environnement enrichi `(enrich` *env* `'x 'y)`

palliatif serait de forcer l'enrichissement d'environnements par des variables accompagnées de leur valeur mais ce serait renoncer à l'écriture explicite de `letrec`.

La programmation de la fonction `enrich` est simple sinon courte.

```
(definitial enrich
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (>= (activation-frame-argument-length *val*) arity+1)
           (let ((env (activation-frame-argument *val* 0)))
             (listify! *val* 1)
             (if (reified-environment? env)
                 (let* ((names (activation-frame-argument *val* 1))
                        (len (- (activation-frame-argument-length
                                 *val* )
                                2 ))
                        (r (reified-environment-r env))
                        (sr (reified-environment-sr env))
                        (frame (allocate-activation-frame
                                (length names) )) )
                   (set-activation-frame-next! frame sr)
                   (do ((i (- len 1) (- i 1)))
                       ((< i 0))
                     (set-activation-frame-argument!
                      frame i undefined-value ) )
                   (unless (every? symbol? names)
                     (signal-exception
                      #f (list "Incorrect variable names" names ) ) )
                   (set! *val* (make-reified-environment
                                frame
                                (checked-r-extend* r names) ))
```

```
                    (set! *pc* (stack-pop)) )
                   (signal-exception
                    #t (list "Not an environment" env) ) ) )
              (signal-exception
               #t (list "Incorrect arity" 'enrich) ) ) ) ) ) )
(define (checked-r-extend* r n*)
  (let ((old-r (bury-r r 1)))
    (let scan ((n* n*)(i 0))
      (cond ((pair? n*) (cons (list (car n*) `(checked-local 0 . ,i))
                              (scan (cdr n*) (+ i 1)) ))
            ((null? n*) old-r) ) ) ) )
```

Toutefois, il nous faut mettre à jour les deux générateurs `meaning-reference` et `meaning-assignment` pour prendre en compte ce nouveau type d'adresse de type `checked-local`.

```
(define (meaning-reference n r tail?)
  (let ((kind (compute-kind r n)))
    (if kind
        (case (car kind)
          ((checked-local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (CHECKED-DEEP-REF i j) ) )
          ((local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (SHALLOW-ARGUMENT-REF j)
                 (DEEP-ARGUMENT-REF i j) ) ) )
          ((global)
           (let ((i (cdr kind)))
             (CHECKED-GLOBAL-REF i) ) )
          ((predefined)
           (let ((i (cdr kind)))
             (PREDEFINED i) ) ) )
        (static-wrong "No such variable" n) ) ) )
(define (meaning-assignment n e r tail?)
  (let ((m (meaning e r #f))
        (kind (compute-kind r n)) )
    (if kind
        (case (car kind)
          ((local checked-local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (SHALLOW-ARGUMENT-SET! j m)
                 (DEEP-ARGUMENT-SET! i j m) ) ) )
          ((global)
           (let ((i (cdr kind)))
             (GLOBAL-SET! i m) ) )
          ((predefined)
           (static-wrong "Immutable predefined variable" n) ) )
        (static-wrong "No such variable" n) ) ) )
```

Sans oublier d'ajouter l'instruction `CHECKED-DEEP-REF` à notre machine à octets :

```
(define-instruction (CHECKED-DEEP-REF i j) 253
  (set! *val* (deep-fetch *env* i j))
```

```
(when (eq? *val* undefined-value)
  (signal-exception #t (list "Uninitialized local variable")) ) )
```

8.6.4 Réification d'environnement clos

Un certain nombre d'interprètes procurent des primitives permettant d'extraire d'une fermeture l'environnement qu'elle a enclos. À côté de cette fonction, `procedure->environment`, les interprètes ajoutent souvent la fonction `procedure->definition` qui extrait d'une fermeture sa définition. Ces deux fonctions sont simples de réalisation dans les interprètes mais deviennent plus complexes et plus consommatrices (de mémoire) dans les compilateurs. Il faut en effet à la fois mémoriser les définitions, ce qui ajoute aux citations, mais aussi être capable de réifier les environnements clos, ce qui nécessite de conserver leur structure et d'ajouter de nouvelles citations. En outre, la fonction `procedure->environment` est très indiscrète puisqu'en réifiant un environnement clos, on peut le modifier, ce qui interdit toute optimisation sur des variables locales car plus rien de local n'est maintenant inatteignable. En ce sens, `procedure->environment` est bien plus coûteuse qu'`export` car elle met à nu l'évaluateur sans limitation sur ce qui peut y être effectué alors qu'`export` restreint ce qui peut être (dé)fait aux seules variables mentionnées.

La fonction `procedure->definition` est, quant à elle, intéressante pour les qualités d'introspection qu'elle procure. On peut ainsi fournir un metteur au point capable de désosser les fonctions à appliquer. Voici un exemple d'emploi de ces deux fonctions. La fonction `trace-procedure1` prend une fonction unaire, l'ausculte et rebâtit une nouvelle fonction unaire comparable à la première mais imprimant son argument en entrée et son résultat en sortie.

```
(define (trace-procedure1 f)
  (let* ((env (procedure->environment f))
         (definition (procedure->definition f))
         (variable (car (cadr definition)))
         (body (cddr definition)) )
    (lambda (value)
      (display (list 'entering f 'with value))
      (eval/b `(begin (set! ,variable ',value)
                      (let ((result (begin . ,body)))
                        (display (list 'result 'is result))
                        result ) )
              (enrich env variable) ) ) ) )
```

En fait, il faut l'avouer, le programme ci-dessus triche et péche par plusieurs points. Une fonction unaire comme `car` fera capoter `trace-procedure1`. Le programme synthétisé pour `eval/b` n'est pas toujours un programme légal car il cite la valeur de `value` qui n'est pas toujours une valeur légalement citable. Néanmoins la fonction `trace-procedure1` réalise un effet important pour la mise au point. Cet effet provient de la conjonction des fonctions d'introspection des fermetures, des environnements de première classe et de l'évaluation explicite.

Donner une implantation possible de ces fonctions permettra de juger de leur coût. La fonction `procedure->definition` est la plus simple, il suffit d'être capable d'associer à toute fermeture l'expression qui la définit, c'est-à-dire une citation. Le problème est : Où mettre cette citation, car il peut y avoir de multiples fermetures associées à

la même définition ? Ce qui caractérise vraiment les fermetures c'est l'adresse de leur code ; or toutes les fermetures de la même abstraction partagent cette adresse. Nous allons faire en sorte d'associer à cette adresse la citation appropriée. Pour cela nous allons utiliser une technique, bien connue des anciens programmeurs en langage d'assemblage, qui est d'insérer des données dans les instructions. Comme notre machine l'interdit, nous allons en fait insérer des citations comme illustré à la figure 8.4.

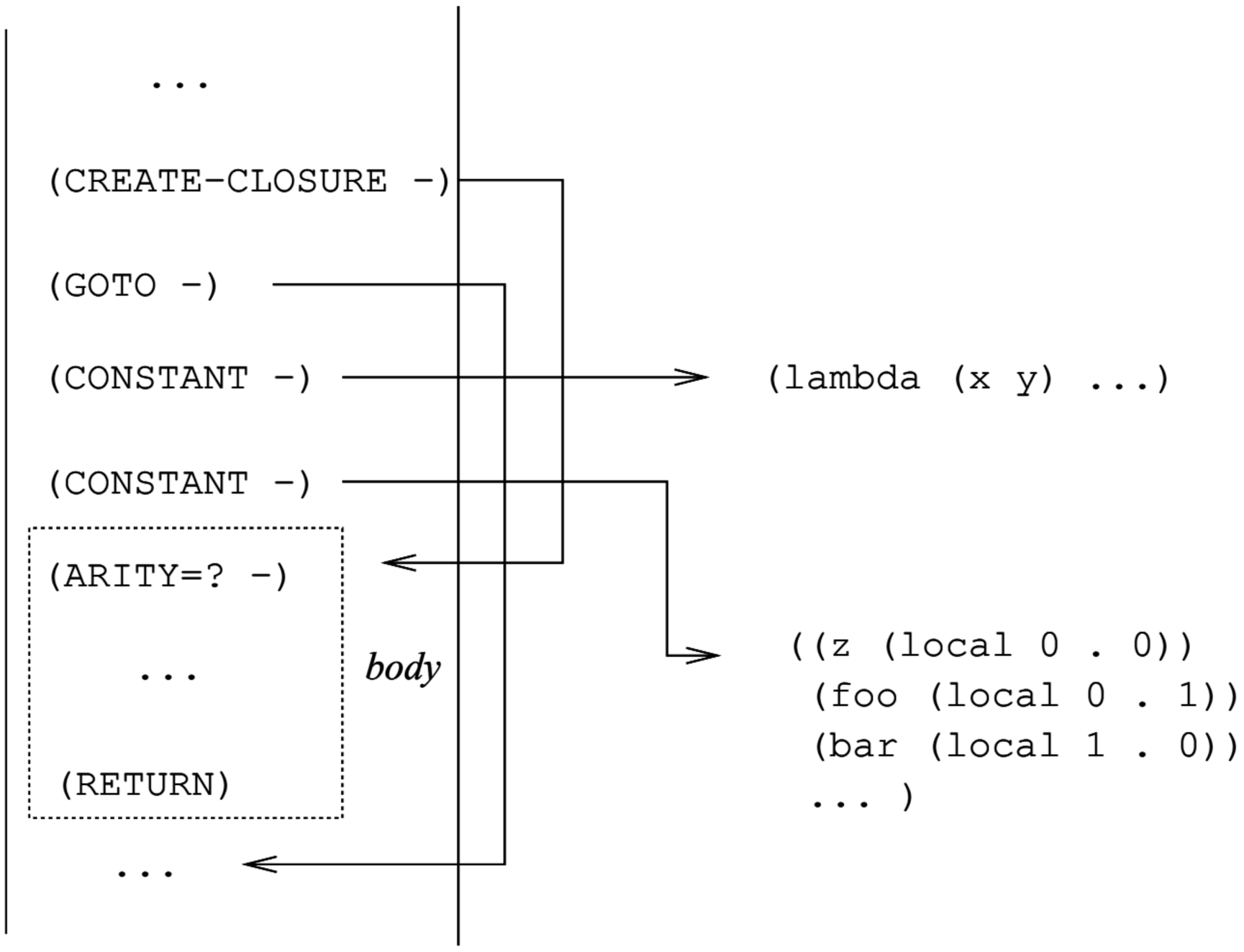


FIGURE 8.4 – Abstraction désossable

La programmation est triviale encore qu'il faille changer les générateurs de code puisqu'il faut leur fournir maintenant la définition et l'environnement statique de compilation dont auront besoin les fonctions `procedure->definition` et `procedure->environment`. La version pour fonction n-aire s'en déduit simplement [8].

```
(define (meaning-fix-abstraction n* e+ r tail?)
  (let* ((arity (length n*))
         (r2 (r-extend* r n*))
         (m+ (meaning-sequence e+ r2 #t)) )
    (REFLECTIVE-FIX-CLOSURE m+ arity `(lambda ,n* . ,e+) r) ) )
(define (REFLECTIVE-FIX-CLOSURE m+ arity definition r)
  (let* ((the-function (append (ARITY=? (+ arity 1)) (EXTEND-ENV)
                               m+  (RETURN) ))
         (the-env (append (EXPLICIT-CONSTANT definition)
                          (EXPLICIT-CONSTANT r) ))
         (the-goto (GOTO (+ (length the-env)
                            (length the-function) ))) )
    (append (CREATE-CLOSURE (+ (length the-goto) (length the-env)))
```

8. Utiliser `EXPLICIT-CONSTANT` permet de ne pas obtenir un `(PREDEFINED0)` à la place d'un `(CONSTANT '())`.

```
                the-goto the-env the-function ) ) )
```

La fonction `procedure->definition` trouve la définition d'une fermeture par la citation qui se trouve deux instructions avant le code de son corps.

```
(definitial procedure->definition
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (>= (activation-frame-argument-length *val*) arity+1)
           (let ((proc (activation-frame-argument *val* 0)))
             (if (closure? proc)
                 (let ((pc (closure-code proc)))
                   (set! *val* (vector-ref
                                *constants*
                                (vector-ref *code* (- pc 3)) ))
                   (set! *pc* (stack-pop)) )
                 (signal-exception
                  #f (list "Not a procedure" proc) ) ) )
           (signal-exception
            #t (list "Incorrect arity" 'enrich) ) ) ) ) ) )
```

La fonction `procedure->environment` extrait l'environnement clos de la fermeture et le réifie en lui adjoignant la carte nom-adresse le rendant intelligible. Cette carte est trouvée une instruction avant le code de son corps.

```
(definitial procedure->environment
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (>= (activation-frame-argument-length *val*) arity+1)
           (let ((proc (activation-frame-argument *val* 0)))
             (if (closure? proc)
                 (let* ((pc (closure-code proc))
                        (r (vector-ref
                            *constants*
                            (vector-ref *code* (- pc 1)) )) )
                   (set! *val* (make-reified-environment
                                (closure-closed-environment proc)
                                r ))
                   (set! *pc* (stack-pop)) )
                 (signal-exception
                  #f (list "Not a procedure" proc) ) ) )
           (signal-exception
            #t (list "Incorrect arity" 'enrich) ) ) ) ) ) )
```

En résumé, l'ajout des fonctions `procedure->definition` et `procedure->environment` accroît les possibilités des programmes en leur permettant de comprendre leur propre code. Elles rendent possible l'écriture d'outils de mise au point introspective. Elles ont néanmoins un coût non négligeable de par la taille des informations à conserver. Elles sont surtout d'une indiscrétion totale car, à tout pouvoir réifier, plus rien n'est occultable de façon définitive. Vous ne pouvez plus vendre du code que vous souhaitez garder secret, ni cacher l'implantation de certains types de données, ni même espérer que certaines optimisations locales auront lieu puisque plus rien ne peut être garanti constant.

On peut cependant pallier quelque peu ces défauts en créant une nouvelle forme

spéciale, disons `reflective-lambda`, de syntaxe suivante :

```
(reflective-lambda (variables) (exportations)
    body )
```

Comme les abstractions qu'elles généralisent, leurs variables sont spécifiées en premier paramètre et leur corps en dernier. Entre les deux, figure la clause d'exportation fixant les seules variables libres du corps qui seront inspectables de l'extérieur dans l'environnement que cette abstraction enclôt. Ainsi une abstraction normale `(lambda (`*variables*`)` *body*`)` n'est autre que `(reflective-lambda (`*variables*`) ()` *body*`)`, c'est-à-dire qu'elle n'exporte rien. En ne rendant que quelques liaisons visibles, on peut en celer d'autres et ainsi se prémunir des éventuels indiscrets.

Un autre problème se pose avec l'inquisitrice fonction `procedure->environment`. Quel est l'environnement exact qu'enclôt l'abstraction de l'exemple suivant ?

```
(let ((x 1)(y 2))
  (lambda (z) x) )
```

L'environnement enclos contient certainement `x`, qui est libre dans le corps de l'abstraction. Capture-t-il `y` qui est présent dans l'environnement lexical ambiant ? La programmation donnée plus haut capture aussi `y` car on peut assumer que le contrat de `procedure->environment` est de permettre d'évaluer n'importe quelle expression (pouvant contenir des `x` et `y`) comme si elle se trouvait dans l'environnement lexical où a été créée la fermeture. Ce n'est donc pas tant l'environnement enclos que retourne `procedure->environment` que l'entier environnement lexical de naissance de la fermeture.

8.6.5 La forme spéciale `import`

Bien souvent, la forme sur laquelle appeler l'évaluateur a une structure assez statique. C'était, par exemple, le cas dans la procédure `trace-procedure1`. Plutôt que de recompiler à la volée, on pourrait imaginer un sorte de précompilation qui permettrait de se brancher sur un environnement qui ne serait disponible qu'à l'exécution. La nouvelle forme spéciale `import` [QD96] répond à ce contrat. Elle a l'allure suivante :

```
(import (variables) environnement
    formes )
```

La forme spéciale `import` évalue les *formes* qui composent son corps dans un environnement lexical un peu particulier. Les variables libres du corps, dont les noms figurent dans la liste *variables* [9], sont à prendre dans *environnement*, les autres sont à prendre dans l'environnement lexical d'appel à `import`. La liste des variables est statique, non calculée ; l'*environnement* est évalué en premier, puis c'est le tour des *formes*.

Voici un exemple inspiré de MEROONET où l'on avait le problème de la représentation des fonctions génériques qui sont, à la fois, des objets et des fonctions. Permettre d'atteindre les variables encloses rend une fermeture manipulable comme un objet dont les champs sont ses variables libres. Ce point est la base de l'identification des fermetures et des objets [10]. Ajouter une méthode à une fonction générique, c'est, dans l'implantation de MEROONET, écrire cette méthode à sa bonne place dans le vecteur

9. Ne citer aucune variable pourrait vouloir dire qu'on les capture toutes.
10. Il y manque d'autres capacités comme de pouvoir spécialiser la classe des fonctions.

des méthodes. En supposant qu'une fonction générique enclose ce vecteur sous le nom `methods`, alors on peut directement écrire :

```
(define (add-method! generic class method)
  (import (methods) (procedure->environment generic)
     (vector-set! methods (Class-number class) method) ) )
```

On peut voir sur cet exemple l'intérêt de la forme `import` par rapport à la fonction `eval/b`. Les références aux variables `class` et `method` sont statiques tandis que seule la variable `methods` reste flottante et ne pourra être résolue que lorsque le second paramètre de `import` sera connu. On a donc gagné la compilation à la volée qu'exécutait `eval/b` et l'on voit que l'on n'a plus à synthétiser des programmes à compiler : `import` est une sorte d'`eval` dégraissé et plus statique puisque l'on connaît statiquement les seules références qui doivent rester flottantes. En ce sens `import` s'apparente au mécanisme de liaison quasi statique proposée en [LF93] ou à celui de [NQ89]. Le nom même d'`import` provient de son cousinage avec `export` : l'une produit des environnements sur lesquelles l'autre se branche. On tient aussi là un mécanisme rudimentaire de modules de première classe.

Comment implanter la forme spéciale `import` ? Comme c'est une forme spéciale, on ajoutera une ligne à l'analyseur syntaxique en chef, `meaning`, pour reconnaître cette forme :

```
...
   ((import) (meaning-import (cadr e) (caddr e) (cdddr e) r tail?)) ...
```

Le générateur d'octets associé stocke la liste des variables flottantes en sommet de pile, évalue l'environnement qui finira dans le registre `*val*`, exécute la nouvelle instruction `CREATE-PSEUDO-ENV` et procède à l'évaluation du corps de la forme `import` dans les cas terminaux ou pas.

```
(define (meaning-import n* e e+ r tail?)
  (let* ((m (meaning e r #f))
         (r2 (shadow-extend* r n*))
         (m+ (meaning-sequence e+ r2 #f)) )
    (append (CONSTANT n*) (PUSH-VALUE) m (CREATE-PSEUDO-ENV)
            (if tail? m+ (append m+ (UNLINK-ENV))) ) ) )
```

L'évaluation du corps de la forme spéciale `import` passe par la construction d'un pseudo-bloc d'activation (de la classe `pseudo-activation-frame`), qui a pour taille le nombre des variables flottantes mentionnées dans la forme `import`. Il se comporte comme un environnement en ce sens que l'on peut les chaîner tout comme les blocs d'activation mais il contient en fait les véritables adresses où chercher les variables flottantes ainsi que l'environnement d'où les extraire. Lors de la création de ce pseudo-bloc d'activation, on calcule par `compute-kind` où se niche la variable flottante et on enregistre non pas sa valeur mais son addresse dans le pseudo-bloc.

```
(define-instruction (CREATE-PSEUDO-ENV) 252
  (create-pseudo-environment (stack-pop) *val* *env*) )

(define-class pseudo-activation-frame environment
  ( sr (* address) ) )

(define (create-pseudo-environment n* env sr)
  (unless (reified-environment? env)
    (signal-exception #f (list "not an environment" env)) )
  (let* ((len (length n*))
         (frame (allocate-pseudo-activation-frame len)) )
    (let setup ((n* n*)(i 0))
```

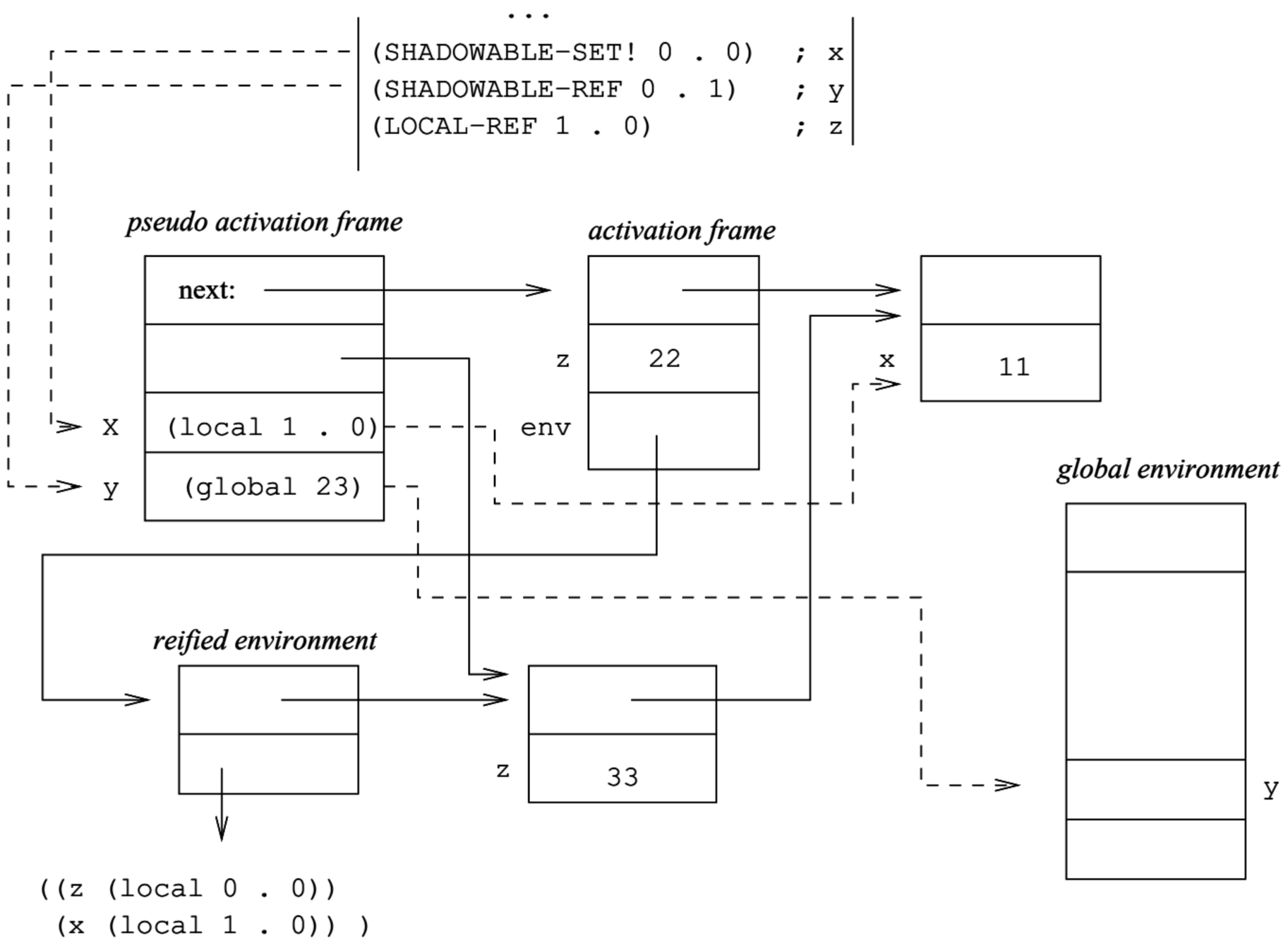


FIGURE 8.5 – Environnements dans `import`

```
  (when (pair? n*)
    (set-pseudo-activation-frame-address!
     frame i (compute-kind (reified-environment-r env) (car n*)) )
    (setup (cdr n*) (+ i 1)) ) )
(set-pseudo-activation-frame-sr! frame
                                 (reified-environment-sr env) )
(set-pseudo-activation-frame-next! frame sr)
(set! *env* frame) ) )
```

Le corps de la forme spéciale `import` doit être compilé spécialement en ce qui concerne les variables flottantes. Celles-ci sont repérées dans l'environnement de compilation `r` par la fonction particulière d'extension `shadow-extend*`. Comme, dit-on, il n'y a pas de problème informatique qu'une indirection bien placée ne saurait résoudre, l'adresse des variables flottantes sera trouvée au moyen d'une indirection. Lorsque l'on aura à chercher la valeur d'une variable flottante, on trouvera, dans un pseudo-bloc d'activation, son adresse que l'on utilisera pour l'extraire de l'environnement explicitement fourni. C'est ce qu'illustre la figure 8.5 pour le programme suivant :

```
(let ((x 11))
  (let ((z 22)
        (env (let ((z 33)) (export))) )
    (import (x y) env
      (list (set! x y) z) ) ) )
```

Les variables flottantes seront compilées spécialement et c'est pourquoi elles sont marquées non moins spécialement, par `shadow-extend*`, dans l'environnement de compilation.

```
(define (shadow-extend* r n*)
```

```
(let enum ((n* n*)(j 0))
  (if (pair? n*)
      (cons (list (car n*) `(shadowable 0 . ,j))
            (enum (cdr n*) (+ j 1)) )
      (bury-r r 1) ) ) )
```

On modifiera `meaning-reference`, ainsi que `meaning-assignment`, pour accepter ces nouveaux types de variable et l'on inventera deux nouvelles instructions gérant les accès à ces variables : `SHADOW-REF` et `SHADOW-SET!`.

```
(define (meaning-reference n r tail?)
  (let ((kind (compute-kind r n)))
    (if kind
        (case (car kind)
          ((checked-local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (CHECKED-DEEP-REF i j) ) )
          ((local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (SHALLOW-ARGUMENT-REF j)
                 (DEEP-ARGUMENT-REF i j) ) ) )
          ((shadowable)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (SHADOWABLE-REF i j) ) )
          ((global)
           (let ((i (cdr kind)))
             (CHECKED-GLOBAL-REF i) ) )
          ((predefined)
           (let ((i (cdr kind)))
             (PREDEFINED i) ) ) )
        (static-wrong "No such variable" n) ) ) )

(define-instruction (SHADOW-REF i j) 231
  (shadowable-fetch *env* i j) )

(define (shadowable-fetch sr i j)
  (if (= i 0)
      (let ((kind (pseudo-activation-frame-address sr j))
            (sr (pseudo-activation-frame-sr sr)) )
        (variable-value-lookup kind sr) )
      (shadowable-fetch (environment-next sr) (- i 1) j) ) )

(define (variable-value-lookup kind sr)
  (if (pair? kind)
      (case (car kind)
        ((checked-local)
         (let ((i (cadr kind))
               (j (cddr kind)) )
           (set! *val* (deep-fetch sr i j))
           (when (eq? *val* undefined-value)
             (signal-exception
              #t (list "Uninitialized local variable") ) ) ) )
        ((local)
         (let ((i (cadr kind))
               (j (cddr kind)) )
           (set! *val* (if (= i 0)
                           (activation-frame-argument sr j)
```

```
                        (deep-fetch sr i j) )) ) )
((shadowable)
 (let ((i (cadr kind))
       (j (cddr kind)) )
   (shadowable-fetch sr i j) ) )
((global)
 (let ((i (cdr kind)))
   (set! *val* (global-fetch i))
   (when (eq? *val* undefined-value)
     (signal-exception #t
       (list "Uninitialized global variable") ) ) ) )
((predefined)
 (let ((i (cdr kind)))
   (set! *val* (predefined-fetch i)) ) ) )
(signal-exception #f (list "No such variable")) ) )
```

La fonction `shadowable-fetch` va chercher à une adresse statiquement connue l'adresse dynamique décrivant où se trouve vraiment la variable. Il faut donc pouvoir décoder tous les types d'adresses possibles, à savoir `local` et `checked-local`, `global` et `predefined`, sans oublier `shadowable` qui peut aussi survenir : c'est ce qu'effectue la fonction `variable-value-lookup`. D'une certaine manière, la forme `import` correspond à une sorte de syntaxe transformant les références aux variables flottantes en des appels à `eval/b`. Pour reprendre l'exemple donné plus haut, la forme `(import (x y) env (list (set! x y) z))` n'est autre que [11] :

```
(list ((eval/b '(lambda (v) (set! x v)) env)
       (eval/b 'y env) )
      z )
```

Résumons tous les types de liaison vues.

Le mécanisme de liaison lexicale, celui que met en œuvre `lambda`, alloue des boîtes pour y mettre des valeurs. Les noms liés permettent de retrouver ces boîtes. Leur durée de vie est illimitée, c'est-à-dire qu'elles ne disparaîtront que lorsqu'elles seront inutiles. La portée des liaisons est restreinte au corps de la forme `lambda`.

Le mécanisme de liaison dynamique, celui que met en œuvre `dynamic-let`, associe un nom à une valeur [12] pendant toute la durée d'un certain calcul. La portée est illimitée durant le calcul, l'association disparaît après.

Le mécanisme de liaison quasi statique plaque des noms sur des liaisons déjà existantes. Comme en plus ce sont les noms mêmes de ces liaisons qui doivent resservir, l'introduction de ce mécanisme perturbe l'alpha-conversion puisque (même en l'absence de `procedure->environment`) la réification d'environnement capture les noms mêmes des liaisons et interdit la connaissance de tous les sites d'emploi [13].

---

11. Afin d'éviter d'engendrer un programme comportant une citation risquant d'être illégale, on a ici engendrée une fermeture prenant la valeur à affecter.

12. Le `dynamic-let` implanté ne permet pas de modifier cette association ; il n'y a pas de forme `dynamic-set!`. Il est simple de contourner cette restriction en liant au nom une donnée muable.

13. On pourrait augmenter la proposition de `reflective-lambda` esquissée plus haut pour, en plus, spécifier la nature des opérations permises sur les liaisons exportées en les limitant, par exemple, à n'être que lues. On pourrait aussi permettre, comme en [LF93], de renommer les variables libres à l'exportation.

8.6.6 Accès simplifié aux environnements

Les précédentes sections ont démontré que la plus grande partie du mécanisme d'évaluation peut être programmée explicitement sauf en ce qui concerne l'accès à l'environnement global. Les précédentes digressions nous ont permis d'apprécier les environnements de première classe aussi allons-nous généraliser les fonctions `global-value` et `set-global-value!` en `variable-value`, `set-variable-value!` et `variable-defined?` qui, respectivement, prennent un environnement de première classe et cherchent la valeur d'une variable, la modifient ou testent si elle est présente. Elles usent des fonctions de décodage des adresses dont usait déjà la fonction `shadowable-fetch`, tout en prenant garde de ne pas utiliser directement `compute-kind` qui crée, à la volée, les variables qu'on lui soumet. Nous omettrons `set-variable-value!` dont la définition se devinera facilement.

```
(definitial variable-value
  (let* ((arity 2)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (= (activation-frame-argument-length *val*) arity+1)
           (let ((name (activation-frame-argument *val* 0))
                 (env (activation-frame-argument *val* 1)) )
             (if (reified-environment? env)
                 (if (symbol? name)
                     (let* ((r (reified-environment-r env))
                            (sr (reified-environment-sr env))
                            (kind
                             (or (let ((var (assq name r)))
                                   (and (pair? var) (cadr var)) )
                                 (global-variable? g.current name)
                                 (global-variable? g.init name) ) ) )
                       (variable-value-lookup kind sr)
                       (set! *pc* (stack-pop)) )
                     (signal-exception
                      #f (list "Not a variable name" name) ) )
                 (signal-exception
                  #t (list "Not an environment" env) ) ) )
           (signal-exception #t (list "Incorrect arity"
                                      'variable-value )) ) ) ) ) )

(definitial variable-defined?
  (let* ((arity 2)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (= (activation-frame-argument-length *val*) arity+1)
           (let ((name (activation-frame-argument *val* 0))
                 (env (activation-frame-argument *val* 1)) )
             (if (reified-environment? env)
                 (if (symbol? name)
                     (let* ((r (reified-environment-r env))
                            (sr (reified-environment-sr env)) )
                       (set! *val*
                             (if (or (let ((var (assq name r)))
                                       (and (pair? var) (cadr var)) )
                                     (global-variable? g.current name)
```

```
                                    (global-variable? g.init name) )
                                #t #f ) )
                          (set! *pc* (stack-pop)) )
                        (signal-exception
                         #f (list "Not a variable name" name) ) )
                      (signal-exception
                       #t (list "Not an environment" env) ) ) )
              (signal-exception #t (list "Incorrect arity"
                                         'variable-defined? )) ) ) ) ) )
```

La fonction `variable-defined?` est une fonction d'introspection des environnements de première classe qui détermine si une variable y figure ou non. La question est intéressante puisque l'on peut notamment enrichir les environnements. Elle dépend cependant de la nature de l'environnement global : muable ou immuable. Si l'environnement global est immuable alors `variable-defined?` est une fonction dont la réponse est constante : si une variable ne figure pas dans l'environnement elle n'y figurera jamais. Si l'environnement global est muable alors il est capable de s'étendre tout en restant égal à lui-même (comme s'il existait une fonction `enrich!`) et `variable-defined?` peut, après avoir répondu Faux, répondre Vrai.

8.7 Interprète réflexif

Une mode est apparue, vers le milieu des années quatre-vingt, les interprètes réflexifs ; mode supportée par une remarquable trouvaille terminologique : les « tours réflexives ». On imagine déjà, s'érigeant des brumes d'un marais immensément plat, une tour dont le sommet se perd dans l'infini d'un ciel sombre dans le plus pur style d'une gravure d'un Rackham. En fait, les fondations de cette titanesque image s'ancrent dans les expérimentations menées autour des FEXPR d'InterLisp, mises à mort par Kent Pitman [Pit80], des environnements de première classe et des continuations.

Qui n'a rêvé en effet d'inventer ou de disposer d'un langage où tout serait redéfinissable, où l'imagination galoperait sans bride, où l'on pourrait jouir sans entrave d'une liberté programmatique totale ? Mais ce rêve se paye d'une lenteur exaspérante, par une quasi impossibilité de compilation, et d'une immersion au sein d'un monde sans loi ni gravité.

Quoiqu'il en soit, cette section va présenter un petit interprète réflexif dans lequel peu de choses sont cachées et intangibles. De nombreuses propositions de tels interprètes existent [dRS84], [FW84], [Wan86], [DM88], [Baw88], [Que89], [IMY92], [JF92]...Ils se distinguent les uns des autres par les vertus réflexives qu'ils procurent ainsi que par leur implantation.

Les interprètes réflexifs se doivent de permettre l'introspection ; ils doivent donc donner moyen aux programmes de saisir à tout moment le contexte de calcul, c'est-à-dire l'environnement lexical et la continuation. Pour cette dernière, `call/cc` existe déjà. Pour ce premier, on peut reprendre la forme `(export)`, aussi nommée `(the-environment)`, qui réifie l'environnement lexical courant. Si réifier est un des impératifs de la réflexivité, la façon de réifier n'est cependant pas indifférente et conditionne les opérations efffectuables.

Puisqu'une fois la borne franchie, il n'est plus de limite [Chr95], il faut aussi au-

toriser les programmes à définir de nouvelles formes spéciales. InterLisp avait expérimenté un mécanisme déjà partiellement présent en Lisp 1.5 sous le nom de FEXPR. Lorsqu'un tel objet est invoqué, au lieu de lui passer ses arguments, on lui fournit le texte de ses paramètres ainsi que l'environnement lexical courant : il pourra alors, à sa guise, les évaluer ou, plus généralement, les manipuler. On introduira donc la nouvelle forme spéciale `flambda` de syntaxe suivante :

`(flambda (`variables...`)` formes...`)`

La première variable recevra l'environnement lexical d'invocation tandis que les suivantes recevront le texte des paramètres d'appel. Avec un tel mécanisme, on pourra trivialement décrire la citation comme :

```
(set! quote (flambda (r quotation) quotation))
```

L'interprète réflexif doit aussi procurer les moyens de se modifier lui-même, ce qui procure toujours des sensations fortes. On fera donc en sorte que les fonctions implantant l'interprète soient accessibles aux programmes interprétés. La forme `(the-environment)` assurera cet effet puisqu'elle permet d'accéder aux variables de l'implantation comme si elles appartenaient aux programmes interprétés.

Voici donc l'interprète réflexif, dont on notera qu'il ne « pèse » que 1362 octets [14] ; ce n'est donc pas une grande dépense en mémoire que de procurer un tel interprète en bibliothèque. Il est écrit dans le langage que compile le compilateur d'octets.

```
(apply
 (lambda (make-toplevel make-flambda flambda? flambda-apply)
   (set! make-toplevel
         (lambda (prompt-in prompt-out)
           (call/cc
            (lambda (exit)
              (monitor (lambda (c b) (exit b))
                ((lambda (it extend error global-env
                          toplevel eval evlis eprogn reference )
                   (set! extend
                         (lambda (env names values)
                           (if (pair? names)
                               (if (pair? values)
                                   ((lambda (newenv)
                                      (begin
                                        (set-variable-value!
                                         (car names)
                                         newenv
                                         (car values) )
                                        (extend newenv (cdr names)
                                                (cdr values) ) ) )
                                    (enrich env (car names)) )
                                   (error "Too few arguments" names) )
                               (if (symbol? names)
                                   ((lambda (newenv)
                                      (begin
                                        (set-variable-value!
                                         names newenv values )
                                        newenv ) )
                                    (enrich env names) )
                                   (if (null? names)
```

14. Une fois compilé car son texte mesure environ 120 lignes, 6000 caractères et 583 doublets.

```
                                (if (null? values)
                                    env
                                    (error
                                     "Too much arguments"
                                     values ) )
                                env ) ) ) ) )
(set! error (lambda (msg hint)
                (exit (list msg hint)) ))
(set! toplevel
      (lambda (genv)
        (set! global-env genv)
        (display prompt-in)
        ((lambda (result)
           (set! it result)
           (display prompt-out)
           (display result)
           (newline) )
         ((lambda (e)
            (if (eof-object? e)
                (exit e)
                (eval e global-env) ) )
          (read) ) )
        (toplevel global-env) ) )
(set! eval
      (lambda (e r)
        (if (pair? e)
            ((lambda (f)
               (if (flambda? f)
                   (flambda-apply f r (cdr e))
                   (apply f (evlis (cdr e) r)) ) )
             (eval (car e) r) )
            (if (symbol? e)
                (reference e r)
                e ) ) ) )
(set! evlis
      (lambda (e* r)
        (if (pair? e*)
            ((lambda (v)
               (cons v (evlis (cdr e*) r)) )
             (eval (car e*) r) )
            '() ) ) )
(set! eprogn
      (lambda (e+ r)
        (if (pair? (cdr e+))
            (begin (eval (car e+) r)
                   (eprogn (cdr e+) r) )
            (eval (car e+) r) ) ) )
(set! reference
      (lambda (name r)
        (if (variable-defined? name r)
            (variable-value name r)
            (if (variable-defined? name global-env)
                (variable-value name global-env)
                (error "No such variable"
                       name ) ) ) ) )
((lambda (quote if set! lambda flambda monitor)
   (toplevel (the-environment)) )
```

```
(make-flambda
 (lambda (r quotation) quotation) )
(make-flambda
 (lambda (r condition then else)
   (eval (if (eval condition r) then else) r) ) )
(make-flambda
 (lambda (r name form)
   ((lambda (v)
      (if (variable-defined? name r)
          (set-variable-value! name r v)
          (if (variable-defined? name global-env)
              (set-variable-value!
               name global-env v )
              (error "No such variable"
                     name ) ) ))
    (eval form r) ) ) )
(make-flambda
 (lambda (r variables . body)
   (lambda values
     (eprogn body
             (extend r variables values) ) ) ) )
(make-flambda
 (lambda (r variables . body)
   (make-flambda
    (lambda (rr . parameters)
      (eprogn body
              (extend r variables
                      (cons rr
                            parameters
                            ) ) ) ) ) ) )
                  (make-flambda
                   (lambda (r handler . body)
                     (monitor (eval handler r)
                              (eprogn body r) ) ) ) )
                'it 'extend 'error 'global-env
                'toplevel 'eval 'evlis 'eprogn 'reference ) ) ) ) ) )
  (make-toplevel "?? " "== ") )
'make-toplevel
((lambda (flambda-tag)
   (list (lambda (behavior) (cons flambda-tag behavior))
         (lambda (o) (if (pair? o) (= (car o) flambda-tag) #f))
         (lambda (f r parms) (apply (cdr f) r parms)) ) )
 98127634 ) )
```

Cette définition est assez subtile. Les quelques explications qui suivent ne seront probablement point superflues. Ajoutons qu'il vaut mieux faire tourner cet interprète pour commencer à y croire !

La forme `(apply ...)` initiale crée quatre variables locales. Les trois dernières encodent (par `make-flambda`), reconnaissent (grâce à `flambda?`) et appliquent (au moyen de `flambda-apply`) les abstractions réflexives ou formes `flambda`. Ces objets sont représentés par des paires pointées ayant en `car` une étiquette égale au nombre `98127634`. Ces objets auront un protocole d'appel bien particulier et c'est pourquoi il nous faut les reconnaître. La quatrième variable, `make-toplevel`, est initialisée avec l'étrange valeur 98127634 puis affectée à sa valeur finale afin de pouvoir procurer ces quatre variables aux programmeurs, elle lance une boucle d'interaction à invites

particularisables. Ces invites sont, au début, `??` et ==.

Cette boucle d'interaction capture sa continuation d'appel à laquelle elle retournera en cas d'erreur ou d'appel à la fonction `exit` [15], qui sera aussi accessible des programmes. La boucle d'interaction elle-même est protégée par une forme `monitor` **[☞ p. 244]** puis introduit et initialise tout un lot de variables qui seront encore une fois partageables avec les programmes interprétés. La variable `it` est liée à la dernière valeur calculée par la boucle d'interaction. La fonction `extend` étend un environnement avec une liste de variables et de valeurs, elle procède aussi à la vérification d'arité. La fonction `error` imprime un message d'erreur puis termine par appel à `exit`.

La fonction `toplevel` implante l'interaction de façon classsique. Les fonctions accompagnant `eval`, à savoir `evlis`, `eprogn` et `reference` sont non moins standards mais ce qui l'est moins, c'est qu'elles sont aussi accessibles des programmes. La fonction `eval` est simplissime : soit l'expression à calculer est une variable ou une citation implicite, soit c'est une forme, auquel cas on évalue le terme en position fonctionnelle. Deux cas s'offrent alors : ou bien c'est une fonction réflexive, auquel cas on l'invoque en lui passant l'environnement courant et les paramètres d'appel, ou c'est une fonction normale, que l'on invoque normalement.

C'est cette souplesse même qui rend le langage défini par cet interprète non compilable. Supposons que l'on définisse la fonction normale suivante :

```
(set! stammer (lambda (f x) (f f x)))
```

On peut maintenant écrire les deux programmes :

```
(stammer (lambda (ff yy) yy) 33)     → 33
(stammer (flambda (ff yy) yy) 33)    → x
```

Comme cela découle de leurs définitions, dans un cas `stammer` exécute son corps, dans l'autre il réifie une partie de celui-ci pour le fournir à son argument `f`. Il faudrait donc compiler doublement toute application fonctionnelle de manière à pouvoir satisfaire les abstractions normales et réflexives. Mais comme tout programme représenté par une liste est possiblement une application fonctionnelle, on doit compiler de même (c'est-à-dire sous une double forme) les formes débutant par les habituelles formes spéciales comme `quote`, `if` et les autres car elles aussi peuvent être redéfinies pour faire toute autre chose. Bref, il n'y a pratiquement plus d'invariants sur lesquels le compilateur pourrait s'appuyer pour produire un code efficace, ce qui laisse libre champ aux joies de l'interprétation.

On définit quelques fonctions réflexives, juste avant de lancer la boucle d'interaction, parce que c'est plus simple de le faire là que par programme. On définit ainsi `quote`, `if`, `set!`, `lambda`, `flambda` et `monitor`. Celles qui manquent peuvent être explicitement ajoutées comme suit :

```
(set! global-env
      (enrich global-env 'begin 'the-environment) )
(set! the-environment
      (flambda (r) r) )
(set! begin
      (flambda (r . forms)
         (eprogn forms r) ) )
```

---

15. Voilà enfin un Lisp d'où l'on sait sortir ! On vérifiera que cela manque aux définitions de COMMON LISP, Scheme et même Dylan.

La variable `global-env` est liée à l'environnement réifié contenant toutes les précédentes fonctions et variables y compris, et c'est là la force du procédé, elle-même ! Mais non seulement l'environnement global est fourni aux programmeurs mais ils peuvent aussi le modifier avec rétroaction sur l'interprète : l'interprète et le programme partagent la même variable `global-env`. L'affecter par l'une ou l'autre des voies mène aux mêmes effets, aux mêmes plaisirs et aux mêmes dangers. Grâce à ce lien bidirectionnel, on peut mieux expliquer l'exemple ci-dessus, ou encore le suivant, où l'on définit une nouvelle forme spéciale : `when`.

```
(set! global-env (enrich global-env 'when))
(set! when
      (flambda (r condition . body)
        (if (eval condition r) (eprogn body r) #f) ) )
```

En premier, on étend l'environnement global avec une nouvelle variable globale, non initialisée bien sûr. On l'initialise immédiatement après avec une fonction réflexive appropriée. Ce qui est très fort est que l'extension de l'environnement global est visible de l'interprète : affecter `when` avec une abstraction réflexive ajoute bien une nouvelle forme spéciale à l'interprète.

Mais, me direz-vous, où peut bien s'ériger ce mythique bâtiment sorti des brumes que vous esquissiez naguère ? Les programmes peuvent se créer de nouveaux niveaux d'interprétation avec la fonction `make-toplevel`. Ils peuvent aussi s'auto-interpréter, on y gagne une lenteur sénatoriale rendant purement théorique cette possibilité. L'auto-interprétation est une qualité un peu différente de la réflexivité. Deux principales difficultés existent pour rendre cet interprète auto-interprétable. La première est d'éviter l'emploi de formes spéciales lorsque celles-ci sont redéfinies afin d'éviter d'être instable : c'est pourquoi le corps de l'abstraction qui lie `set!` et les autres est réduit à `(toplevel (the-environment))`. D'autre part les `flambda` doivent être reconnues par les deux niveaux d'interprètes et c'est pourquoi nous avons pris une étiquette `flambda-tag` unique mais falsifiable.

Une dernière possibilité est que l'on peut saisir (réifier) ce que l'on est en train de faire (continuation et environnement compris), réfléchir sur ce que l'on était en train de faire, imaginer ce que l'on aurait plutôt dû faire et se mettre à faire cela. Suivant les codages adoptés pour la réification, de nouvelles possibilités d'introspection sont offertes comme de scruter l'environnement ou les blocs de contrôle dans la continuation [Que93a]. Ce sont ces capacités d'introspection qui ont valu le nom de « réflexif » à ce type d'interprètes.

8.7.1 La forme `define`

Un interêt de l'interprète réflexif précédent est qu'il permet une définition opérationnelle de la très complexe forme `define` de Scheme. Comme on l'a déjà expliqué, `define` est une forme spécialement spéciale : elle agit comme une déclaration doublée d'une affectation. Affectation car après exécution, la variable aura une valeur bien définie. Déclaration, car elle introduit une nouvelle variable sitôt que le texte mentionnant `define` est préparé pour l'exécution. Cette déclaration est une mutation de l'environnement global. Ce sont tous ces aspects que représente la définition suivante de la forme spéciale `define`, à l'exclusion des aspects syntaxiques que `define`

possède aussi et qui lui permettent d'accepter des variantes comme `(define (foo x) (define (bar) ...) ...)` et de figurer dans diverses postures dites de `define` internes. On peut le mesurer, la forme `define` est spécialement complexe.

```
(set! global-env (enrich global-env 'define))
(set! define (flambda (r name form)
               (if (variable-defined? name r)
                   (set-variable-value! name r (eval form r))
                   ((lambda (rr)
                      (set! global-env rr)
                      (set-variable-value! name rr (eval form rr)) )
                    (enrich r name) ) ) ))
```

La forme `define` teste tout d'abord si la variable à définir appartient déjà à l'environnement global, auquel cas la définition n'est qu'une affectation (R4RS §5.2.1). Dans le cas contraire, une nouvelle liaison est créée au sein de l'environnement global, l'environnement global est mis à jour pour inclure cette nouvelle liaison et la valeur à affecter est calculée dans ce nouvel environnement afin d'autoriser les récursions.

8.8 Conclusions

Ce chapitre a présenté divers aspects de l'évaluation explicite qui peut apparaître sous la forme d'une forme spéciale ou d'une fonction. Suivant les qualités que l'on désire préserver dans les langages, on pourra préférer des formes dégraissées d'évaluation comme les environnements de première classe ou la portée quasi statique. On le voit : l'art de la conception linguistique est délicat et insoupçonnée est l'immensité de son champ de solutions.

Ce chapitre montre aussi clairement les heureux choix que Lisp a su prendre (ou plutôt son inspiré créateur, John McCarthy) qui tout en réduisant les codages/encodages et en mêlant les habituels niveaux de langage et de métalangage a pu augmenter si largement le champ expérimental.

8.9 Exercices

Exercice 8.1 : Pourquoi la fonction `variables-list?` ne teste-t-elle pas la non-cyclicité des listes de variables ?

Exercice 8.2 : La forme spéciale `eval/ce` compile à la volée l'expression qui lui est soumise, ce qui est du gâchis si l'on souhaite évaluer plusieurs fois de suite la même expression. Imaginez un dispositif corrigeant ce défaut.

Exercice 8.3 : Améliorez la définition de `eval/at` en fonction de `eval/ce` afin de vous débarrasser de la variable indûment capturée. On peut utiliser `gensym`.

Exercice 8.4 : L'utilisateur peut-il définir `variable-defined?` lui-même ?

Exercice 8.5 : Faites tourner l'interprète réflexif en pur Scheme.

8.10 Lectures recommandées

L'article [dR87] expose remarquablement la montée des aspects réflexifs en Lisp. [Mul92] propose des sémantiques algébriques pour les aspects réflexifs de Lisp tandis que, concernant les tours réflexives, on pourra consulter les récents travaux de [JF92] et [IMY92].

9

Macrologie

Ignorées, injustement décriées, insuffisamment justifiées théoriquement, les macros n'en demeurent pas moins l'une des assises les plus fondamentales de Lisp et n'ont en effet pas peu contribué à l'extraordinaire longévité de ce langage. Si les fonctions permettent d'abstraire des calculs, les objets des données, les macros, elles, permettent d'abstraire des structures de programmes. Ce chapitre va présenter ce que sont les macros et les problèmes qu'elles posent. C'est de loin l'un des sujets les moins étudiés de Lisp et au sein duquel la plus grande variabilité existe parmi les implantations de Lisp ou Scheme. Ce chapitre contient peu de programmes mais tentera, en une suite de notices ponctuelles, de baliser le domaine où évoluent ces êtres mal connus : les macros.

Inventées par Timothy P. Hart [SG93] en 1963, peu après la publication du manuel de référence de Lisp 1.5, les macros se sont avérées à l'usage l'un des ingrédients essentiels de Lisp. Les macros autorisent les programmeurs à imaginer et mettre en œuvre le langage approprié avec lequel exprimer leur problème. À l'instar des mathématiques où l'on ne cesse d'inventer de nouvelles abréviations appropriées à l'expression de concepts — et l'on ne mettra jamais assez en exergue tout le pouvoir expressif qu'offrent des notations adaptées — les dialectes de Lisp autorisent d'étendre le langage au moyen de nouvelles constructions syntaxiques. Que l'on ne s'y trompe pas, il ne s'agit pas d'augmenter le langage au moyen d'une bibliothèque de fonctions couvrant un domaine particulier : un Lisp doté d'une bibliothèque graphique de dessin n'est pas plus qu'un Lisp. Les extensions visées concernent l'introduction de nouvelles formes syntaxiques qui, elles, accroissent véritablement le pouvoir linguistique du programmeur.

Étendre un langage, c'est introduire une nouvelle notation énonçant que l'on pourra écrire X quand on signifiera Y. À chaque fois que le programmeur écrit X, il aurait pu, s'il était moins paresseux, écrire directement Y. Mais, justement il est intelligemment paresseux et élimine, au profit d'une forme simple, les détails insipides qui, ainsi, n'encombrent plus sa pensée. Pour assurer plus de souplesse, la règle d'abréviation est en général paramétrable : lorsque l'on écrit $X(t_1, \ldots, t_n)$, on entend de fait $Y(t_1, \ldots, t_n)$. Les macros ne sont pas un « truc » mais une technique d'abstraction éminemment utile, opérant sur les programmes par le biais de leur représentation.

La plupart des langages impératifs n'ont qu'un nombre figé de formes spéciales.

On peut y trouver la boucle `while` et, parfois, la boucle `until` mais si un problème particulier demande, par exemple, un nouveau type de boucle, il n'est en général pas possible de l'y ajouter. En Lisp, il suffira d'introduire cette nouvelle notation. Voici un exemple, chaque fois que l'on écrira `(repeat :while` *p* `:unless` *q* `:do` *body*`...)`, on entendra en fait :

```
(let loop ()
  (if p (begin (if (not q) (begin body...))
               (loop) )) )
```

Cet exemple est volontiers outrancier : il prend des mots-clés superflus (reconnaissables par les deux-points : les préfixant) alors que l'usage en Lisp est plutôt de se passer de ce bruit syntaxique. **[☞ Ex. 9.1]** Il introduit une variable locale, `loop`, qui masque éventuellement une variable de même nom que p, ou q ou $body$, pourrait vouloir référencer. Les amateurs de boucles pourront aussi se référer à l'Everest des boucles (la macro `loop` dont l'implantation populaire mesure plusieurs dizaines de pages) décrit en [Ste90, chapitre 26].

Malheureusement, comme beaucoup de concepts et particulièrement les plus avancés ou les plus subtils, les macros peuvent être dévoyées. Le but de ce chapitre est d'en dévoiler les problèmes et d'en exhiber les beautés. Pour ce faire, nous allons procéder à une reconstruction logique des macros qui nous permettra de recenser les problèmes et les sources de variantes.

9.1 La préparation

Les évaluateurs évolués des chapitres immédiatement précédents distinguaient deux phases dans le traitement des programmes : la *préparation* (c'est le mot utilisé dans IS-Lisp) suivie de l'*exécution*. L'évaluation **[☞ p. 175]** peut donc être vue comme `(run (prepare` *expression*`))`. C'était très nettement apparent dans l'interprète rapide et dans le compilateur vers du code-octet où les expressions préparées étaient successivement un arbre de **[☞ p. 213]** suspensions ou un vecteur d'octets. Au pire, dans les tous premiers interprètes, pouvait-on assimiler la préparation à l'identité.

La préparation elle-même peut être divisée en de multiples phases. Par exemple, sachant que les expressions à évaluer sont initialement des chaînes de caractères, de nombreux systèmes Lisp procurent le concept de macro-caractères permettant d'influencer l'analyse syntaxique des chaînes lors de leur transformation en une Sexpression. L'exemple le plus marquant de macro-caractères est celui de l'apostrophe qui, lorsqu'elle est lue, lit l'expression qui suit et l'insère dans une forme préfixée par le symbole `quote` : ce que l'on peut matérialiser par `(list (quote quote) (read))`. Donner la possibilité d'influencer le lecteur de Lisp, la fonction `read`, permet, dans une certaine mesure, d'adapter encore plus le langage à ses besoins. C'est par exemple le cas de Bigloo [SW94] qui, lorsqu'il compile un programme écrit en Caml Light, choisit une fonction de lecture appropriée dont le seul contrat est de lui retourner une Sexpression à compiler : cette fonction de lecture, ce sont les premières passes du compilateur de Caml Light [LW93]. Nous ne nous appesantirons pas plus sur les macro-caractères qui sont très dépendants de l'algorithme de lecture des Sexpressions.

La phase de préparation est souvent assimilée à une phase de compilation, celle-ci pouvant être de complexité très variable. L'enjeu important est de maintenir une stricte séparation entre cette phase et la suivante (l'exécution) afin de clarifier leur relation, de minimiser leur interface commune et surtout d'assurer la répétabilité des expériences. Deux visions s'affrontent, que nous qualifierons d'hypothèse du *monde central* et des *mondes multiples*.

9.1.1 Mondes multiples

Avec des mondes multiples, le processus de préparation d'une expression s'achève par la production d'un résultat souvent stocké dans un fichier (un `.o` ou `.fasl` classiquement). L'exécution d'un tel fichier est le fait d'un processus distinct qui parfois s'adjoint des facultés de regroupement d'expressions préparées séparément (ou *édition de liens*). C'est le mode qu'utilise la plupart des langages classiques où existe la notion de compilation indépendante ou séparée et qui a l'intérêt de permettre de factoriser les préparations (et aussi de mieux gérer les espaces de noms grâce aux directives d'importation et d'exportation). Les mondes sont dits multiples parce que l'expression préparée est la seule communication qui peut exister entre le préparateur d'expressions et l'évaluateur final ; ces deux processus sont dans des mondes distincts ; aucun effet du premier n'est perceptible du second :

$$\left.\begin{array}{ll} \textit{expression}_0 & \xrightarrow{\texttt{prepare}} \textit{expression-préparée}_0 \\ \textit{expression}_1 & \xrightarrow{\texttt{prepare}} \textit{expression-préparée}_1 \\ \ldots & \xrightarrow{\texttt{prepare}} \ldots \\ \textit{expression}_n & \xrightarrow{\texttt{prepare}} \textit{expression-préparée}_n \end{array}\right\} \texttt{run}$$

9.1.2 Monde central

En revanche, l'hypothèse du monde central stipule que le centre de l'univers est la boucle d'interaction fondamentale (le *toplevel*) et que, donc, tous les calculs lancés depuis celle-ci cohabitent dans un même espace mémoire où l'on ne peut limiter, ni même contrôler, les communications par altération de ressources globalement visibles (variables globales, listes de propriétés, etc.). Dans ce monde central, la boucle d'interaction enchaîne successivement la lecture d'une expression, sa préparation puis son évaluation. Cette boucle d'interaction joue le rôle d'un interprète de commandes (un véritable *shell*) mais permet néanmoins de préparer des expressions sans les évaluer immédiatement après : c'est ce que fait la classique fonction `compile-file` dans de nombreux systèmes. La fonction `compile-file` prend le nom d'un fichier contenant une séquence d'expressions, la prépare et produit un fichier préparé. L'évaluation d'un fichier préparé peut aussi s'effectuer depuis la boucle d'interaction au moyen de la fonction `load` ou de l'une de ses dérivées. On voit ainsi que l'on peut factoriser la préparation d'expressions et donc entrelacer préparation et évaluation.

L'hypothèse du monde central n'est pas mauvaise en soi ; elle n'est que le reflet du monde unique que procure un système d'exploitation comme UN⋆X, où l'utilisateur a un état interne sur lequel il compte et qui repose à la fois sur le système de fichiers et sur les variables locales ou globales, les alias, etc. que lui procure son interprète de

commandes favori : `sh`, `tcsh` ... Mais l'important est d'assurer, le plus simplement possible, la répétabilité des expériences.

Lorsque l'on bâtit un logiciel, il est bon d'avoir une méthode sûre pour obtenir un exécutable à partir de ce logiciel. Deux reconstructions partant des mêmes sources doivent aboutir au même résultat : c'est un principe intellectuel bien ancré. Il est, sans trop de problème, assuré par les mondes multiples parce que l'on contrôle facilement les informations que l'on soumet en entrée. Il est plus difficile à mettre en œuvre avec l'hypothèse du monde central car l'on maîtrise mal son état interne et les communications cachées [1]. Le préparateur (le compilateur) au lieu d'être réinitialisé à chaque préparation, reste le même et se modifie peu à peu au fil de celles-ci puisqu'il travaille dans une image mémoire sans cesse enrichie et toujours moins maîtrisée.

En résumé, la préparation d'un programme se doit d'être un processus totalement maîtrisable !

9.2 La macroexpansion

On souhaite disposer d'un mécanisme élaboré d'abréviation qui ne peut qu'appartenir à la phase de préparation. Cette dernière peut donc être scindée en deux parties : la *macroexpansion* suivie de la préparation proprement dite. Deux écoles encore s'affrontent sur ce nouveau terrain : la macroexpansion *endogène* ou *exogène*. La préparation d'une expression peut se matérialiser par la succession `(really-prepare (macroexpand` *`expression`*`))` et nous ne nous intéresserons qu'au *macroexpanseur*, c'est-à-dire à la fonction qui implante le processus de macroexpansion : `macroexpand`. D'où vient-elle ?

9.2.1 Mode exogène

L'école exogène [QP91b, DPS94b] stipule que la fonction `macroexpand` est fournie *indépendamment* de l'expression à préparer, par exemple par la donnée de directives de préparation. La fonction `prepare` ressemble donc un peu à :

```
(define (prepare expression directives)
  (let ((macroexpand (generate-macroexpand directives)))
    (really-prepare (macroexpand expression)) ) )
```

Plusieurs mises en œuvre existent que nous allons illustrer avec le compilateur de code-octet naguère présenté. **[☞ p. 253]** On supposera que les principales fonctionnalités de cette chaîne de compilation sont : `run` pour faire tourner une application et `build-application` pour en construire une. On supposera aussi que `compile.so` est l'application correspondant au compilateur entier. Dans les exemples qui suivent, la directive de macroexpansion mentionne simplement le nom de l'application chargée de la macroexpansion.

La macroexpansion peut s'effectuer en cascade comme dans le haut de la figure 9.1. Dans ce cas une commande comme "`compile file.scm expand.so`" (où le fi-

1. Qui connaît le sens de toutes les variables d'environnement que lui révèle `printenv` sous UN⋆X et de tous les fichiers `.rc` dont son confort dépend ?

chier à compiler est précisé en première position et le macroexpanseur en seconde position) correspond (en termes UN⋆X-iens) à :

```
run expand.so < file.scm | run compile.so > file.so
```

La macroexpansion peut aussi s'effectuer par synthèse d'un nouveau compilateur (ici `tmp.so`), voir le bas de la figure 9.1. La commande "`compile file.scm expand.so`" est alors analogue à :

```
( build-application compile.so expand.so -o tmp.so ;
  run tmp.so ) < file.scm > file.so
```

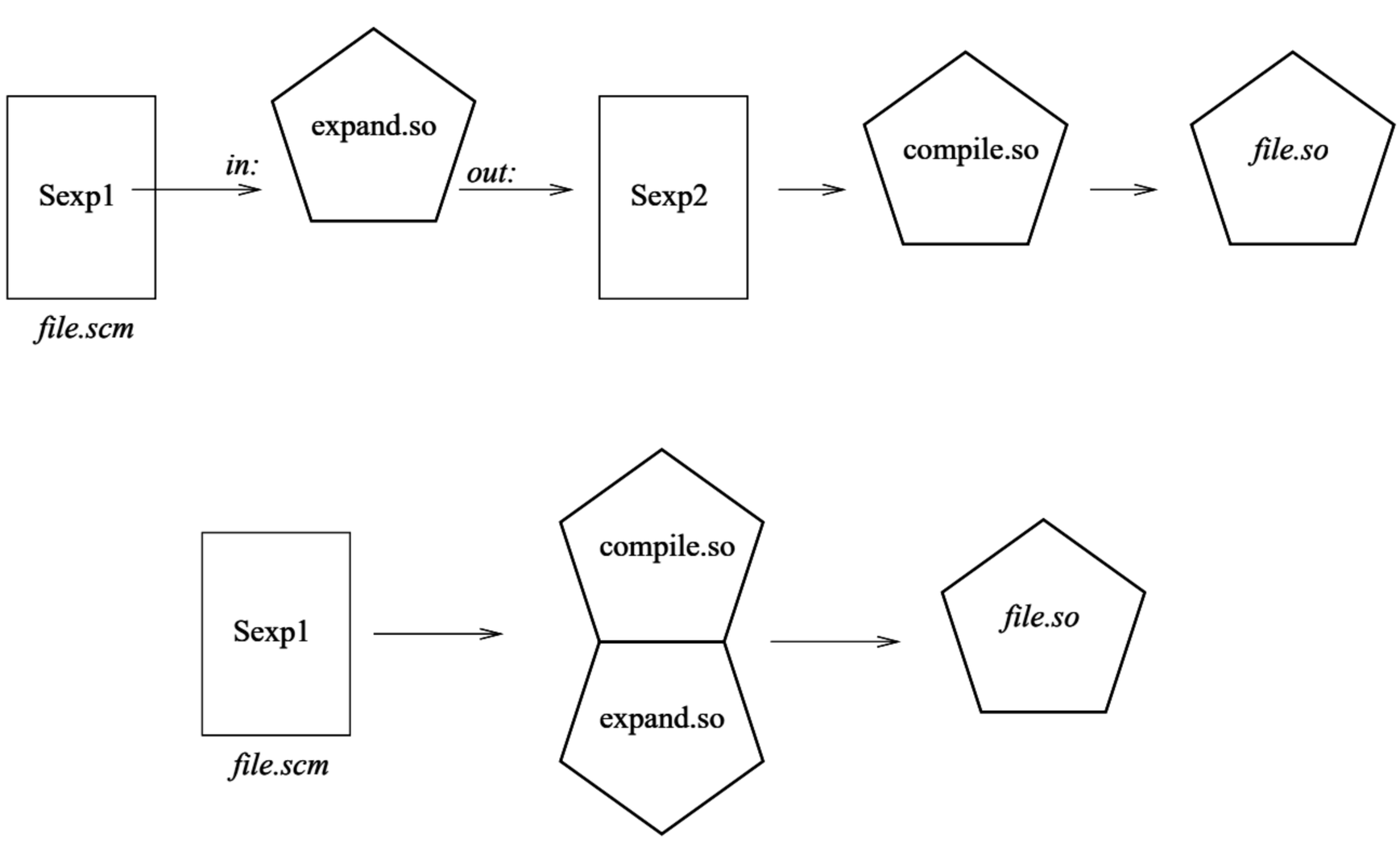


FIGURE 9.1 – Deux exemples de mode exogène (avec le compilateur de code-octet) ; les pentagones représentent les applications.

Parmi les problèmes masqués, il reste à définir un protocole d'échange d'informations entre le préparateur et le macroexpanseur. Ce dernier doit recevoir l'expression à expanser (il peut la recevoir comme une valeur normale ou sous la forme d'un nom de fichier dans lequel il devra la lire), il doit ensuite renvoyer le résultat à son appelant (et, là encore, il peut retourner une Sexpression ou le nom d'un fichier dans lequel est écrit le résultat). Passer par des fichiers n'est pas absurde : c'est la technique utilisée par le compilateur C (`cc` et `cpp`) et elle garantit l'absence de communications cachées. Passer par des Sexpressions nécessite par contre que le préparateur sache appeler (évaluer ou charger, par `load`) dynamiquement le macroexpanseur, puisqu'ils résident tous deux dans le même espace mémoire.

Plutôt que d'imposer un macroexpanseur monolithique, on peut aussi vouloir le bâtir à partir d'éléments plus petits. On peut donc souhaiter composer diverses abréviations. Cela revient à étendre le langage des directives afin de pouvoir spécifier le rassemblement d'abréviations. Cela bien sûr impose que les macros soient composables, ce qui implique la définition de protocoles assurant, par exemple, l'associativité et la commutativité des définitions de macros.

Ainsi donc, la vision exogène associe, à un programme, un macroexpanseur indépendamment bâti. Cette organisation permet d'assurer la répétabilité des expériences

ainsi que la non persistence de débris provenant de la macroexpansion dans les programmes préparés. Outre la parfaite séparation recherchée entre macroexpansion et évaluation, cette solution ne limite pas l'imagination car n'importe quel macroexpanseur est licite pourvu qu'il conduise à une Sexpression. On peut même utiliser `cpp`, `m4` ou `perl` pour cette tâche ! Toutefois, il est certain que pour spécifier un algorithme transformant des Sexpressions, Lisp semble le plus adéquat. On voit cependant bien au sein de la vision exogène qu'il n'y a aucune connexion entre le langage du macroexpanseur et le langage en cours de préparation. Le macroexpanseur n'a qu'à rechercher les sites à expanser puisque les définitions de macros ne sont pas dans le texte à préparer.

9.2.2 Mode endogène

L'école endogène se passe de directives et affirme que toute l'information nécessaire à l'expansion se trouve *dans* l'expression à préparer. En d'autres termes, la fonction de préparation est définie comme :

```
(define (prepare expression)
  (really-prepare (macroexpand expression)) )
```

La différence fondamentale avec la variante précédente est que maintenant l'algorithme de macroexpansion préexiste, peut uniquement être paramétré et seulement dans ses limites propres. L'algorithme prédéfini de macroexpansion a donc maintenant la double tâche de trouver les définitions de macros et non plus seulement les sites d'emploi de ces dites macros. La définition des abréviations passe donc par un texte, une Sexpression débutant souvent par un mot-clé comme `define-macro` ou `define-syntax`, qu'il s'agit de convertir en un expanseur, c'est-à-dire en une fonction dont le rôle est de transformer une abréviation en le texte qu'elle représente : le macroexpanseur contient donc l'équivalent d'une fonction convertissant un texte en un programme ! Là fut alors le coup de génie : plutôt que d'inventer un langage spécial pour les macros, on utilisera la fonction `eval` tout simplement. En d'autres termes, le langage de définition des macros est Lisp !

Si là est le génie, là est aussi le problème. Les macros relèvent d'une technique subtile qui a été longtemps assimilée à la magie noire : la macrologie ! En fait, c'est précisément cette identification de langages qui brouille la perception de ce que sont vraiment les macros. Longtemps elles furent vues comme de vraies fonctions sauf que leur protocole d'appel était étrange : les arguments n'étaient point évalués mais le résultat de la macro l'était. Ce modèle, s'il pouvait convenir à un monde central interprété, a été abandonné à la suite des avancées compilatoires et notamment de l'article de Kent Pitman [Pit80]. Une macro est un processus dont le rôle social est de convertir des textes remplis d'abréviations en de nouveaux textes dénués d'abréviations.

Que le langage de définition des macros soit Lisp est d'ailleurs combattu dans certaines propositions de macros où ce langage est restreint à un langage de filtrage : ce sont les macros proposées en appendice du R^4RS. On perd en expressivité puisque certaines transformations ne sont plus exprimables mais on gagne en clarté pour les macros simples qui forment néanmoins l'essentiel des macros que l'on écrit. Pour donner un exemple, les programmes associés à ce livre contiennent (à ce moment de la rédaction) 63 définitions de macros usant de `define-syntax` contre seulement 5 em-

plois de la plus libérale `define-abbreviation` [2] ; les trois derniers emplois de celles-ci décrivent les macros fondamentales de MEROONET (`define-class`, `define-generic` et `define-method`) qui ne peuvent être programmées avec `define-syntax`.

En résumé, que l'on impose son procédé de macroexpansion ou que l'on se contente de paramétrer un mécanisme préexistant (au prix de l'incorporation d'un évaluateur local à la macroexpansion), les deux modèles procurent préférentiellement Lisp comme langage de description des macros. Contrairement au mode exogène, le mode endogène impose la présence d'un évaluateur au sein du macroexpanseur : c'est son trait dominant. Il importe cependant de ne point confondre le langage d'écriture des macros d'avec le langage qui fait l'objet de la préparation : ils sont apparentés mais non confondus. Lorsque d'antédiluviens manuels décrivent les macros comme des mécanismes opérant par double évaluation, ils péchent par omission en ne mentionnant pas que les deux évaluations relèvent de deux évaluateurs différents.



FIGURE 9.2 – Macroexpansions exogène (à gauche) et endogène (à droite)

9.3 Appels aux macros

Le macroexpanseur doit trouver où se situent les fameuses abréviations qu'il doit remplacer. Un certain nombre de langages ont des analyseurs syntaxiques élaborés qu'il est possible d'étendre avec de nouvelles productions grammaticales fixant les extensions syntaxiques souhaitées, par exemple [Car93]. Ce peut être aussi simple que des déclarations syntaxiques qualifiant un opérateur de unaire/binaire ou préfixe/infixe/postfixe ; on peut souvent aussi ajouter une précédence numérique.

En Lisp, la représentation des Sexpressions favorise l'extraction du `car` des formes aussi le schéma suivant est-il le plus répandu : une liste dont le `car` est un mot-clé est un appel à la macro portant ce même nom. Ce schéma a de nombreuses vertus : il est simple, extensible et uniforme. Une abréviation (une macro) porte un nom qui est associé à une fonction (un *expanseur*). L'algorithme de macroexpansion est donc assez simple : il parcourt récursivement l'expression à traiter et, dès qu'il identifie un appel à une macro, il invoque l'expanseur associé en lui fournissant la Sexpression déclencheuse. L'expanseur peut trouver dans le `cdr` de cette expression de quoi paramétrer l'expansion à laquelle il se livre.

Se rencontre aussi la notion de macrosymbole, comme en COMMON LISP, où l'on peut lier le déclenchement d'une abréviation à l'occurrence d'un symbole. Cela permet souvent une forme plus aérée, dégagée de parenthèses superflues lorsque l'on

2. Nous avons pris ce nom exotique afin de ne pas entrer en conflit avec les diverses sémantiques que confèrent à `define-macro` les différents évaluateurs Lisp ou Scheme.

associe un traitement à ce qui ressemble superficiellement à une variable. Un macro-symbole permet de déclencher une abréviation à la lecture d'une variable, on peut également considérer qu'une forme `(set! _ π)` est un déclencheur qui survient lors de l'écriture dans une variable.

Une macro n'est donc que l'association d'une fonction d'expansion à un nom assortie d'une condition de déclenchement. C'est une sorte de liaison dans ce qui apparaît comme un nouvel espace de noms réservé aux macros. La macro n'est précisément ni la liaison ni l'expanseur ni la condition de déclenchement mais le tout que forment les trois au sein du macroexpanseur. Comme tout espace de noms convenablement géré, on souhaite au minimum pouvoir y définir des macros globales et locales : `define-abbreviation` et `let-abbreviation` permettent d'introduire de telles macros.

Le but de la fonction de macroexpansion est de convertir une Sexpression en un programme assimilable. La conséquence est que la recherche des sites d'appels de macros s'effectue sur une Sexpression et non sur un programmme. Certes, cette Sexpression comporte des ilôts qui sont déjà des programmes ; certes aussi les sites d'appels ressemblent à des applications fonctionnelles mais fondamentalement ce n'est qu'une Sexpression et rien n'interdit une macro d'offusquer le bon goût. Ainsi la Sexpression `(foo . 5)` pourrait-elle être définie comme une abréviation de `(vector-ref foo 5)` !

Le problème du macroexpanseur est donc d'arpenter une Sexpression : une structure grammaticalement molle au regard de la grammaire des programmes. Cela pose certains problèmes de préséance entre les macros et le comportement lexical. Ainsi, si `bar` est une macro (et que `lambda` n'en est point une), l'expression `(bar 34)` figurant dans `((lambda (bar) (bar 34)) ...)` est-elle un site d'emploi de la macro `bar` ? Le problème vient de la présence de la variable locale nommée aussi `bar` : masque-t-elle la macro `bar` ? Le R^{4}RS prend position en faveur du respect de la lexicalité.

Il est malheureusement aussi nécessaire d'exposer un peu l'algorithme d'expansion de manière à savoir, lorsqu'il arpente une Sexpression, les positions en lesquelles il peut trouver des sites d'appel à des macros. Par exemple, peut-on écrire `(let (foo) ...)` dans un contexte où `(foo)` est une macro engendrant une suite de liaisons ? De nombreux autres exemples existent comme `(cond (foo) ...)` ou `(case key (foo) ...)`. Il est de bon goût en Lisp de respecter le plus possible la grammaire des formes de manière à rendre confuse la différence entre fonction et macro. Sans information supplémentaire, `(bar ...)` peut être un appel à la fonction `bar` ou un site d'appel à la macro de même nom.

9.4 Expanseurs

Quel est le contrat d'un expanseur ? Deux solutions au moins coexistent. En *mode classique*, le résultat délivré par l'expanseur est considéré comme une Sexpression qui peut encore comporter des abréviations, c'est pourquoi elle est réexpansée jusqu'à n'en plus contenir. Si l'on nomme `find-expander` la fonction qui associe à un nom de macro son expanseur, alors l'expansion d'un appel à une macro `(foo . expr)` peut se définir comme :

```
(macroexpand '(foo . expr)) →
    (macroexpand ((find-expander 'foo) '(foo . expr)))
```

Un autre mode existe, plus compliqué, qui se nomme *Expansion Passing Style* (EPS) [DFH88]. Là, le contrat est que l'expanseur doit retourner l'expression totalement débarrassée de ses abréviations : il n'y a donc pas de `macroexpand` attendant le retour de l'appel à l'expanseur. La difficulté est que l'expanseur doit expanser récursivement les sous-termes composant l'expression qu'il engendre. Il ne peut le faire que s'il sait lui-même invoquer le mécanisme de macroexpansion. Une solution est de faire en sorte que les expanseurs aient accès à la variable globale `macroexpand`, dont la valeur est la fonction de macroexpansion. Une autre plus astucieuse prend en compte le besoin de macros locales et modifie le protocole d'appel des expanseurs de manière à ce qu'ils prennent, en second argument, la fonction courante de macroexpansion. Lispiennement on a alors :

```
(macroexpand '(foo . expr)) →
    ((find-expander 'foo) '(foo . expr) macroexpand)
```

Les deux systèmes ne sont pas équivalents et clairement EPS est au moins supérieure au mode classique puisqu'elle permet de le programmer. Une différence majeure de comportement concerne le besoin de macros locales. Supposons que l'on désire introduire localement une abréviation. On peut en EPS, étendre la fonction `macroexpand` reçue pour traiter cette abréviation localement sur les sous-termes. En mode classique, par exemple en COMMON LISP, la même demande est traitée ainsi : une macro spéciale `macrolet` modifie l'état interne du macroexpanseur pour y ajouter les macros locales, expanse son corps puis remodifie l'état interne pour y retirer les macros qu'elle avait précédemment introduites. La macro `macrolet` est primitive car on ne pourrait la recréer si d'aventure elle manquait. Il n'est aussi pas possible d'inactiver, en mode classique, une macro externe au `macrolet` (elle est forcément visible ou masquée par une macro locale) alors qu'en EPS, on peut introduire localement une syntaxe arbitrairement différente de la syntaxe ambiante : il suffit d'employer récursivement sur ses sous-termes une autre fonction que la fonction `macroexpand` reçue. De nombreux exemples de trace ou de curryfication sont présentés en [DFH88].

La plupart des abréviations sont des effets locaux, c'est-à-dire qu'elles ne conduisent qu'à une substitution ponctuelle d'un texte par un autre. Avoir accès à la fonction d'expansion courante, comme le permet EPS, est important car on peut alors programmer simplement des transformations plus complexes que celles que procure le mode classique sans avoir à programmer un arpenteur de code. Pensez, en guise d'exemple, à la transformation de programmes visant à introduire des boîtes. **[☞ p. 108]** Il est possible de le traiter en EPS en effectuant un premier arpentage de l'expression pour y trouver les variables liées et modifiables puis un second arpentage pour transformer toutes les références (tant en lecture qu'en écriture) à ces variables.

Par contre, EPS n'est pas toute puissante et certaines transformations sont hors de sa portée. **[☞ p. 135]** L'extraction des citations n'est pas une transformation ponctuelle car elle impose, lorsque l'on rencontre une citation, de la transformer en une référence à une variable globale. Jusque-là pas de problème, mais on se doit aussi de créer cette variable globale (par une forme `define` insérée au bon endroit), ce qui n'est pas un effet localement textuel ! Une solution serait d'enserrer le programme tout entier dans une macro, disons `with-quotations-extracted`, qui retournerait une séquence de définitions de citations suivie du programme transformé.

D'autres macros peuvent souhaiter créer des variables globales. C'est le cas, par

exemple, de la macro `define-class` de MEROONET, qui peut syntaxiquement apparaître au sein d'une forme `let` et dont le contrat est de créer une variable globale contenant l'objet représentant la classe. Ce problème n'est généralement pas traité par les systèmes de macros et impose alors que `define-class` soit une forme spéciale ou une macro employant des fonctions magiques, c'est-à-dire connues des seules implantations.

Si EPS est si intéressante, on peut se demander pourquoi elle n'est pas plus souvent utilisée. La première raison est la complexité du modèle mais la principale raison, déjà mentionnée, est, qu'en grande majorité, les macros sont simples et n'ont que faire de ce surplus de puissance qui, en fait, ralentit la macroexpansion. En effet, une des propriétés intéressantes du modèle classique est que l'on ne revient jamais sur les expressions complètement expansées (lorsque le macroexpansé ne débute pas par un mot-clé de macro, alors c'est une application fonctionnelle qui ne sera jamais plus révisée ; la macroexpansion procède en une seule passe linéaire), alors que le résultat d'une expansion en style EPS peut toujours être repris par un expanseur englobant favorisant ainsi l'occurrence de ré-expansions superflues.

9.5 Acceptabilité du macroexpansé

Le résultat de la macroexpansion doit être, par contrat, un programme propre à être préparé. Il est réputé ne plus contenir d'abréviations, c'est-à-dire apparaître exactement comme il aurait pu être d'emblée écrit. Il y a donc quelques écueils à éviter dont le premier est que, la macroexpansion étant un calcul, ce calcul est susceptible de ne jamais se terminer, ce qui entraînerait *ipso facto* la non terminaison de la phase de préparation. Qu'un compilateur boucle, cela n'arrive pas souvent mais c'est le prix à payer pour disposer d'un puissant système de macros ne limitant pas les calculs que l'on peut y mener.

Une façon simple de boucler est de réintroduire l'expression à macroexpanser dans son macroexpansé. C'est ce qui arrive fort simplement avec la macro `while`, si on la définit comme :

LOOP
```
(define-abbreviation (while condition . body)
  `(if ,condition (begin (begin . ,body)
                         (while ,condition . ,body) )) )
```

La réintroduction du texte même que l'on est en train de macroexpanser dans le résultat de la macroexpansion (et si l'on est en modèle classique d'expansion) provoque une expansion (aux deux sens du terme) sans fin !

La même erreur peut être commise, de façon encore plus subreptice en COMMON LISP, grâce au mot-clef `&whole` qui permet de récupérer la forme d'appel même à la macro. Dans la définition suivante, le résultat de la macroexpansion contient l'expression originale.

COMMON LISP
```
(defmacro while (&whole call)
  (let ((condition (cadr call))
        (body      (cddr call)) )
    `(if ,condition (begin (begin . ,body) ,call)) ) )
```

Dans le temps, de nombreux interprètes macroexpansaient à la volée les Sexpressions qu'ils recevaient. Pour éviter d'expanser encore et encore les mêmes choses, ils

pouvaient adopter une technique de mémorisation (pour *memoization*) ou encore se replier sur la stratégie de *macro écrasante* (pour *displacing macro*), qui consistait à remplacer physiquement la Sexpression déclenchant la macroexpansion par son macroexpansé. Dans ce cas là, la précédente macro `while` aurait fabriqué une structure cyclique, ce qui ne pose pas de problème pour être interprété mais qui ferait boucler le compilateur qui ne s'attend pas à devoir traiter autre chose que des DAGs, c'est-à-dire des arbres avec d'éventuels partages de branches [Que92a]. On peut décrire cette variante en réécrivant la macro `while` comme :

COMMON LISP
```
(defmacro while (&whole call)
  (let ((condition (cadr call))
        (body      (cddr call)) )
    (setf (car call) 'if)
    (setf (cdr call) `(,condition (begin (begin . ,body) ,call)))
    call ) )
```

Ce que nous venons de montrer sur des programmes peut aussi survenir sur les citations qui peuvent elles aussi être cycliques et faire boucler certains compilateurs. **[☞ p. 133]**

Le second écueil, bien plus pernicieux que le premier, est que le macroexpansé peut contenir des valeurs intruses provenant de la macroexpansion. Le contrat moral du programmeur est que le résultat de la macroexpansion doit être un programme qu'il aurait pu écrire directement, ce qui implique que ce programme a nécessairement une forme écrivable. C'est ainsi que Scheme insiste sur le fait que les citations doivent être formulées avec une syntaxe externe afin de prohiber la citation de valeurs quelconques.

Donnons un exemple particulièrement tordu d'une citation comportant une valeur non écrivable : c'est ce que fabrique la macro suivante, par insertion d'une continuation dans la valeur citée.

BAD TASTE
```
(define-abbreviation (incredible x)
  (call/cc (lambda (k) `(quote (,k ,x)))) )
```

Quel sens accorder à ce galimatias ? Considérons quelques-unes des hypothèses précédemment mentionnées. L'écriture du macroexpansé dans un fichier semble compromise, il n'y a pas de graphie standard pour les continuations ! Que peut signifier l'invocation d'une continuation prise dans un processus différent ou, en termes C, l'invocation pendant l'exécution d'un `a.out` d'une continuation capturée par `cpp` ? En l'absence de consensus sur ce que cela signifie, il faut juste dire « non » !

L'exemple ci-dessus avait utilisé une continuation pour bien marquer l'étrangeté de la situation. On évitera de même d'inclure toute valeur qui n'a pas de représentation écrite, aussi n'incluera-t-on pas de primitives, de fermetures ou de flux d'entrée/sortie : on n'écrira donc pas plus `(',(lambda (y) car) x)` que `` `(',car x) `` ou que `(f ',(current-output-port))`, même, si dans un monde unique et interprété, cela peut ne pas être erroné.

En résumé, on se répétera *in petto* la règle d'or des macros : point n'engendrera de programme que tu ne puisses directement écrire !

9.6 Définition des macros

Deux formes permettent de déclarer des macros globales (avec une portée qui sera analysée dans une prochaine section) ou locales. Ce sont `define-syntax`, `let-rec-syntax` et `let-syntax` en Scheme, `defmacro` et `macrolet` en COMMON LISP, `define-abbreviation` dans ce livre.

Définir une macro passe par la construction d'une fonction : l'expanseur, puis par l'enregistrement de cet expanseur sous un nom approprié. L'expanseur est écrit dans le langage des macros, qui est une instance de Lisp. Nous allons analyser plus avant ces conséquences dans les différents mondes et modes auparavant mentionnés.

9.6.1 Mondes multiples

Rappelons qu'avec des mondes multiples, la macroexpansion est un processus se déroulant dans un espace mémoire séparé de l'évaluation finale.

Mode endogène

Dans l'école endogène, le texte définissant la macro (la forme `define-abbreviation`) permet de synthétiser à la volée un expanseur grâce à l'évaluateur implantant le langage des macros. Dans ce cas, le mot-clé `define-abbreviation` peut ne rien être d'autre qu'un marqueur syntaxique que recherche le macroexpanseur. La définition suivante présente un macroexpanseur très naïf implantant cette stratégie endogène.

```
(define *macros* '())
(define (install-macro! name expander)
  (set! *macros* (cons (cons name expander) *macros*)) )
(define (naive-endogeneous-macroexpander exps)
  (define (macro-definition? exp)
    (and (pair? exp)
         (eq? (car exp) 'define-abbreviation) ) )
  (if (pair? exps)
      (if (macro-definition? (car exps))
          (let* ((def       (car exps))
                 (name      (car (cadr def)))
                 (variables (cdr (cadr def)))
                 (body      (cddr def)) )
            (install-macro! name (macro-eval
                                  `(lambda ,variables . ,body) ))
            (naive-endogeneous-macroexpander (cdr exps)) )
          (let ((exp (expand-expression (car exps) *macros*)))
            (cons exp (naive-endogeneous-macroexpander
                       (cdr exps) )) ) )
      '() ) )
```

Ce macroexpanseur prend une séquence d'expressions (la suite des expressions d'un fichier à compiler) et consulte la variable globale `*macros*` qui contient (sous la forme d'une liste d'association) les macros courantes. Les expressions sont tour à tour expansées (par la sous-fonction `expand-expression`) avec les macros courantes. Lorsqu'une définition de macro est rencontrée, elle est traitée spécialement, à la volée, comme une évaluation sertie au sein de la macroexpansion. La fonction d'évaluation est représentée par `macro-eval` qui peut être différente de l'`eval` natif : `macro-eval`

implante le langage des macros alors qu'`eval` implante le langage vers lequel on expanse. Cela ressemble à de la compilation croisée. Lorsque l'expanseur est construit, il est inséré par `install-macro!` dans la liste des macros courantes. On conçoit bien la différence de monde qui existe entre les définitions de macros et les autres expressions qui ne sont qu'expansées. C'est ce qu'illustre l'exemple correct suivant :

si/chap9b.scm

```
(define (fact1 n) (if (= n 0) 1 (* n (fact1 (- n 1)))))
(define-abbreviation (factorial n)
  (define (fact2 n) (if (= n 1) 1 (* n (fact2 (- n 1)))))
  (if (and (integer? n) (> n 0)) (fact2 n) `(fact1 ,n)) )
(define (some-facts)
  (list (factorial 5) (factorial (+ 3 2))) )
```

Il ne serait pas sain de confondre, en mode endogène, `fact1` et `fact2`, car la définition de `fact1` n'est qu'expansée et aucunement évaluée : elle n'existe donc pas encore lorsque la macro `factorial` est définie. C'est pourquoi, cette macro utilise pour ses besoins propres sa propre version : `fact2`. L'expansion des trois expressions données ci-dessus conduit donc à :

si/chap9b.escm

```
(define (fact1 n) (if (= n 0) 1 (* n (fact1 (- n 1)))))
(define (some-facts) (list 120 (fact1 (+ 3 2))))
```

On pourrait rendre l'exemple plus complexe en renommant `fact1` et `fact2` simplement `fact` : il faudrait alors mentalement comprendre à quoi se réfèrent les différentes occurences du mot `fact` au sein de la définition de `factorial` et ainsi déterminer que la première occurrence se rapporte à la valeur de la variable `fact` locale à la macro (au moment de l'expansion) tandis que la seconde se réfère à la variable globale `fact` (au moment de l'exécution).

Il existe cependant des variantes au sein du mode endogène. Certains macroexpanseurs effectuent une première passe extrayant de la suite d'expressions à expanser, les définitions de macros. Celles-ci sont alors définies et servent à expanser le reste du module. Ce sous-mode pose des problèmes lorsque des appels à des macros définissent de nouvelles macros car le macroexpanseur risque de ne point les voir. De plus, le fait que des macros puissent s'appliquer avant même leur définition est assez déroutant. Là encore, un ordre gauche-droite semble mentalement le plus adapté.

Une autre variante, importante et que l'on retrouvera plus tard, est de faire de `define-abbreviation` une macro prédéfinie.

Mode exogène

Dans le mode exogène, les macros ne sont plus définies dans les expressions à préparer mais dans les expressions qui définissent le macroexpanseur. On supposera que ce dernier est modulaire, c'est-à-dire que l'on peut définir de façon séparée des macros indépendantes. La meilleure façon de définir ces macros est, bien sûr, d'avoir

une macro pour ce faire et l'on emploiera le mot-clé `define-abbreviation`, dont voici la définition méta-circulaire :

```
(define-abbreviation (define-abbreviation call . body)
  `(install-macro! ',(car call) (lambda ,(cdr call) . ,body)) )
```

Son rôle est, lorsque le macroexpanseur sera spécifié par une directive appropriée, de lui adjoindre la macro présentement définie. On supposera donc que les directives de macroexpansion spécifient un moteur d'expansion qu'enrichiront des appels à `install-macro!`. La maîtrise de la macroexpansion est alors complètement différente puisque les macros sont définies dans des modules et utilisées dans d'autres modules. Supposons disposer d'un premier module dont le contenu soit :

si/chap9c.scm

```
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))
(define-abbreviation (factorial n)
  (if (and (integer? n) (> n 0)) (fact n) `(fact ,n)) )
```

L'expansion de ce module sera :

si/chap9c.escm

```
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))
(install-macro! 'factorial
   (lambda (n) (if (and (integer? n) (> n 0)) (fact n) `(fact ,n))) )
```

Si maintenant l'on souhaite utiliser la macro `factorial` pour la macroexpansion d'un autre module contenant la définition de la fonction `some-facts`, module que voici :

si/chap9d.scm

```
(define (some-facts)
  (list (factorial 5) (factorial (+ 3 2))) )
```

L'on obtiendra comme résultat :

si/chap9d.escm

```
(define (some-facts)
  (list 120 (fact (+ 3 2))) )
```

La première occurrence de `fact` dans la macro `factorial` ne pose pas de problème : c'est la référence à la variable globale `fact` du même module `si/chap9c.scm`, ce qui est évident si l'on contemple son macroexpansé `si/chap9c.escm`. En revanche, la seconde occurrence de `fact` référence la variable `fact` telle qu'elle existera à l'exécution du module `si/chap9d.escm` : cette référence est libre dans le macroexpansé et il se peut très bien qu'il n'y ait point de variable `fact` à l'exécution !

La solution de facilité est d'adjoindre à `si/chap9d.scm` (par édition de liens ou copier/coller) un module parmi ceux que l'on a sous la main : `si/chap9c.scm`, ce qui définit une variable globale `fact` répondant au besoin. Mais, à ce moment, vous

aurez aussi importé la définition d'une macro, `factorial`, qui se trouvait dans le même module, dont vous n'avez que faire pour l'évaluation de `some-facts` et qui peut même introduire de nouvelles erreurs puisqu'elle invoque la fonction `install-macro!` qui n'a aucune raison d'exister dans l'application engendrée.

La seconde solution est de mieux comprendre les dépendances qui existent entre les macros et leurs bibliothèques d'exécution et surtout, *surtout*, les différents moments de calcul. L'art des macros est complexe car il requiert la maîtrise du temps. Reprenons l'ensemble `fact1`, `fact2` et `factorial`. Lorsqu'un site d'emploi de la macro `factorial` est découvert, la macro a besoin, pour s'expanser, de la fonction `fact2`. Par contre, son résultat, l'expression macroexpansée, a besoin, pour s'exécuter, de la fonction `fact1`. On dira que `fact1` appartient à la *bibliothèque d'exécution* de la macro `factorial` tandis que `fact2` appartient à la *bibliothèque d'expansion* de `factorial`. Les « durées de vie » de ces deux bibliothèques n'ont rien à voir : la bibliothèque d'expansion n'est requise que pendant la macroexpansion tandis que la bibliothèque d'exécution n'est utile que pour l'évaluation du macroexpansé.

Une solution est alors de définir, en mode exogène, les macros et leur bibliothèque d'expansion dans un module `si/bibexp` puis, **[☞ Ex. 9.5]** dans un module séparé leur bibliothèque d'exécution `si/bibrun`. Toujours pour la factorielle, l'on écrira :

si/bibexp.scm

```
(define (fact2 n) (if (= n 0) 1 (* n (fact2 (- n 1)))))
(define-abbreviation (factorial n)
  (if (and (integer? n) (> n 0)) (fact2 n) `(fact1 ,n)) )
```

et dans cet autre module :

si/bibrun.scm

```
(define (fact1 n) (if (= n 0) 1 (* n (fact1 (- n 1)))))
```

Lorsqu'une directive mentionnera l'utilisation de la macro `factorial`, le macroexpanseur chargera le module `si/bibexp.scm` et le langage des directives enregistrera qu'à l'exécution, la bibliothèque `si/bibrun.scm` devra être liée à l'application ; ainsi pourra-t-on très exactement ne retenir à l'exécution que les seules ressources nécessaires [DPS94b]. La figure 9.3 récapitule ces opérations : elle montre sous une forme de jeu de « cubes » comment, dans un premier temps, l'on construit le compilateur adapté à la compilation du programme et comment l'on obtient le petit exécutable autonome auquel on est en droit de s'attendre.

9.6.2 Monde unique

On pourrait croire, après la section précédente, que l'hypothèse du monde unique règle tous les problèmes : il n'en est rien !

Si l'on est dans un monde unique, l'évaluateur lit des expressions, les macroexpanse, prépare le macroexpansé enfin, évalue le résultat de la préparation. L'état du macroexpanseur est donc inclus dans l'état de la boucle d'interaction. Dans cette hypothèse, il est logique que l'évaluateur lié à la macroexpansion endogène (nommé

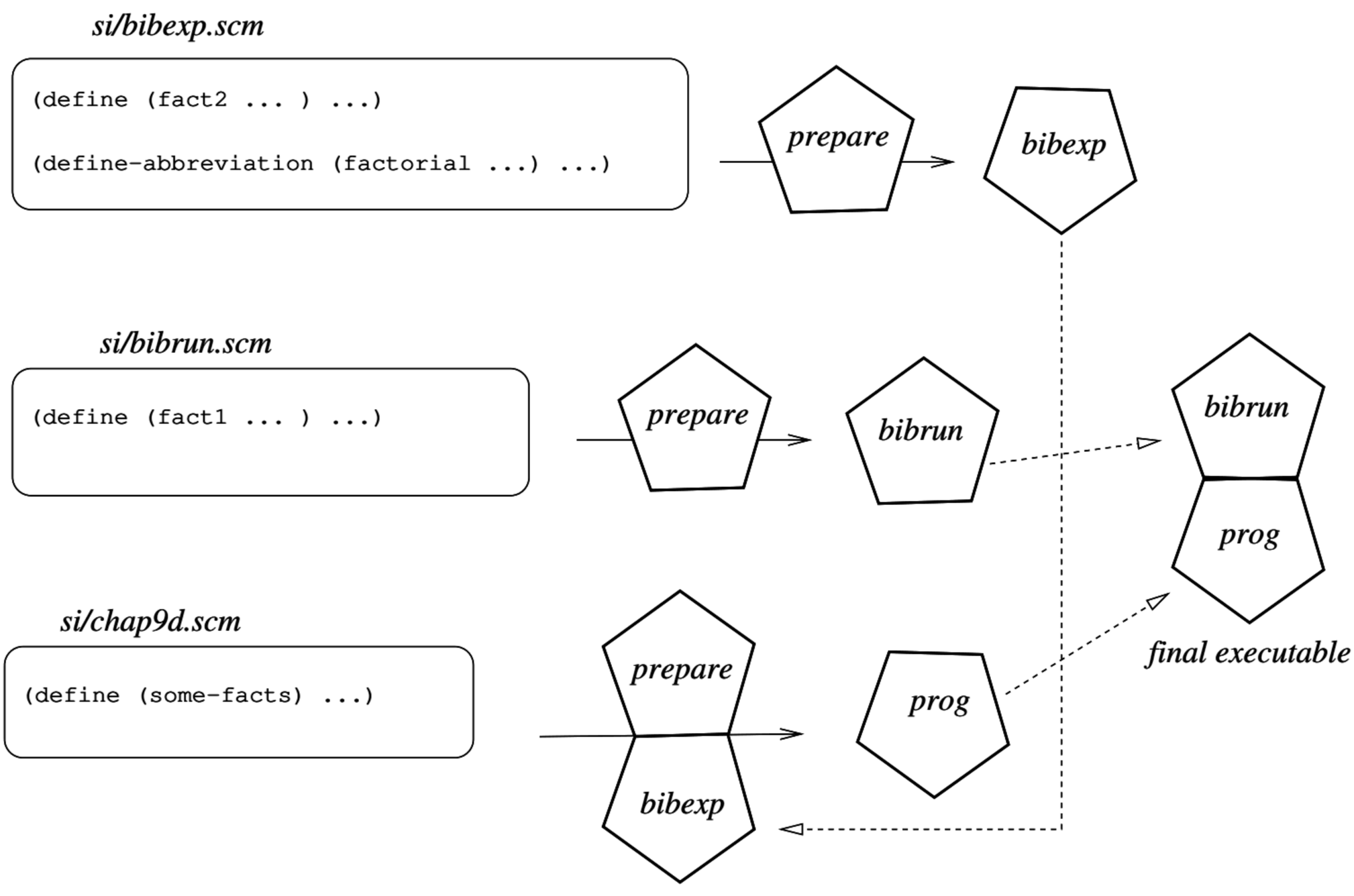


FIGURE 9.3 – Mondes multiples : construction d'une application

ci-dessus `macro-eval`) soit tout simplement la fonction `eval` de l'évaluateur (mais avec toutes les variantes possibles vues au chapitre 8). **[☞ p. 257]** Comme le monde est unique, la macroexpansion a accès (en lecture ou écriture) à tout ce qui est présent dans ce monde. Il peut sembler alors aisé de ne point faire de différence entre bibliothèque d'expansion et bibliothèque d'exécution puisqu'il suffit de soumettre à la boucle d'interaction :

```
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))
(define-abbreviation (factorial n)
  (if (and (integer? n) (> n 0)) (fact n) `(fact ,n)) )
```

Le problème est que maintenant on rend le partage (ou la vente) de logiciels très difficile car il est malaisé et souvent impossible d'extraire ce qui est seulement nécessaire pour exécuter ou regénérer le programme que l'on souhaite partager. Pour être sûr de ne rien oublier, on peut livrer son image mémoire entière (mais c'est cher payé). On peut essayer de recourir à un élagueur (pour *tree-shaker*) cherchant à éliminer (de façon saine) ce que l'on peut prouver inutile : c'est une sorte de GC dont l'efficacité n'est pas grandiose. Enfin et c'est visible sur nombre de programmes Lisp ou Scheme présents sur Internet, on peut s'abstenir de livrer des programmes contenant des macros : on les livre tout expansés (ce sont les *macroless programs*) ou avec seulement des macros locales. Et l'on a ainsi renié le pouvoir expressif que les macros apportent puisqu'on a livré quelque chose d'exécutable mais plus humainement lisible.

Si l'on ne croit pas aux outils automatiques d'extraction, il ne reste plus qu'à faire la part soi-même et distinguer les ressources seulement utiles à la macroexpansion, celles seulement utiles à l'exécution et enfin, les plus problématiques, celles qui sont

utiles aux deux phases.

Afin de préparer de façon asynchrone des programmes, existe la fonction `compile-file`. Deux variantes s'opposent au sein du monde central, dont l'une pourrait être qualifiée de *monde uniquement unique*. La distinction concerne les macros disponibles pour la préparation du module soumis à `compile-file` ; les trois solutions ci-dessous présentées se rencontrent dans la nature ...

1. Dans le monde uniquement unique, il n'y a qu'un seul espace de macros et il est valable partout. Les macros courantes sont donc les macros que peuvent utiliser les expressions du module en cours de préparation. C'est le cas de la figure 9.4 : la macro `factorial` définie dans la boucle d'interaction est bien perçue par le module compilé. Sous-question, que se passe-t-il si le module définit lui-même des macros ?

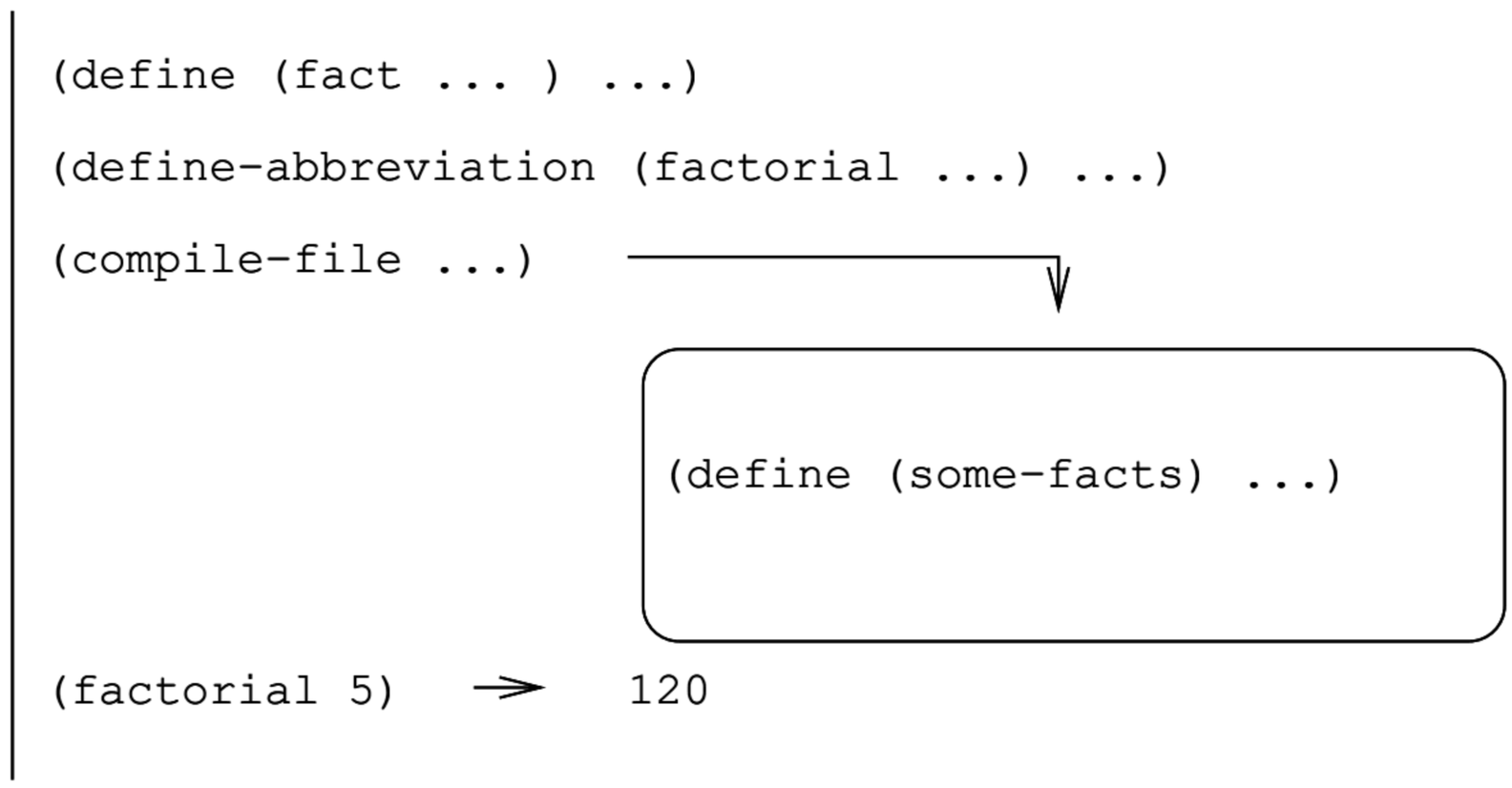


FIGURE 9.4 – Monde uniquement unique

   (a) Dans le monde uniquement unique, ces nouvelles macros sont ajoutées aux macros courantes, l'espace des macros ne cesse donc d'augmenter (à moins que ne soit permis de supprimer des macros). Ce sera le cas dans la figure 9.5 si l'expression `(factorial 5)` soumise à la boucle d'interaction retourne 120.

   (b) Une seconde réponse serait que les macros définies dans le module ne sont perçues que par la préparation du dit module. On distingue donc les macros *super-globales* des macros *seulement globales* aux modules. Dans la figure 9.5, ce sera le cas si l'expression `(factorial 5)` soumise à la boucle d'interaction est erronée.

2. Enfin si le monde n'est pas uniquement unique, alors les seules macros perçues par le module sont celles que le module définit de façon endogène. Dans ce cas, il y a l'espace des macros de la boucle d'interaction et les espaces de macros, initialement vides, propres à chaque module préparé. Mais bien que ces espaces de macros soient séparés, la macroexpansion se passe quand même dans

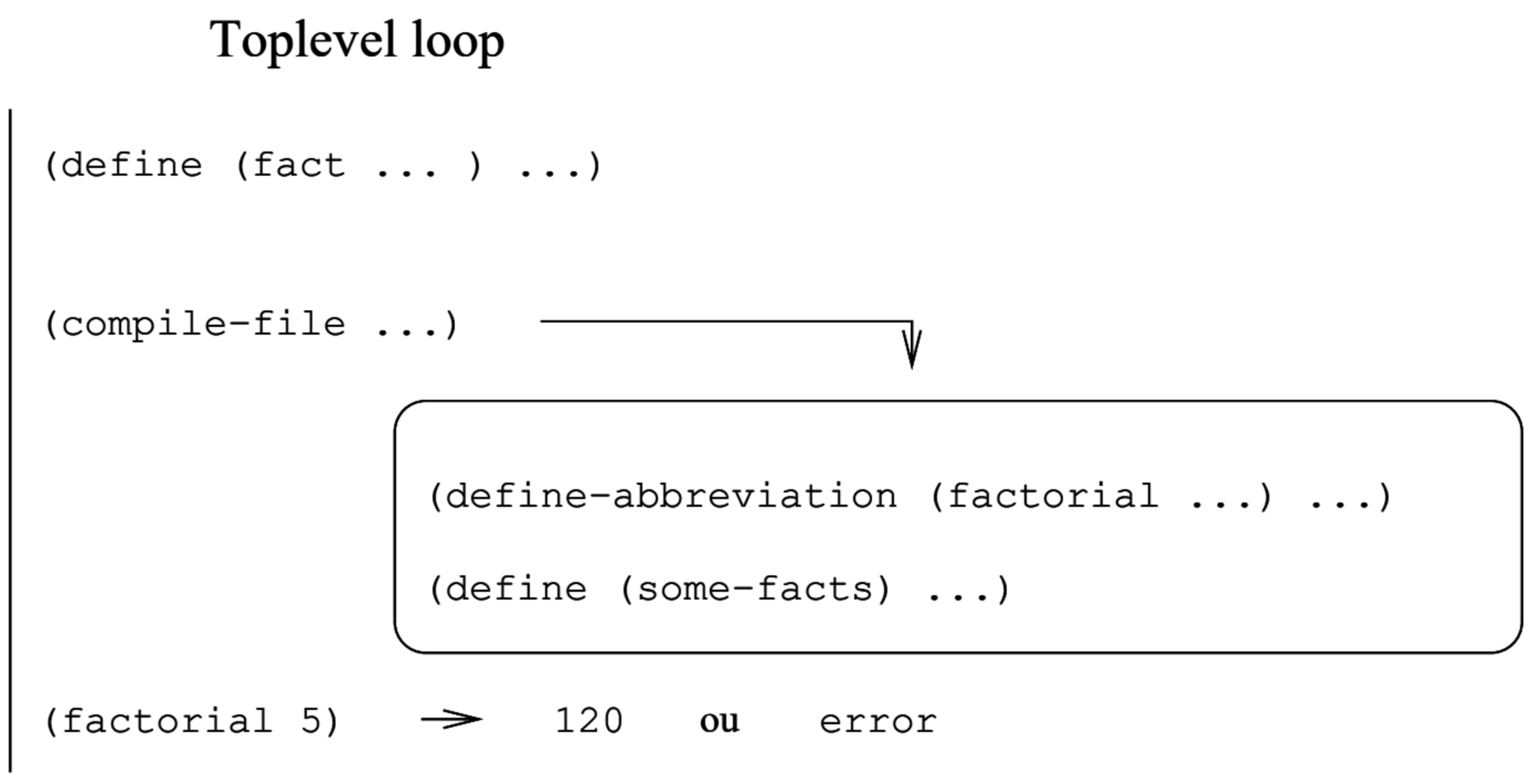


FIGURE 9.5 – Monde unique à compilation endogène

un monde unique et peut donc profiter de toutes les resources globales (voir à la figure 9.5, la macro `factorial` locale au module compilé peut employer la fonction `fact` de la boucle d'interaction).

Un examen attentif des figures 9.4 et 9.5 nous révèle que, dans tous les cas, la fonction `fact` appartenant à la bibliothèque d'expansion de `factorial` avait été définie au niveau de la boucle d'interaction. Ce faisant, elle est globalement visible alors qu'en fait elle n'est probablement utile qu'à la macro `factorial`. On pourrait la mettre en fonction interne à la macro (comme c'était le cas dans `si/chap9b.scm`). **[☞ p. 307]**

Que se passerait-il si `fact` était utile à plus d'une macro ? Le problème est que le macroexpanseur ne sait rien reconnaître d'autre que des définitions de macros alors que souvent l'on aimerait définir des variables globales, des fonctions utilitaires ou des données diverses pour l'usage exclusif d'un jeu de plusieurs macros. Si l'hypothèse des mondes multiples en mode exogène assure naturellement cela, ce n'est pas le cas du mode endogène ou du monde unique ; aussi les tenants de ces hypothèses ont-ils introduit la possibilité de forcer une évaluation au sein du macroexpanseur. Ce trait a pour nom `eval-when` en COMMON LISP et autres versions de Scheme. Pour ne point interférer, nous le nommerons, pour notre part, `eval-in-abbreviation-world`.

Le but de la forme `eval-in-abbreviation-world` est de permettre de définir des constantes, des fonctions utilitaires, de dérouler des calculs quelconques au sein et pour le seul usage du macroexpanseur. C'est en fait reconnaître explicitement le besoin de séparer ce monde du monde ultérieur de l'évaluation. Dans la figure 9.6, la fonction `fact` est définie dans l'évaluateur du macroexpanseur et `factorial` peut alors l'employer pour expanser `some-facts`. De retour à la boucle d'interaction fondamentale et suivant que les mondes ont des « fuites » ou pas, `factorial` et/ou `fact` peuvent être encore visibles et invocables.

Pour programmer ce comportement, il suffit de demander au macroexpanseur de reconnaître les formes débutant par le mot-clé `eval-in-abbreviation-world` et de les évaluer à la volée avec l'évaluateur approprié, c'est-à-dire celui que nous avions

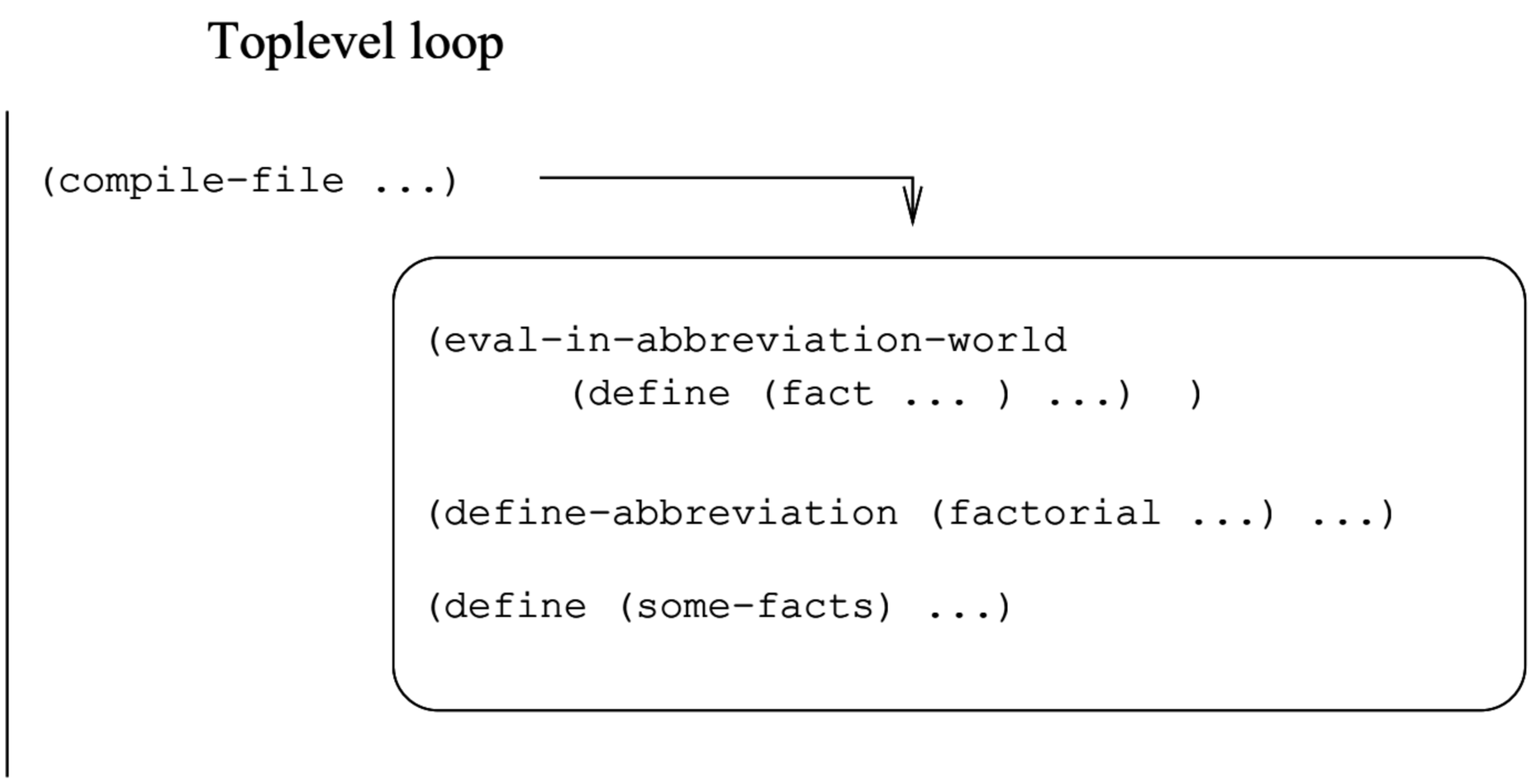


FIGURE 9.6 – Évaluation endogène

nommé `macro-eval` **dans la fonction** `naive-endogeneous-macroexpander`.

```
(define (expand-expression exp macroenv)
  (define (evaluation? exp)
    (and (pair? exp)
         (eq? (car exp) 'eval-in-abbreviation-world) ) )
  (define (macro-call? exp)
    (and (pair? exp) (find-expander (car exp) macroenv)) )
  (define (expand exp)
    (cond ((evaluation? exp) (macro-eval `(begin . ,(cdr exp))))
          ((macro-call? exp) (expand-macro-call exp macroenv))
          ((pair? exp)
           (let ((newcar (expand (car exp))))
             (cons newcar (expand (cdr exp))) ) )
          (else exp) ) )
  (expand exp) )
```

La fonction `macro-eval` est l'évaluateur tapi derrière le mécanisme de macroexpansion. Il peut être vu comme une instance de la fonction usuelle `eval`, mais pourvue d'un environnement global propre, non partagé avec l'évaluateur dont use la boucle d'interaction. Dans ce cas et pour la figure 9.6, `(factorial 5)` ou `(fact 5)` seraient toutes deux erronées.

Les formes apparaissant sous `eval-in-abbreviation-world` doivent être considérées comme des directives exécutables incluses dans un magma en cours d'expansion. En particulier, ces formes ne peuvent utiliser ce qui sera plus tard l'environnement lexical courant. On ne peut écrire :

```
(let ((x 33))                                                    WRONG
  (eval-in-abbreviation-world (display x)) )
```

Encore que si l'on souhaite introduire localement une variable pour le seul bénéfice du macroexpanseur (ce que `eval-in-abbreviation-world` ne sait point faire puisque, fondamentalement, ce n'est qu'un `eval` à *toplevel*), on puisse procurer, comme en

COMMON LISP I [Ste84] mais pas COMMON LISP II [Ste90], la forme spéciale `compiler-let` qu'il suffit de faire reconnaître par la fonction `expand-expression`.

Si l'expansion ne peut atteindre l'environnement d'exécution final, réciproquement ce dernier ne peut atteindre l'environnement d'expansion. On ne peut donc (heureusement) toujours pas écrire :

```
(define-abbreviation (foo ...) ...)
(apply foo ...)                                                    WRONG
```

Muni du mécanisme de `eval-in-abbreviation-world`, il est simple de réexpliquer ce qu'est `define-abbreviation` : c'est elle-même une macro :

```
(define-abbreviation (define-abbreviation call . body)
  `(eval-in-abbreviation-world
     (install-macro! ',(car call) (lambda ,(cdr call) . ,body))
     #t ) )
```

L'intérêt est que maintenant l'on peut, partout, placer des définitions de macros et plus seulement en position proéminente (pour *toplevel form*). Ainsi si l'on veut rassembler plusieurs expressions en une unique séquence, on pourra écrire en une seule fois :

```
(begin (define-abbreviation (foo x) ...)
       (bar (foo 34)) )
```

Ceci ne doit cependant pas, c'est une question de bon goût, trop préjuger de l'ordre de macroexpansion et conduire à s'autoriser l'écriture très relâchée :

```
(if (bar) (begin (define-abbreviation (foo x) ...)                 TASTELESS
                 (hux) )
          (foo 35) )
```

La plupart des macroexpanseurs distinguent les expressions en position proéminentes des autres. C'est, par exemple, ce qui distingue les `define` internes ou globaux. Les macroexpanseurs assurent aussi que les expressions proéminentes soient traitées séquentiellement de la gauche vers la droite, ce qui permet, par exemple, de définir dans l'ordre une classe puis ses sous-classes.

En fait, les précédents exemples ont caché quelques détails d'implantation dus au fait que plusieurs évaluateurs coexistent et que les définitions précédentes ne sont pas toutes dirigées vers les mêmes évaluateurs. De la même manière qu'avec les interprètes réflexifs, se pose le problème de l'accès à la structure de données définissant les macros courantes. **[☞ p. 257]** La forme `install-macro!` présente dans la définition de `define-abbreviation` sera évaluée par `macro-eval` mais doit modifier l'ensemble des macros courantes tel que scruté par la fonction `find-expander`, qui n'est pas évaluée par `macro-eval` mais par `eval` ! Ces problèmes ne sont point importants pour l'instant.

L'un des intérêts d'`eval-in-abbreviation-world` est d'introduire des définitions de variables globales dans le macroexpanseur car, souvent, les macros ont un comportement dépendant de l'implantation sous-jacente. Par exemple, une macro construisant dans son macroexpansé un appel à la fonction `apply` peut inspecter son environnement pour savoir si la fonction `apply` qui s'y trouve est binaire (comme suffisant pour le R^{2}RS) ou n-aire (comme en R^{4}RS). Mais inspecter son environnement signifie que le macroexpansé a pour cible ce même environnement. Dans le cas de logiciels complexes (et l'on détaillera le cas de MEROON par la suite), il est parfois utile d'expanser pour une autre implantation, auquel cas il est nécessaire que l'implantation cible soit

définie par des caractéristiques (pour *features*) inspectables (voir notamment les SRFI *Scheme Request for Implementation* et le SRFI 0 qui normalise cette notion). On peut donc observer des macros comme :

```
(define-abbreviation (generate-foo x y z)
  (if (memq 'binary-apply *features*)
      `(apply foo (cons x (cons y z)))
      `(apply foo x y z) ) )
```

La variable `*features*` doit être visible du macroexpanseur, ce qui requiert de l'avoir définie auparavant grâce à `eval-in-abbreviation-world`. On pourra donc caractériser une implantation en définissant la variable `*features*` que pourront consulter les macros concernées. Dans l'expression qui suit et bien que l'apparence soit trompeuse, la forme `define` est une forme proéminente du langage des macros, c'est donc une définition de la variable globale `*features*` :

```
(eval-in-abbreviation-world
  (define *features* '(31bit-fixnum binary-apply)) )
```

9.6.3 Évaluation simultanée

Une variante importante, souvent présente dans l'hypothèse du monde unique, est l'évaluation simultanée à la préparation. Au fur et à mesure que les expressions sont expansées, elles sont evaluées par la boucle d'interaction. C'est ce que suggère la définition suivante :

```
(define (simultaneous-eval-macroexpander exps)
(define (macro-definition? exp)
  (and (pair? exp)
       (eq? (car exp) 'define-abbreviation) ) )
(if (pair? exps)
    (if (macro-definition? (car exps))
        (let* ((def       (car exps))
               (name      (car (cadr def)))
               (variables (cdr (cadr def)))
               (body      (cddr def)) )
          (install-macro!
           name (macro-eval `(lambda ,variables . ,body)) )
          (simultaneous-eval-macroexpander (cdr exps)) )
        (let ((e (expand-expression (car exps) *macros*)))
          (eval e)
          (cons e (simultaneous-eval-macroexpander (cdr exps))) ) )
    '() ) )
```

Remarquez que deux évaluateurs apparaissent dans cette définition — `eval`, l'évaluateur de la boucle d'interaction et — `macro-eval`, l'évaluateur du macroexpanseur. Nous les avons distingués car ils représentent des processus distincts intervenant à des moments distincts.

9.6.4 Redéfinition de macros

La fonction `install-macro!` le sous-entendait : si l'on redéfinit une macro, on modifie l'état du macroexpanseur, ce qui altère les traitements qui lui restent à effectuer. Il est certain que, sous interprète et pour mettre au point des macros, il est assez sympathique, lorsqu'une fonction est définie et contient un site d'appel à une macro,

de redéfinir cette macro jusqu'à la mettre au point via cette fonction. Cela signifie que l'interprète diffère le plus possible l'expansion des macros. Mais, la plupart du temps et pour assurer des résultats répétables, les expressions ne sont expansées qu'une seule fois : elles deviennent donc indépendantes des éventuelles redéfinitions des macros. Les macros ont un comportement que l'on peut simplement qualifier d'hyperstatique. **[☞ p. 113]**

9.6.5 Comparaisons

Un des grands problèmes relatifs aux macros est que, si l'on n'est pas satisfait du système de macros que procure l'implantation, il est difficile d'en changer et cela a probablement nui à l'expérimentation débridée dont ont profité d'autres domaines de Lisp/Scheme.

Le système des mondes multiples en mode exogène semble le plus précis mais le moins répandu. Le système des mondes multiples en mode endogène n'est qu'une instance du mode exogène avec un moteur à expanser prédéfinissant certaines macros comme `define-abbreviation` et les autres. Enfin le système du monde unique est le plus répandu mais le moins maîtrisable en terme de livraison de logiciels. Cette comparaison est un peu rapide car elle ne tient pas compte de deux nouvelles facettes que nous allons maintenant discuter : la compilation des macros et l'existence de macros définissant des macros.

Compilation des macros

Dans ce que nous avons précédemment exposé, la macroexpansion d'un appel à une macro est le plus souvent le fait d'un interprète. Ainsi l'efficacité semble compromise, surtout si la transformation de programme que réalise la macro est longue et complexe (mais c'est rare). En mondes multiples et mode exogène, **[☞ p. 299]** on construit le compilateur ad hoc par assemblage de modules préparés (compilés) aussi l'efficacité est-elle présente. En mode endogène, une solution est d'utiliser une fonction `macro-eval` compilante. En monde unique, on peut employer `compile-file` et `load`. Le problème est comment compiler une définition de macro ?

La question n'est aucunement oiseuse ! Les macroexpanseurs que nous avons présentés scrutaient les expressions à expanser pour y trouver des formes `define-abbreviation` et définir à la volée ces mêmes macros. Il faut donc distinguer la compilation de la définition de la macro et son installation. Si `define-abbreviation` est une macro s'expansant en une forme `install-macro!`, alors c'est gagné ! Si, par contre, `define-abbreviation` est un mot-clé syntaxique, il ne reste plus qu'à assurer que le corps des macros n'est qu'un appel interprété à une fonction compilée. Une fois compilé, il faut encore charger le programme préparé et là on se pose encore de multiples problèmes comme ce que peut bien signifier `(eval-in-abbreviation-world (load file))`. La fonction `load` n'est qu'un enrobage autour de `eval` mais dans le monde de la macroexpansion, c'est `macro-eval` qu'il faut utiliser et non `eval` ! Sans parler d'appel explicite à `eval` comme dans `(eval-in-abbreviation-world (eval '(define-abbreviation ...)))`, où, là encore, `eval` doit référencer `macro-eval` et non l'évaluateur de la boucle d'interaction. Cette discussion montre bien que les deux

environnements globaux propres à `eval` et `macro-eval` sont bien séparés et procurent, sous de mêmes noms, des entités différentes.

On retiendra que le définisseur de macro gagne à être lui-même une macro et non un marqueur syntaxique que recherche le macroexpanseur. Ainsi on peut séparer la préparation de la macro de son installation. Ce point sera repris par la suite. **[☞ p. 320]**

Macros définissant des macros

Il est de temps en temps utile de définir des macros qui engendrent elles-mêmes des macros. **[☞ p. 323]** Ce n'est pas là un aspect étrange mais seulement l'application logique des principes d'abstraction et de liberté. Par exemple, dans un système d'objets, on peut souhaiter en définissant la classe `Point` que les accesseurs `Point-x` et `Point-y` ne soient pas des fonctions mais des macros. **[☞ p. 413]** Dans ce cas, la macro `define-class` doit engendrer ces nouvelles macros.

Prenons l'exemple d'une autre macro `define-alias` qui souhaite définir son premier argument à l'égal de son second. Nous la donnons en deux exemplaires suivant que l'on aime ou pas enchâsser les *backquotes* dans les *backquotes*.

```
(define-abbreviation (define-alias newname oldname)
  `(define-abbreviation (,newname . parameters)
     `(,',oldname . ,parameters) ) )
(define-abbreviation (define-alias newname oldname)
  `(define-abbreviation (,newname . parameters)
     (cons ',oldname parameters) ) )
```

En mode classique, où le résultat d'une macroexpansion est réexpansé, si `define-abbreviation` est une macro, il n'y a pas de problème. En revanche, si `define-abbreviation` n'est qu'un marqueur syntaxique, le macroexpanseur peut très bien ne pas s'apercevoir de cette définition s'il n'analyse pas le résultat des macroexpansions. On abandonnera donc l'idée d'un marqueur syntaxique au bénéfice de macros prédéfinies et mêmes primitives car l'on ne saurait les recréer si elles venaient à manquer.

9.7 Portée des macros

La portée des macros locales, introduites par `let-syntax` ou `letrec-syntax` en Scheme, ne pose guère de problèmes. Ce n'est pas le cas des macros définies par `define-syntax` car l'on peut distinguer plusieurs cas d'emploi des macros. Nous prendrons pour discuter cette section, le cas de MEROON, un système d'objets construit au dessus de Scheme qui a été porté (compilé ou interprété) sur de nombreuses implantations de Scheme. Le système MEROON est décrit sous une forme essentielle nommée MEROONET au chapitre 11. Trois types de macros peuvent y être distinguées :

1. **Type 1 – macros ponctuelles** : Une macro ponctuelle est définie en un point et utilisée tout de suite après. On peut donc la transformer en une macro locale définie par une forme `let-syntax` ou `macrolet` si on en dispose, ce qui n'est pas le cas partout. Pour donner un exemple, la fonction `make-fix-maker` de MEROONET construit à la main un « triangle » de fermetures `(lambda (a b c`

`...) (vector cn a b c ...))`. **[☞ p. 412]** La macro suivante permet de le faire automatiquement :

```
(define-abbreviation (generate-vector-of-fix-makers n)
  (let* ((numbers (iota 0 n))
         (variables (map (lambda (i) (gensym)) numbers)) )
    `(case size
       ,@(map (lambda (i)
                (let ((vars (list-tail variables (- n i))))
                  `((,i) (lambda ,vars (vector cn . ,vars))) ) )
              numbers )
       (else #f) ) ) )
```

Une macro ponctuelle peut être abandonnée dès que consommée ; sa portée est très restreinte et ne dépasse pas les bornes du module où elle apparaît.

2. **Type 2 – macros locales à un système** : La définition de MEROON se décompose en quelques 25 petits fichiers. Un certain nombre d'abréviations sont utilisées tout au long de ces fichiers, par exemple la macro `when`. La portée de cette macro est donc l'ensemble des sources de MEROON mais pas plus car on n'a pas le droit de polluer l'extérieur de MEROON.
3. **Type 3 – macros exportées** : MEROONET exporte trois macros qui sont `define-class`, `define-generic` et `define-method`. Ces macros sont destinées aux utilisateurs de MEROON mais elles servent aussi à l'auto-amorçage même de MEROON. Notez que la macro `when` ne peut apparaître dans le macroexpansé des macros de type 3 car cela confèrerait à `when` une portée plus grande que prévue. Le langage du macroexpansé des macros exportées est celui de l'utilisateur et non celui de MEROON lui-même.

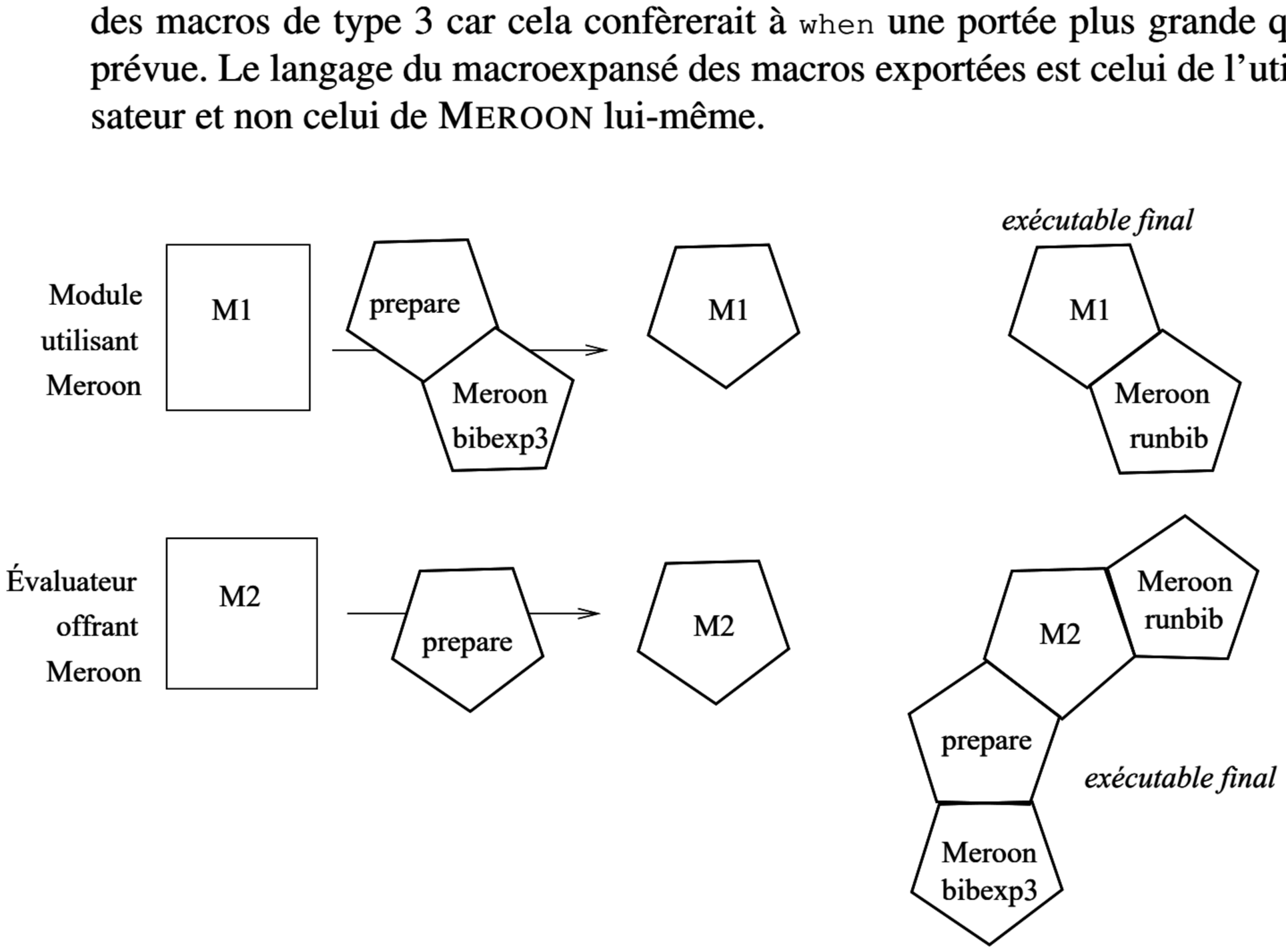


FIGURE 9.7 – Usage des macros

Non seulement trois sortes de macros ont pu être distinguées mais trois sortes

d'usage pour ces macros vont être répertoriés comme illustré à la figure 9.7. Ces trois usages sont :

1. **Préparer les sources de** MEROON : Il s'agit d'expanser les sources de MEROON et donc de faire disparaître tous les emplois de macros de type 1, 2 et 3. Par contre, quelque chose de la définition des macros de type 3 doit nécessairement demeurer puisqu'elles sont exportées.
2. **Préparer un module qui utilise** MEROON : Préparer un module qui utilise MEROON, c'est expanser les emplois des macros de type 3 de MEROON qu'il utilise. Il faut donc installer dans le préparateur de ce module, les macros de type 3 de MEROON, c'est-à-dire greffer au préparateur la bibliothèque d'expansion des macros de type 3 de MEROON. Notez que l'on n'a pas besoin de la bibliothèque d'expansion des macros de type 1 ou 2 de MEROON.
3. **Engendrer un interprète procurant** MEROON **à l'usager** : Dans ce dernier cas, on souhaite construire une boucle d'interaction procurant les macros de type 3 de MEROON, elles doivent donc être installées, accompagnées de leur bibliothèque d'expansion, dans le macroexpanseur lié à la boucle d'interaction.

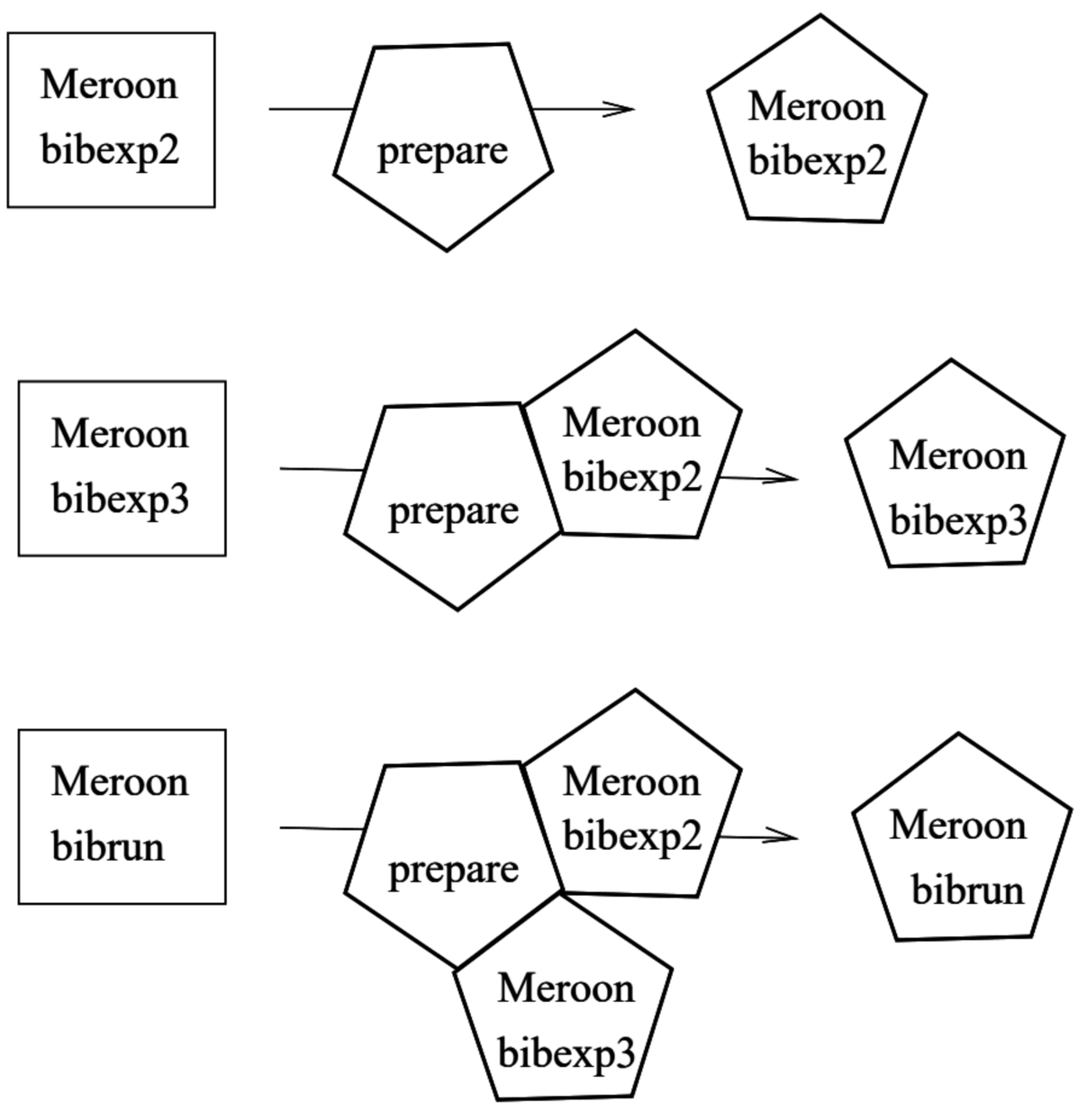


FIGURE 9.8 – Auto-amorçage de MEROON

La figure 9.8 illustre une façon d'obtenir les modules de MEROON par auto-amorçage. La bibliothèque d'expansion des macros de type 2 ne sert qu'à la seule préparation du reste des sources de MEROON. On le voit la construction d'un logiciel complexe ressemble à un diagramme de portage en T [ES70] où l'on manipule différents langages et leurs implantations.

Au regard des différents types et usages de macros que nous venons d'expliciter, on peut constater que les mondes multiples sont assez adaptés tandis que le monde

unique ne permettra généralement pas de limiter la portée des macros à leur seule zone d'emploi. Avec une notion de *package* permettant de structurer plus finement les noms utilisés (comme en COMMON LISP ou en Ilog-Talk), il suffit de donner aux macros d'usage restreint des noms internes, ce qui limitera les possibilités de collisions futures. En revanche, dans un monde unique, les trois usages de macros ci-dessus décrits sont tous confondus. Pour utiliser MEROON, on le charge et c'est tout !

Le chapitre 11 décrit MEROONET et, s'il procure trois macros, il ne les utilise pas pour ses besoins propres afin d'éviter des questions désagréables. Les macros qu'il définit (de type 3 suivant la précédente terminologie) le sont avec la macro `define-meroonet-macro` dont nous allons détailler l'implantation. La première question vis-à-vis d'un système de macro existant est de déterminer de quel type il est. Ensuite, il faut le mettre en branle et comme, en général, il n'est pas possible d'accéder à des fonctions telles que `install-macro!`, tout doit être fait avec le seul définisseur de macros de l'implantation visé. Cela va de la simple équivalence entre `define-meroonet-macro` et `define-abbreviation` jusqu'à la définition alambiquée suivante qui assure que la macro est à la fois utile tout de suite et le sera ultérieurement quand le fichier préparé où elle figure sera chargé dynamiquement.

```
(define-abbreviation (define-meroonet-macro call . body)
  `(begin (define-abbreviation ,call . ,body)
          (eval '(define-abbreviation ,call . ,body)) ) )
```

9.8 Évaluation et expansion

Macroexpansion et évaluation sont intimement liées dans la boucle d'interaction puisqu'immédiatement enchaînées. Cette section examine de plus près ce couplage.

La macroexpansion est la première phase de la préparation. L'évaluation est la phase qui suit la préparation. Si la fonction `eval` ne représente que l'évaluation alors le langage qu'accepte `eval` n'est que le langage pur dénué de toute macro. Il faut donc, si l'on désire profiter des macros, expanser soi-même les expressions que l'on fournira à `eval`. Ainsi dans un système d'évaluation multi-fenêtres pourra-t-on voir simultanément tourner dans des fenêtres différentes, deux macroexpanseurs différents. Mais ce n'est pas très pratique, peu conforme aux usages courants et, en plus, on y perdrait probablement les macros prédéfinies usuelles comme `cond`, `let` ou `case`. Ainsi donc, la fonction d'évaluation `eval` est-elle souvent fournie comme `(lambda (e) (pure-eval (macroexpand e *macros*)))` où `pure-eval` est l'évaluateur du langage pur tandis que `macroexpand` est la fonction publique de macroexpansion et que `*macros*` contient les macros connues de la boucle d'interaction.

Ce que nous venons de dire d'`eval` est bien sûr aussi vrai de `macro-eval` qui en est une autre instance dotée de sa propre variable `*macros*` contenant les seules macros connues de la préparation. De multiples questions se posent alors sur ce couplage entre macroexpansion et évaluation. Nous avions mentionné l'existence d'`eval/ce` pour *eval in current environment*, cette forme spéciale capturait tout le contexte lexical visible. **[☞ p. 263]** Si `eval/ce` sait expanser les expressions qu'elle reçoit, alors capture-t-elle le contexte de macroexpansion ? En d'autres termes, si j'écris :

```
(let-syntax ((foo ...))
  (eval/ce (read)) )
```

Une expression lue contenant un site d'appel à `foo` sera-t-elle convenablement expansée par `eval/ce` ? Si c'est vrai alors non seulement `eval/ce` doit garder une trace de ce qu'est le contexte lexical mais aussi conserver l'état mémoire du macroexpanseur tout entier afin de préserver les capacités d'expansion de `foo`. Comme cela nuirait à l'indépendance de la macroexpansion, cela est considéré comme de mauvais goût. Mais alors, la bonne solution est que l'évaluation soit pure et ne force pas la macroexpansion d'emblée, mais alors ce n'est plus pratique, etc.

Lorsque l'on définit une macro, celle-ci est visible pour la suite de la macroexpansion. Supposons que j'aie défini l'utile macro `when` alors puis-je valablement écrire ?

```
(define-abbreviation (whenever condition . corps)
  (when condition (display `(whenever is called)))
  `(when ,condition . ,corps) )
```

Cette macro utilise deux fois le mot `when`. La seconde occurrence se situe dans le macroexpansé et ne pose pas de problème. La première par contre figure dans le calcul de macroexpansion et est donc évaluée par l'évaluateur de la macroexpansion qui a priori ne connaît point la macro `when` : le langage des macros n'est pas celui que l'on expanse ! Il aurait fallu auparavant, pour que la macro `when` soit utilisable par le langage de définition des macros, fournir cette définition :

```
(eval-in-abbreviation-world
  (define-abbreviation (when condition . body)
    `(if ,condition (begin . ,body)) ) )
```

Le problème est similaire au traitement de l'expression :

```
(define-abbreviation (foo)
  (define-abbreviation (bar)
    (when ...)
    (wrek) )
  (hux) )
```

La définition interne de `bar` est a priori destinée à étoffer le langage des macros et non le langage courant. L'expression précédente définit la macro `bar` comme macro d'emploi possible pour l'écriture de macro. Si la sémantique semble nette, la pragmatique est touchée car l'on a maintenant un problème de régression infinie. Le langage de définition des macros et le langage de définition des macros pour le langage de définition des macros ne sont a priori pas les mêmes. Autrement dit et dans l'exemple précédent, il n'est pas sûr que la forme `when` présente dans la définition de `bar` soit comprise. Si l'on avait des langages vraiment différents, le problème serait plus apparent encore. On peut par exemple imaginer que les macros sont traitées par `cpp` et que les macros, pour `cpp`, sont elles-mêmes obtenues par un programme `perl`. Il devient évident alors qu'une définition de macro à un niveau n'a d'implications que sur ce niveau.

Comment résoudre ce problème ? On peut soit s'attacher à la sémantique et avoir de multiples niveaux de langage [Que96] mais, fort heureusement, on a rarement besoin de dépasser le second niveau. On retrouve là les problèmes de niveaux de langage que l'on avait vu avec les interprètes réflexifs. **[☞ p. 287]**

Une autre solution serait de fusionner tous les niveaux de langage utilisés par le macroexpanseur, ce qui reviendrait à adopter l'hypothèse du monde unique pour

l'évaluateur de macroexpansion. La boucle est alors bouclée et tout ce que l'on avait séparé redeviendrait lié.

Une troisième solution, celle du R^{4}RS, est d'adopter une syntaxe pour le langage d'expression des macros (filtrage et reconstruction) interdisant la possibilité de macros pour ce dernier langage.

En résumé et une fois encore, il importe de bien séparer les langages mis en jeu. Dans une implantation de Scheme procurant distribution et parallélisme, il n'est peut-être pas bon que soient disponibles parallélisme et distribution pour l'expansion des macros. On pourrait de même interdire (mais comment ?) que puissent être écrites des macros employant des continuations non locales comme dans :

TASTELESS

```
(define-abbreviation (foo x)
  (call/cc (lambda (k)
             (set! the-k k)
             x )) )
(define-abbreviation (bar y)
  (the-k y) )
```

9.9 Emploi des macros

Cette section détaille quelques emplois typiques des macros. Si leur but est d'opérer une transformation de programme, cette dernière peut être souhaitée pour plusieurs desseins. On distinguera :

— Les *raccourcis*, où il s'agit de limiter le nombre de caractères frappés (surtout au niveau de la boucle d'interaction et en cours de mise au point). Écrire, par exemple, `(trace foo)` pour rendre visibles les appels à la fonction nommée `foo` requiert une macro pour ne point évaluer le paramètre `foo`.

— Les *enjolivures*, où l'on cherche à masquer une syntaxe disharmonieuse : par exemple `bind-exit` plutôt que `call/ep` pour éviter un `lambda` superflu.

— Les *masques*, où le but primordial est de cacher l'implantation sous-jacente susceptible de varier ou dont l'emploi direct est déconseillé. Citons, par exemple, `define-class` en MEROONET ou `syntax-rules` en Scheme.

Parmi les masques, on trouve aussi des macros dont les buts sont de s'abstraire de problèmes de portage. Utiliser une macro, **[☞ p. 314]** `meroon-apply`, pour masquer le fait qu'il y a dans la nature des `apply` binaires ou n-aires est une pratique courante. Un autre problème de portage est lié au caractère booléen de `()` en Scheme qui est Vrai en R^{4}RS mais ne l'est pas nécessairement en R^{3}RS. On peut s'affranchir de ce problème en employant partout une macro, disons `meroon-if`, définie comme suit :

```
(define-abbreviation (meroon-if condition consequent . alternant)
  `(if (let ((tmp ,condition))
         (or tmp (null? tmp)) )
       ,consequent . ,alternant ) )
```

Ce problème peut bien sûr être évité si le programme est robuste mais il reste que certains logiciels intéressants n'ont pas été écrit robustement. Les préfixer d'une telle macro est un moyen simple de les porter vers une autre implantation.

Parmi les masques, on trouve aussi des macros dont le rôle est d'assurer que certaines optimisations soient effectuées indépendamment du compilateur cible. Une très

importante amélioration est d'intégrer des fonctions (pour *inlining*). Certains compilateurs procurent une telle directive mais comme cette directive n'est pas uniformément répandue et que son application, par delà les frontières d'un module, est problématique, le plus simple est de réaliser soi-même cette intégration. Rendre les appels à une fonction intégrables, c'est associer à la fonction une macro de même nom. On définira donc :

```
(define-abbreviation (define-inline call . body)
  (let ((name      (car call))
        (variables (cdr call)) )
    `(begin
       (define-abbreviation (,name . arguments)
         (cons (cons 'lambda (cons ',variables ',body))
               arguments ) )
       (define ,call (,name . ,variables)) ) ) )
```

Cette macro souffre elle-même de problèmes de portabilité car dans certains dialectes, il n'est pas possible d'avoir à la fois une macro et une fonction de même nom. Dans d'autres dialectes, c'est autorisé et la seconde définition (la fonction) va remplacer complètement la première définition (la macro) qui ne sera donc plus accessible. De toute façon, il est quasiment exclu que la fonction soit récursive sans faire boucler la macro (voir cependant [Bak92a]). Enfin lorsqu'une fonction est intégrée, il vaut mieux éviter de l'utiliser comme valeur de première classe (par exemple en premier argument d'`apply`), ce qui permet de ne pas définir la fonction associée.

La macro précédente est facilement ingérable en mondes multiples puisque la définition de la fonction n'est qu'expansée et grâce à la macro qui précède. Dans le code expansé résultant, seule la fonction demeurera dans tous ses emplois tandis que tous les sites d'appels à la macro éponyme auront été expansés ce qui était l'effet escompté.

9.9.1 Autres caractéristiques

Les macros en Scheme R^4RS ont quatre grandes caractéristiques : *(i)* elles sont *hygiéniques* ce que nous analyserons en section 9.10 ; *(ii)* elles sont définies par filtrage ; *(iii)* elles construisent le macroexpansé par substitution ; *(iv)* elles n'ont pas d'état interne. La définition par filtrage est très intéressante car elle permet de décrire finement la forme que doit respecter l'appel aux macros et, surtout, elle évite d'écrire soi-même le code permettant de vérifier cette conformité. Par exemple, la plupart des macros dont le dernier argument forme un `begin` implicite, ne testent pas que l'ultime `cdr` est bien `()` : c'est automatiquement vérifié (avec un message d'erreur approprié) par un filtre. Des compilateurs efficaces de filtres existent et sont décrits en détail dans la littérature [Que90b, QG92]. La construction par substitution s'apparente à la notation *backquote* mais restreint les calculs que l'on peut mener aux seules opérations sur les listes. En particulier, elle ne procure pas les opérations arithmétiques. **[☞ Ex. 9.2]**

Que les macros n'aient pas d'état interne est plus gênant car cela interdit d'avoir des macros contextuelles comme `define-class` qui doit tenir à jour la hiérarchie des classes connues afin d'y trouver les informations nécessaires à l'introduction de sous-classes (noms et nombre des champs hérités par exemple).

On peut de même imaginer une macro `(date)` retournant une chaîne de caractères

contenant la date courante au moment de l'expansion. Cette macro peut servir à l'instar de RCS de Walter Tichy ou SCCS d'Eric Allman à maintenir des versions de logiciels.

9.9.2 Arpenteur de code

La plupart des macros que l'on écrit prennent une expression et en retournent une autre qu'elles construisent à partir des sous-termes de la première expression. Ce sont des macros très superficielles qui n'ont pas besoin d'arpenter l'expression d'entrée. Pourtant une macro effectuant, par exemple, une traduction d'expressions arithmétiques infixes en formes Lisp, donc préfixes, a besoin d'analyser plus profondément son argument. Prenons le cas, par exemple, de la macro `with-slots` de CLOS que nous adapterons au contexte de MEROONET. Les champs d'un objet, disons d'une instance de `Point`, sont manipulés via les fonctions de lecture/écriture comme `Point-x` ou `set-Point-y!` : il serait plus simple de les manipuler directement par leur nom de champ `x` ou `y`, par exemple dans le contexte d'une définition de méthode. On pourrait ainsi écrire, similairement à Smalltalk [GR83] :

```
(define-handy-method (double (o Point))
  (set! x (* 2 x))
  (set! y (* 2 y))
  o )
```

à la place de :

```
(define-method (double (o Point))
  (set-Point-x! o (* 2 (Point-x o)))
  (set-Point-y! o (* 2 (Point-y o)))
  o )
```

La nouvelle macro `define-handy-method` doit donc maintenant arpenter son corps pour y convertir, en accès normaux, les références et les affectations aux variables portant le nom des champs de la classe discriminée. Les accès à ces champs pourront utiliser des accesseurs non sûrs (c'est-à-dire ne testant pas l'appartenance de leur argument à la classe) puisque cette appartenance est garantie par le mécanisme de discrimination. La simplicité des accès se double donc d'une efficacité accrue.

Pour réaliser cela, il faut tout d'abord se rappeler que le corps de la méthode n'est pas un programme mais une Sexpression devant être macroexpansée avant de pouvoir être analysée. Il est donc nécessaire d'avoir accès au mécanisme de macroexpansion. Comme celui-ci peut procurer des macros locales, il est nécessaire d'avoir accès à l'expanseur même qui s'occupe de la forme originale. On supposera sans se préoccuper du mécanisme exact que cet expanseur est la valeur de la variable (globale ou locale) nommée `macroexpand` et toujours visible depuis le corps des macros.

Une fois le corps de `define-handy-method` expansé, alors c'est maintenant un vrai programme qu'il faut analyser. Analyser des programmes, c'est ce que nous faisons depuis le début de cet ouvrage et cela passe par la connaissance des formes spéciales du langage en cours d'analyse. On n'ira pas chercher des références aux champs dans des citations, il faudra, par contre, tenir compte des formes de liaison pouvant masquer les champs. Écrire un arpenteur de code n'est pas très compliqué [Cur89, Wat93]), **[☞ p. 343]** ce qui l'est plus est de connaître exactement le jeu de formes spéciales du langage ou plutôt de l'implantation que l'on utilise.

Les formes spéciales sont les points de référence d'une implantation mais cette dernière s'offre parfois quelques privautés comme *(i)* de procurer des formes spéciales supplémentaires (par exemple `let` ou `letrec`) ou des formes spéciales occultes et propres à l'implantation (par exemple `define-class`); *(ii)* d'implanter des formes spéciales sous forme de macros (par exemple `begin`). En l'absence de connaissances réflexives sur le langage, il est difficile de s'en sortir car comment traiter particulièrement les formes `begin` si celles-ci ont disparu ? Comment trouver les conditionnelles si `if` est une macro s'expansant en la forme spéciale `typecase` ? Comment arpenter une forme spéciale inconnue comme `define-class` qui comporte des sous-structures (les définitions des champs) qui syntaxiquement ressemblent à des applications fonctionnelles ? Comment deviner que `bind-exit` est une forme spéciale établissant une liaison pouvant masquer une variable de même nom ?

Un bon moyen terme serait de fixer le jeu de formes spéciales et d'interdire qu'il y en ait plus ou moins. Cela signifierait que les implantations qui usent de macros spéciales ne pourraient le faire que dans une phase postérieure à la macroexpansion. Mais en l'absence de connaissances réflexives sur le langage et ses implantations, il n'est pas possible d'écrire des arpenteurs de code portables en Scheme et nous devons renoncer à pouvoir écrire `define-handy-method`.

9.10 Captures inopinées

Un aspect des macros qui a été très étudié ces dernières années [KFFD86, BR88, CR91a] est relatif à la notion d'*hygiène*. Le macroexpansé contient des symboles qui sont en quelque sorte « libres », c'est-à-dire sans attache et donc susceptibles d'interférer (capturer ou être capturé) avec le contexte où est employé la macro. Voici un exemple qui conjugue ces deux effets :

```
(define-abbreviation (acons key value alist)
  `(let ((f cons)) (f (f ,key ,value) ,alist)) )
(let ((cons list)
      (f    #f) )
  (acons 'false f '()) )
```

La référence à `cons`, qui figure dans le macroexpansé de la macro `acons` et qui désigne la brave fonction `cons` usuelle, va être capturée par la variable locale `cons` présente dans la forme `let` qui entoure le site d'appel à `acons`. Inversement, le macroexpansé établit une liaison pour la variable `f` qui va empêcher le second argument de l'appel à `acons` de prendre la valeur `#f` qu'il aurait normalement dû prendre : la macro a intercepté la variable `f`. Rendre les macros hygiéniques permet de se délivrer automatiquement de ces deux problèmes.

Le second problème se résoud depuis plus de trente ans en Lisp par un simple renommage : la variable `f` qu'introduit le macroexpansé ne doit interférer en rien avec l'environnement lexical d'emploi de la macro. On va donc effectuer une α-conversion, engendrer un `gensym` nouveau et inimitable, afin de protéger cette variable. On écrira donc plus sereinement :

```
(define-abbreviation (acons key value alist)
  (let ((f (gensym)))
    `(let ((,f cons)) (,f (,f ,key ,value) ,alist)) ) )
```

Le premier de ces problèmes, le fait de produire une référence libre à `cons` dans le macroexpansé, est plus compliqué à traiter. Dans le cas de la présente macro, ce que l'on souhaite est que l'écrivain de la macro puisse spécifier que la référence à `cons` est la référence à la variable globale `cons` et nulle autre. Or justement, la variable globale `cons` était celle qui était visible depuis la définition de la macro `acons` ; d'où la *règle d'hygiène* : les références libres dans le macroexpansé sont celles qui étaient visibles depuis l'endroit où était définie la macro.

Dans le cas de `cons` et dans certains systèmes Lisp ou Scheme, il existe souvent une méthode pour référencer une variable globale indépendamment du contexte, l'équivalent d'une forme `(global cons)` ou `lisp:cons`. Mais l'exemple suivant montrera que l'on peut aussi vouloir associer, à une variable du macroexpansé, une liaison locale.

```
(let ((results '())
      (compose cons) )
  (let-syntax ((push (syntax-rules ()
                       ((push e)
                        (set! results (compose e results)) ) )))
    π
    results ) )
```

Dans tout le corps π de cette expression, on souhaite que la macro locale `push` compose son argument avec le contenu de la variable `results` et remplace celle-ci avec ce résultat-là. Pour être hygiénique, il faut assurer qu'aucune liaison interne à π ne peut modifier le sens de `push`. Il y a basiquement deux solutions : *(i)* renommer toutes les variables internes à π de manière à ce que `results` et `compose` y soient toujours visibles et inambiguës ; *(ii)* renommer `results` et `compose` de façon cohérente, c'est-à-dire dans les liaisons du `let`, dans la définition de `push` et, bien sûr, en dernière expression du corps du `let`.

Bien que nous ayons parlé de « capture » de liaison, il n'y a en fait rien de tel dans les macros hygiéniques : elles ne capturent pas de liaisons car celles-ci n'existent pas encore puisque l'on est en phase de macroexpansion ! Plus fort encore, le R^{4}RS ne réserve pas de mot-clés, ce qui rend possible de définir des macros portant le nom de forme spéciale. Ainsi donc, même le mot `set!` qui apparaît libre dans le macroexpansé de `push` pourrait être capturé par une macro locale à π. On ne capture donc pas des liaisons puisque les formes spéciales en sont dépourvues, on capture seulement le sens, l'intention des mots !

L'hygiène des macros est une remarquable propriété mais que l'on ne peut exclusivement adopter car il existe des macros importantes qui ne sont pas hygiéniques. L'exemple le plus courant est la macro `loop` dont on souhaite sortir grâce à la fonction `exit`. Si l'on écrivait :

WRONG

```
(define-syntax loop
  (syntax-rules ()     ; should be (exit) instead
    ((loop e1 e2 ...)
     (call/cc (lambda (exit)
                (let loop () e1 e2 ... (loop)) )) ) ) )
```

On ne pourrait jamais sortir de cette boucle car la variable `exit` introduite dans le macroexpansé ne peut, hygiène oblige, être capturée par une quelconque référence à `exit` dans l'une des formes `e1 e2 ...`, puisque, si `exit` figurait parmi elles, son intention serait de mentionner ce qu'`exit` désigne au site d'appel. La solution est de

mentionner le symbole `exit` comme devant rester libre dans le macroexpansé et, pour cela, le faire figurer dans les parenthèses qui suivent le mot `syntax-rules`. Le bon défaut est que tout soit hygiénique (c'est ce que l'on a d'ailleurs sans effort pour la variable locale `loop`) mais il faut parfois s'en affranchir.

Plusieurs implantations rivales de macros hygiéniques existent sur le réseau qui pourront être lues avec intérêt. Notre solution va s'affranchir du filtrage, ne limitera pas les calculs effectuables dans les macros et correspondra à une implantation de bas niveau que nous pensons être plus simple que celles apparaissant dans la littérature [CR91b, DHB93]. Notre implantation proposera une variante nouvelle où l'on mentionnera explicitement les mots dont on veut fixer le sens. Donnons de suite un exemple : la forme `with-aliases` prendra en premier argument une suite de couples variable-mot et liera *pour la macroexpansion* ces variables à la signification qu'ont ces mots dans le contexte courant. Le sens préservé de ces mots est accessible via ces variables et peut être inséré dans le macroexpansé. Les paramètres suivants forment le corps de la forme `with-aliases` dont le rôle est de calculer le macroexpansé. Le précédent exemple se réécrira donc en :

```
(let ((results '())
      (compose cons) )
  (with-aliases ((setq set!) (results results) (compose compose))
    (let-abbreviation  ( ( (push e)
                           `(,setq ,results
                                   (,compose ,e ,results) ) ) )
      π
      results ) ) )
```

Tandis que la macro `loop` précédente se définira comme :

```
(with-aliases ((cc call/cc)(lam lambda) (ll let))
  (define-abbreviation (loop . body)
    (let ((loop (gensym)))
      `(,cc (,lam (exit) (,ll ,loop () ,@body (,loop)))) ) ) )
```

Dans ces deux exemples, tous les termes, dont on souhaitait qu'ils fussent figés, font l'objet d'un `with-aliases` qui capture leur sens. Ils sont ensuite réinsérés dans les macroexpansés et donc indépendants du contexte d'emploi de ces macros.

On remarquera que la notation *backquote* n'est pas très appropriée ici car la proportion d'éléments variants est très importante. Ce système de macros est de bas niveau car il ne procure pas automatiquement l'hygiène mais seulement les outils pour l'obtenir. C'est ce système de macros que nous allons développer dans la section prochaine.

9.11 Un système de macros

Cette section décrit l'implantation d'un système de macros procurant les fonctionnalités suivantes :

— `define-abbreviation` pour définir une macro globale ;
— `let-abbreviation` pour définir une macro locale ;
— `eval-in-abbreviation-world` pour évaluer dans le monde des macros ;
— `with-aliases` pour préserver des sens.

Afin de bien montrer la différence qui existe entre la préparation et l'exécution, les programmes seront expansés et, à la volée, transformés en objets. Les objets résultant pourront être **[☞ p. 175]** évalués par un interprète rapide apparenté à celui du chapitre 6 ou compilés vers C par le compilateur du chapitre 10. L'expansion a donc lieu en une seule **[☞ p. 341]** passe et les formes sont figées en objets dès qu'expansées.

9.11.1 Objectification

Que l'on nous pardonne ce néologisme audacieux à ne pas confondre avec son double savant, *réification*, que nous avons déjà employé dans un autre sens. Le programme à compiler est tout d'abord converti en un objet. Lors de sa conversion, sa syntaxe aura été vérifiée et normalisée de façon à ne plus être la source de soucis ultérieurs. Cette transformation rappelle l'interprétation rapide dont on épouse ici les mêmes buts toutefois nous la ferons de manière ad hoc. Au lieu de faire apparaître des fermetures sans arguments contenant tous les ingrédients nécessaires à leur évaluation, comme au chapitre sur l'interprétation rapide, nous allons en faire **[☞ p. 201]** des objets qui pourront non seulement être évalués (ce qui permet d'éprouver la robustesse de cette transformation) mais plus généralement manipulés. Voici donc la litanie des classes que nous retenons pour ces objets :

```
(define-class Program Object ())
(define-class Reference Program (variable))
(define-class Local-Reference Reference ())
(define-class Global-Reference Reference ())
(define-class Predefined-Reference Reference ())
(define-class Global-Assignment Program (variable form))
(define-class Local-Assignment Program (reference form))
(define-class Function Program (variables body))
(define-class Alternative Program (condition consequent alternant))
(define-class Sequence Program (first last))
(define-class Constant Program (value) )
(define-class Application Program ())
(define-class Regular-Application Application (function arguments))
(define-class Predefined-Application Application
  (variable arguments) )
(define-class Fix-Let Program (variables arguments body))
(define-class Arguments Program (first others))
(define-class No-Argument Program ())
(define-class Variable Object (name))
(define-class Global-Variable Variable ())
(define-class Predefined-Variable Variable (description))
(define-class Local-Variable Variable (mutable? dotted?))
```

Dans cette énumération figurent de nombreux points déjà discutés : par exemple, les applications closes sont traitées spécialement et les appels aux fonctions prédéfinies sont identifiés. La classe `Program` ne contient que des éléments évaluables : les références ou affectations en sont. En revanche, les instances de variables, représentant les liaisons, ne sont point des programmes mais des instances de `Variable`. La définition des classes ne présente que peu de singularités. Elles ont en général autant de champs qu'elles comportent de composants syntaxiques. La classe des variables

locales comporte toutefois deux booléens qualifiant leur modifiabilité et leur méthode de liaison : les variables pointées (`dotted?`) devant récupérer une liste d'arguments subissent un traitement spécial.

La lecture de l'expression à compiler est supposée réalisée par `read` ou tout autre équivalent retournant la Sexpression lue. En fait, il serait judicieux de lire en utilisant des `cons` à cinq champs mémorisant dans quel fichier, à quelle ligne et quelle colonne sont les expressions lues afin de donner d'excellents avertissements en cas d'erreur. D'autre part, les symboles pourraient disposer d'un champ supplémentaire contenant l'éventuelle macro associée : la recherche d'une telle macro serait alors très rapide. Que les symboles et les paires soient augmentés d'un tel champ pendant la phase de macroexpansion n'est pas gênant puisque rien n'en restera après expansion.

Cette Sexpression est alors arpentée pour être convertie en un objet de classe `Program` et, au passage, les macros seront identifiées et expansées. La fonction principale s'appelle `objectify` et prend en second argument l'environnement lexical de préparation. Basiquement, sur une forme, elle analyse le terme en position fonctionnelle et déclenche le traitement approprié. Les traitements ne sont plus câblés mais se trouvent dans le champ `handler` des objets de classe `Magic-Keyword` (selon les termes de [SS75]). La fonction `objectify` pave le chemin menant au traitement des macros et sera expliquée plus bas.

```
(define-class Magic-Keyword Object (name handler))
(define (objectify e r)
  (if (atom? e)
      (cond ((Magic-Keyword? e) e)
            ((Program? e)       e)
            ((symbol? e)        (objectify-symbol e r))
            (else               (objectify-quotation e r)) )
      (let ((m (objectify (car e) r)))
        (if (Magic-Keyword? m)
            ((Magic-Keyword-handler m) e r)
            (objectify-application m (cdr e) r)) ) ) )
```

Il ne reste plus qu'à expliciter les différentes sous-fonctions spécialisées de conversion. La plupart se contentent de fabriquer un objet dont les champs sont initialisés par le résultat de l'arpentage récursif des sous-termes de la Sexpression correspondante. C'est notamment le cas de `objectify-alternative` et `objectify-sequence` : cette dernière analyse cependant la forme initiale afin de la normaliser en séquences binaires.

```
(define (objectify-quotation value r)
  (make-Constant value) )
(define (objectify-alternative ec et ef r)
  (make-Alternative (objectify ec r)
                    (objectify et r)
                    (objectify ef r) ) )
(define (objectify-sequence e* r)
  (if (pair? e*)
      (if (pair? (cdr e*))
          (let ((a (objectify (car e*) r)))
            (make-Sequence a (objectify-sequence (cdr e*) r)) )
          (objectify (car e*) r) )
      (make-Constant 42) ) )
```

L'application est un peu plus complexe puisqu'elle doit analyser l'expression ini-

tiale et la catégoriser comme une application close, un appel à une fonction prédéfinie ou une application normale. Ce choix s'effectue simplement par analyse de l'objet correspondant au terme en position fonctionnelle. Le seul point peut-être obscur, mais déjà abordé, **[☞ p. 191]** est l'existence et l'usage de description permettant de connaître l'arité des fonctions prédéfinies afin de mieux les compiler. La classe `Functional-Description` sera détaillée plus tard lorsque nous nous intéresserons à la mise en place de l'environnement prédéfini.

```
(define (objectify-application ff e* r)
  (let ((ee* (convert2arguments (map (lambda (e) (objectify e r))
                                     e* ))) )
    (cond ((Function? ff)
           (process-closed-application ff ee*) )
          ((Predefined-Reference? ff)
           (let* ((fvf (Predefined-Reference-variable ff))
                  (desc (Predefined-Variable-description fvf)) )
             (if (Functional-Description? desc)
                 (if ((Functional-Description-comparator desc)
                      (length e*)
                      (Functional-Description-arity desc) )
                     (make-Predefined-Application fvf ee*)
                     (objectify-error
                      "Incorrect predefined arity" ff e* ) )
                 (make-Regular-Application ff ee*) ) ) )
          (else (make-Regular-Application ff ee*)) ) ) )
(define (process-closed-application f e*)
  (let ((v* (Function-variables f))
        (b  (Function-body f)) )
    (if (and (pair? v*) (Local-Variable-dotted? (car (last-pair v*))))
        (process-nary-closed-application f e*)
        (if (= (number-of e*) (length (Function-variables f)))
            (make-Fix-Let (Function-variables f) e* (Function-body f))
            (objectify-error "Incorrect regular arity" f e*) ) ) ) )
```

La liste des arguments est traduite en un unique objet (de classe `Arguments`) et terminée par un objet de la classe `No-Argument`. Le nombre d'arguments peut être déterminé grâce à la fonction générique `number-of`.

```
(define (convert2arguments e*)
  (if (pair? e*)
      (make-Arguments (car e*) (convert2arguments (cdr e*)))
      (make-No-Argument) ) )

(define-generic (number-of (o)))
(define-method (number-of (o Arguments))
  (+ 1 (number-of (Arguments-others o))) )
(define-method (number-of (o No-Argument)) 0)
```

Comme d'habitude, on distinguera au sein des applications closes le cas particulier, rare et encombrant des fonctions n-aires appliquées. Les arguments supplétifs sont regroupés en une liste, on modifie donc la variable pointée associée pour qu'elle redevienne normale. **[☞ p. 188]**

```
(define (process-nary-closed-application f e*)
   (let* ((v* (Function-variables f))
          (b  (Function-body f))
          (o (make-Fix-Let
              v*
```

```
(let gather ((e* e*) (v* v*))
  (if (Local-Variable-dotted? (car v*))
      (make-Arguments
       (let pack ((e* e*))
         (if (Arguments? e*)
             (make-Predefined-Application
              (find-variable? 'cons g.predef)
              (make-Arguments
               (Arguments-first e*)
               (make-Arguments
                (pack (Arguments-others e*))
                (make-No-Argument) ) ) )
             (make-Constant '()) ) )
       (make-No-Argument) )
      (if (Arguments? e*)
          (make-Arguments (Arguments-first e*)
                          (gather (Arguments-others e*)
                                  (cdr v*) ) )
          (objectify-error
           "Incorrect dotted arity" f e* ) ) ) )
    b )) )
(set-Local-Variable-dotted?! (car (last-pair v*)) #f)
o ) )
```

Enfin, les abstractions sont analysées et transformées grâce à `objectify-function` qui use de `objectify-variables-list` pour traiter la liste des variables et ainsi enrichir l'environnement lexical dans lequel est traité le corps de la fonction.

```
(define (objectify-function names body r)
  (let* ((vars (objectify-variables-list names))
         (b    (objectify-sequence body (r-extend* r vars))) )
    (make-Function vars b) ) )
(define (objectify-variables-list names)
  (if (pair? names)
      (cons (make-Local-Variable (car names) #f #f)
            (objectify-variables-list (cdr names)) )
      (if (symbol? names)
          (list (make-Local-Variable names #f #t))
          '() ) ) )
```

Finalement, les variables sont l'objet des soins attentifs d'`objectify-symbol` qui les recherche dans l'unique environnement lexical statique et courant : `r`. Lorsque la variable n'est pas trouvée, elle est ajoutée à l'environnement global modifiable grâce à la fonction `objectify-free-global-reference`. Nous adoptons donc ici un mode de définition automatique des nouvelles variables.

```
(define (objectify-symbol variable r)
  (let ((v (find-variable? variable r)))
    (cond ((Magic-Keyword? v)        v)
          ((Local-Variable? v)       (make-Local-Reference v))
          ((Global-Variable? v)      (make-Global-Reference v))
          ((Predefined-Variable? v) (make-Predefined-Reference v))
          (else (objectify-free-global-reference variable r)) ) ) )
(define (objectify-free-global-reference name r)
  (let ((v (make-Global-Variable name)))
    (insert-global! v r)
    (make-Global-Reference v) ) )
```

L'environnement `r` est grosso modo une liste de variables locales suivies de va-

riables globales puis prédéfinies. C'est l'environnement statique qui ne contient pas les valeurs de ces variables, apanage de l'évaluateur subséquent. Cet environnement est représenté par une liste d'instances d'`Environnement`. Il est possible d'étendre cet environnement par des variables locales ou par de nouvelles variables globales, ces dernières par accrochage physique dans la partie globale de l'environnement que l'on sait retrouver grâce à la fonction `find-global-environment`.

```
(define-class Environment Object (next))
(define-class Full-Environment Environment (variable))
(define (r-extend* r vars)
  (if (pair? vars)
      (r-extend (r-extend* r (cdr vars)) (car vars))
      r ) )
(define (r-extend r var)
  (make-Full-Environment r var) )
(define (find-variable? name r)
  (if (Full-Environment? r)
      (let ((var (Full-Environment-variable r)))
        (if (eq? name
                 (cond ((Variable? var) (Variable-name var))
                       ((Magic-Keyword? var)
                        (Magic-Keyword-name var) ) ) )
            var
            (find-variable? name (Full-Environment-next r)) ) )
      (if (Environment? r)
          (find-variable? name (Environment-next r))
          #f ) ) )
(define (insert-global! variable r)
  (let ((r (find-global-environment r)))
    (set-Environment-next!
     r (make-Full-Environment (Environment-next r) variable) ) ) )
(define (mark-global-preparation-environment g)
  (make-Environment g) )
(define (find-global-environment r)
  (if (Full-Environment? r)
      (find-global-environment (Full-Environment-next r))
      r ) )
```

Le traitement de l'affectation en profite pour annoter toutes les variables locales subissant une affectation et positionne leur champ `mutable?` à Vrai. Ce champ sera utilisé ultérieurement au chapitre 10. **[☞ p. 341]**

```
(define (objectify-assignment variable e r)
  (let ((ov (objectify variable r))
        (of (objectify e r)) )
    (cond ((Local-Reference? ov)
           (set-Local-Variable-mutable?!
            (Local-Reference-variable ov) #t )
           (make-Local-Assignment ov of) )
          ((Global-Reference? ov)
           (make-Global-Assignment (Global-Reference-variable ov)
                                  of ) )
          (else (objectify-error
                 "Illegal mutated reference" variable )) ) ) )
```

Lorsque l'expression initiale est transformée en un objet, il est simple d'écrire un évaluateur pour interpréter cet objet ce qui permet donc de vérifier le procédé de tra-

duction. L'évaluateur ressemble à ceux présentés aux chapitres précédents et notamment à celui (à objets) du chapitre 3 **[☞ p. 67]** (quelque peu mâtiné de l'interprète rapide du chapitre 6). **[☞ p. 175]**

Rappelons toutefois les grandes lignes de cet évaluateur telles qu'elles transparaîtront par la suite. L'évaluation est menée par la fonction générique `evaluate` qui prendra deux arguments : le premier sera une instance de `Program` et le second une instance d'`Environment` (une sorte de A-liste de couples variables-valeurs). L'environnement initial compte les seules variables prédéfinies. Sa partie statique est la valeur de `g.predef` tandis que sa partie dynamique est la valeur de `sg.predef` et associe aux variables prédéfinies fonctionnelles des instances de `RunTime-Primitive`. L'environnement d'exécution peut être étendu par `sr-extend`.

9.11.2 Les formes spéciales

Le précédent traducteur ne connaissait aucune forme spéciale, il faut pour chacune d'entre elles indiquer comment la transformer en une instance de `Program`. Nous allons associer à chaque mot-clé un arpenteur approprié qui se contentera d'appeler l'une des fonctions précédentes. Elles sont toutes bâties sur le même modèle.

```
(define special-if
  (make-Magic-Keyword
   'if (lambda (e r)
         (objectify-alternative (cadr e) (caddr e) (cadddr e)
                                r ) ) ) )
(define special-begin
  (make-Magic-Keyword
   'begin (lambda (e r)
            (objectify-sequence (cdr e) r) ) ) )
(define special-quote
  (make-Magic-Keyword
   'quote (lambda (e r)
            (objectify-quotation (cadr e) r) ) ) )
(define special-set!
  (make-Magic-Keyword
   'set! (lambda (e r)
           (objectify-assignment (cadr e) (caddr e) r) ) ) )
(define special-lambda
  (make-Magic-Keyword
   'lambda (lambda (e r)
             (objectify-function (cadr e) (cddr e) r) ) ) )
```

On pourrait bien sûr définir d'autre formes spéciales que nous ajouterions aux précédents mots-clés magiques pour former la liste : `*special-form-keywords*`.

```
(define *special-form-keywords*
  (list special-quote
        special-if
        special-begin
        special-set!
        special-lambda
        ;; cond, letrec, etc.
        special-let
        ) )
```

9.11.3 Les niveaux d'évaluation

On l'a vu, la macroexpansion endogène cache un évaluateur en elle. Comme nous ne voulons pas confondre ces différents évaluateurs (le résultat de l'objectification n'est pas forcément évalué de suite dans le même espace mémoire ; le chapitre 10 montrera, en effet, comment compiler vers C), **[☞ p. 341]** nous allons introduire une tour d'évaluateurs [dRS84, Que96] et prendre ainsi l'hypothèse la plus pure où tous les évaluateurs sont distincts : le langage, le langage des macros, le langage des macros de macros, etc., auront tous des environnements globaux différents. Ces évaluateurs seront caractérisés par la classe `Evaluator` définissant leurs principales caractéristiques.

```
(define-class Evaluator Object
  ( mother
    Preparation-Environment
    RunTime-Environment
    eval
    expand
    ) )
```

Ce qui suit est assez complexe et nécessite de bien distinguer les différents environnements en jeu comme illustré à la figure 9.9. La fonction d'expansion d'un étage utilise la fonction d'évaluation de l'étage suivant mais qui elle-même commence par analyser les expressions qu'elle reçoit et donc par les expanser. La régression infinie sera évitée lorsqu'un étage expanse une expression dénuée de macros, auquel cas il n'a pas besoin de l'étage suivant. En général, deux à trois étages semblent suffire. Chacun des étages possède un environnement de préparation (pour `expand`) et un environnement d'exécution (pour `eval`), sauf peut-être pour le rez-de-chaussée dont on n'utilise que l'expanseur.

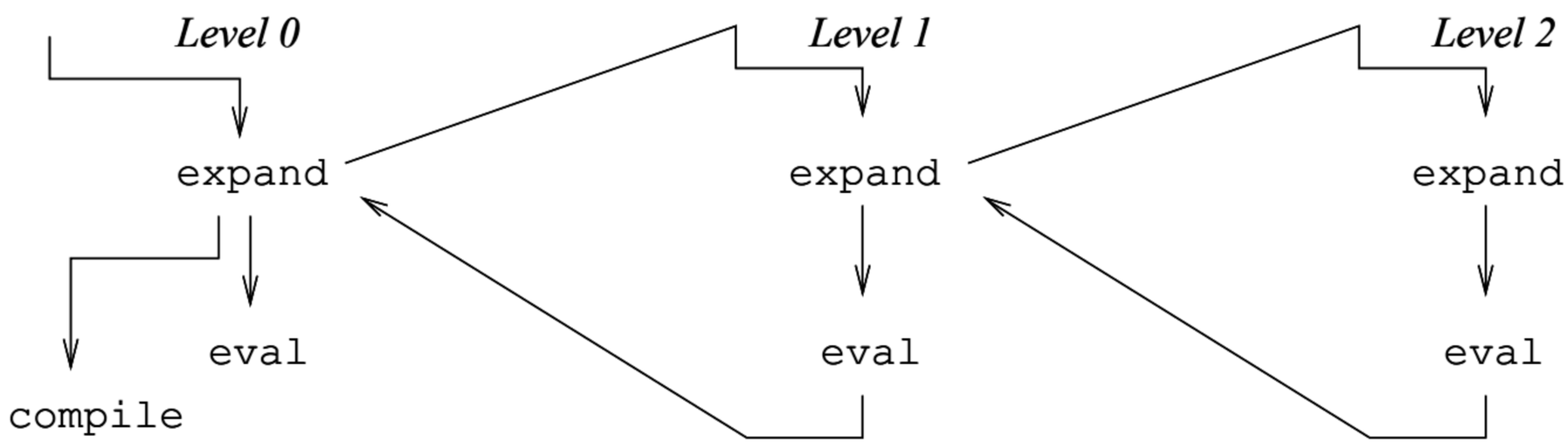


FIGURE 9.9 – Tour d'évaluateurs

Les étages de la tour sont construits, un par un, à la demande, par la fonction `create-evaluator`. Celle-ci bâtit un nouvel étage au dessus de celui qu'elle reçoit en argument. Elle crée un couple de fonctions, `expand` et `eval`, ainsi que deux environnements de préparation et d'exécution. La fonction `eval` expanse systématiquement son argument avant de l'évaluer grâce à un moteur d'évaluation nommé `evaluate`, dont nous ne dirons rien d'autre si ce n'est qu'il n'a besoin que de l'environnement d'exécution, conservé dans le champ `RunTime-Environment` de l'instance d'`Evaluator` courant, et qu'il modifie physiquement en cas de besoin. La fonction d'expansion utilise le moteur à objectifier `objectify` avec un environnement de préparation, lui-même stocké dans le champ `Preparation-Environment` de l'instance d'`Evaluator` courant.

Pour nous simplifier la vie, les variables globales découvertes lors de l'expansion sont *ipso facto* créées dans l'environnement d'exécution associé grâce à la fonction `enrich-with-new-global-variables!`. La fonction `eval` est installée réflexivement dans son propre environnement d'exécution comme une primitive prédéfinie : il y a une fonction `eval` différente par étage. L'environnement de préparation est étendu par toutes les formes spéciales accumulées dans la variable `*special-form-keywords*` qui doivent être visibles de tous les étages. Il est aussi étendu par les macros prédéfinies, résultat de l'appel à `(make-macro-environment level)` que nous détaillerons plus bas.

```
(define (create-evaluator old-level)
  (let ((level 'wait)
        (g     g.predef)
        (sg    sg.predef) )
    (define (expand e)
      (let ((prg (objectify
                   e (Evaluator-Preparation-Environment level) )))
        (enrich-with-new-global-variables! level)
        prg ) )
    (define (eval e)
      (let ((prg (expand e)))
        (evaluate prg (Evaluator-RunTime-Environment level)) ) )
    ;; Create resulting evaluator instance
    (set! level (make-Evaluator old-level 'wait 'wait eval expand))
    ;; Enrich environment with  eval
    (set! g (r-extend* g *special-form-keywords*))
    (set! g (r-extend* g (make-macro-environment level)))
    (let ((eval-var (make-Predefined-Variable
                      'eval (make-Functional-Description = 1 "") ))
          (eval-fn (make-RunTime-Primitive eval = 1)) )
      (set! g (r-extend g eval-var))
      (set! sg (sr-extend sg eval-var eval-fn)) )
    ;; Mark the beginning of the global environment
    (set-Evaluator-Preparation-Environment!
     level (mark-global-preparation-environment g) )
    (set-Evaluator-RunTime-Environment!
     level (mark-global-runtime-environment sg) )
    level ) )
```

Si l'on a besoin d'un expanseur ou d'un évaluateur, il suffira d'invoquer la fonction `create-evaluator` et d'en retirer, par simple lecture de champs, la fonction `expand` ou `eval` souhaitée. Attention, par rapport à un expanseur classique, la fonction `expand` retourne une instance de `Program` et non une expression de Scheme. Une simple fonction de conversion permettra de retrouver une telle expression. **[☞ Ex. 9.4]**

9.11.4 Les macros

Les macros prédéfinies avaient été ajoutées à l'environnement de préparation et étaient synthétisées par la fonction `make-macro-environment`. Les quatre macros prédéfinies sont `eval-in-abbreviation-world`, `define-abbreviation`, `let-abbreviation` et `with-aliases`. Toutes supposent l'existence d'un évaluateur appartenant à l'étage supérieur que la fonction `make-macro-environment` se doit de créer. Mais, pour éviter une régression infinie, la création de cet étage constitue une promesse qui

ne sera réalisée que si l'on invoque l'évaluateur associé. **[☞ Ex. 9.3]** Lorsqu'un étage supplémentaire est créé, il ne sera pas recréé à nouveau et perdurera, accumulant les définitions qu'il reçoit.

```
(define (make-macro-environment current-level)
  (let ((metalevel (delay (create-evaluator current-level))))
    (list (make-Magic-Keyword 'eval-in-abbreviation-world
           (special-eval-in-abbreviation-world metalevel) )
          (make-Magic-Keyword 'define-abbreviation
           (special-define-abbreviation metalevel))
          (make-Magic-Keyword 'let-abbreviation
           (special-let-abbreviation metalevel))
         (make-Magic-Keyword 'with-aliases
          (special-with-aliases metalevel) ) ) ) )
```

La macro `eval-in-abbreviation-world` est la plus simple, elle se contente d'évaluer son corps avec l'évaluateur de l'étage supérieur dont elle requiert la construction effective si besoin est. Comme toute macro, le résultat est à nouveau soumis à la fonction d'objectification de l'étage courant.

```
(define (special-eval-in-abbreviation-world level)
  (lambda (e r)
    (let ((body (cdr e)))
      (objectify ((Evaluator-eval (force level))
                  `(,special-begin . ,body) )
                 r ) ) ) )
```

La macro `define-abbreviation` permet de créer ou de modifier des macros globales. L'expanseur associé est créé puis évalué à l'étage supérieur, un nouveau mot-clé magique est créé dans l'environnement global de préparation de l'étage courant. Lorsque la macro est invoquée (par la fonction `invoke` qui est moyen différent de `evaluate` de démarrer le moteur d'évaluation), elle déroulera ses calculs dans l'environnement d'exécution de l'étage supérieur (enclos dans la fermeture de l'expanseur). Le résultat d'une telle macro est, ici, `#t`. On aurait pu retourner le nom de la macro définie mais cela aurait constitué un débris (un symbole) de quelques octets.

```
(define (special-define-abbreviation level)
  (lambda (e r)
    (let* ((call      (cadr e))
           (body      (cddr e))
           (name      (car call))
           (variables (cdr call)) )
      (let ((expander ((Evaluator-eval (force level))
                       `(,special-lambda ,variables . ,body) )))
        (define (handler e r)
          (objectify (invoke expander (cdr e)) r) )
        (insert-global! (make-Magic-Keyword name handler) r)
        (objectify #t r) ) ) ) )
```

On peut créer de manière similaire des macros locales. La différence est que ces mots-clés magiques sont insérés en tête de l'environnement de préparation de l'étage courant afin d'avoir une portée locale.

```
(define (special-let-abbreviation level)
  (lambda (e r)
    (let ((level  (force level))
          (macros (cadr e))
          (body   (cddr e)) )
      (define (make-macro def)
```

```
    (let* ((call      (car def))
           (body      (cdr def))
           (name      (car call))
           (variables (cdr call)) )
      (let ((expander ((Evaluator-eval level)
                       `(,special-lambda ,variables . ,body) )))
        (define (handler e r)
          (objectify (invoke expander (cdr e)) r) )
        (make-Magic-Keyword name handler) ) ) )
(objectify `(,special-begin . ,body)
           (r-extend* r (map make-macro macros)) ) ) ) )
```

La plus complexe des macros est `with-aliases`, dont l'effet est multiple. Elle doit capturer le sens d'un certain nombre de mots dans l'expanseur courant ; elle doit lier ces sens à des variables pour l'évaluateur de l'étage supérieur pendant la durée de l'expansion de son corps. C'est cette interaction de portée, de durée et d'étages qui la rend si complexe.

```
(define (special-with-aliases level)
  (lambda (e current-r)
    (let* ((level   (force level))
           (oldr    (Evaluator-Preparation-Environment level))
           (oldsr   (Evaluator-RunTime-Environment level))
           (aliases (cadr e))
           (body    (cddr e)) )
      (let bind ((aliases aliases)
                 (r       oldr)
                 (sr      oldsr) )
        (if (pair? aliases)
            (let* ((variable (car (car aliases)))
                   (word     (cadr (car aliases)))
                   (var      (make-Local-Variable variable #f #f)) )
              (bind (cdr aliases)
                    (r-extend r var)
                    (sr-extend sr var (objectify word current-r)) ) )
            (let ((result 'wait))
              (set-Evaluator-Preparation-Environment! level r)
              (set-Evaluator-RunTime-Environment! level sr)
              (set! result (objectify `(,special-begin . ,body)
                                      current-r ))
              (set-Evaluator-Preparation-Environment! level oldr)
              (set-Evaluator-RunTime-Environment! level oldsr)
              result ) ) ) ) ) )
```

On pourra noter que la modification de l'environnement d'exécution de l'étage supérieur s'accomoderait mieux d'une liaison dynamique afin d'être sûr que le bon environnement d'exécution soit restauré après expansion du corps de la forme `with-aliases`.

Les variables liées par `with-aliases` ont pour valeur des résultats de la fonction `objectify` c'est-à-dire des instances de `Program` ou `Magic-Keyword`. Comme ces instances peuvent apparaître dans le résultat d'une macroexpansion, il faut que la fonction `objectify` sache reconnaître ces cas dans les expressions qu'elle traite. C'est pourquoi la fonction `objectify` commençait par tester si l'expression reçue est un mot-clé magique ou un programme déjà objectifié. **[☞ p. 329]**

9.11.5 Limites

L'hygiène permet de jouer de façon statique avec les mots et les liaisons, de saisir des sens et de les employer dans des contextes où ils n'étaient peut-être pas visibles (ni même utilisables). On peut par exemple écrire :

```
(let ((count 0))
  (with-aliases ((c count))
    (define-abbreviation (tick) c) )
  (tick) )
(let ((count 1)(c 2))
  (tick) )
```

La macro globale `tick` référence la variable locale `count` qui n'est plus visible du second site d'appel de la macro `tick` : on obtient donc une erreur d'un nouveau type : référence à une variable inexistante (et même si une variable de même nom (`count`) figure dans l'environnement d'appel à `tick`) ! Cela est encore plus visible sur la forme « désobjectifiée » équivalente :

```
((LAMBDA (COUNT501)
  (BEGIN #T ;;  (tick) → count501
         COUNT501 ))
 0 )
((LAMBDA (COUNT502 C503) COUNT501) 1 2)
```

La position de `with-aliases` doit être bien choisie, car c'est une sorte de forme `let` pour l'évaluateur de l'étage supérieur. Si l'on écrit :

```
(define-abbreviation (loop . body)
  (with-aliases ((cc call/cc)(lam lambda) (ll let))
    (let ((loop (gensym)))
      `(,cc (,lam (exit) (,ll ,loop () ,@body (,loop)))) ) ) )
```

les captures de sens qui sont prises ici au niveau de la définition de la macro ne sont valables que dans le monde des macros de macros. En effet, `with-aliases` apparaît dans `define-abbreviation` et est donc évaluée dans le monde des macros lorsque l'abréviation `loop` surgit dans le monde normal. La variable `cc` est liée, dans le monde des macros, au sens qu'a le mot `call/cc` dans ce monde. Un appel à la macro `loop` conduira à une erreur du type "`cc : variable inconnue`".

La capture du sens des variables locales ou globales (à l'exception des mots-clés de formes spéciales ou autres mots-clés comme `else` ou `=>`) aurait pu être effectuée par le biais d'environnements de première classe et des formes spéciales `import` et `export`. **[☞ p. 281]** Toutefois ici, le mécanisme est plus puissant puisqu'il peut aussi capturer l'essence de mots-clés de formes spéciales et surtout parce qu'il est statique et non dynamique.

Le mécanisme de macros ici décrit permet que soient écrites des macros prédéfinies (que l'utilisateur ne pourrait écrire lui-même) comme, par exemple, des macros créant des variables globales par utilisation directe de la fonction `insert-global!`.

Le système a toutefois encore des imperfections. Si l'utilisateur souhaite parcourir les expressions macroexpansées, il rencontrera au moins deux problèmes. Le premier est que la fonction `expand` n'a pas été rendue visible (une façon de faire serait que l'`expand` d'un niveau soit accessible comme la valeur d'une variable de l'étage supérieur de façon que `(eval-in-abbreviation-world (expand 'ε))` soit identique à ε).

D'autre part, on est obligé de connaître la structure des sous-classes de `Program` si on veut parcourir ces objets.

Un des buts de ce système de macros était de montrer que macroexpansion hygiénique et compilation sont très liées puisqu'elles bénéficient d'un important fond commun. Si l'on retire du code précédent la partie liée à l'évaluation et à l'objectification proprement dites, il ne reste de dépendant spécifiquement de la macroexpansion que la fonction `objectify`, la fonction `objectify-symbol` et les fonctions liées aux quatre macros prédéfinies soit une centaine de lignes, ce qui est très peu.

Un vrai système de macros aurait de nombreux autres détails à régler *(i)* programmer les syntaxes essentielles comme `or`, `and`, `letrec`, les `define` internes etc. ; *(ii)* programmer la notation *backquote* qui doit entrelacer ses calculs avec l'objectification-expansion ; *(iii)* autoriser les macro-symboles afin de résoudre le problème de `define-handy-method` vu plus haut. **[☞ p. 324]** Son but est toutefois atteint qui était d'introduire la notion de capture d'intention.

9.12 Conclusions

On peut résumer le problème des macros en deux points : elles sont indispensables mais il n'y a pas deux systèmes pareils. Il existe en effet un grand nombre d'implantations possibles et divergentes dont nous avons essayé de baliser l'espace. Peu de manuels de référence de systèmes Lisp ou Scheme indiquent précisément quel est le modèle exact de macros qu'ils procurent mais il faut avouer, à leur décharge, que ce chapitre est probablement le premier document tentant de décrire l'immense variété des comportements possibles.

9.13 Exercices

Exercice 9.1 : Définissez la macro `repeat` donnée en tout début de chapitre et rendez-la hygiénique.

Exercice 9.2 : Définissez, avec `define-syntax`, une macro prenant en argument une séquence d'expressions et imprimant entre elles leur numéro d'ordre. Par exemple, `(enumerate` π_1 π_2 `...`π_n`)` s'expansera en quelque chose qui, évalué, imprimera 0, puis calculera π_1, imprimera 1, calculera π_2, etc.

Exercice 9.3 : Modifiez le système de macros pour implanter la variante correspondant au monde uniquement unique.

Exercice 9.4 : Écrivez une fonction convertissant une instance de `Program` en une Sexpression équivalente.

Exercice 9.5 : Discernez dans les programmes composant la définition de MEROONET **[☞ p. 395]** ce qui relève de la bibliothèque d'expansion et/ou d'exécution.

9.14 Lectures recommandées

On pourra trouver en [Gra93] une très intéressante apologie des macros. On pourra aussi consulter des articles plus théoriques comme [QP90] ou plutôt centrés sur les problèmes d'expansion hygiénique comme [KFFD86, DFH88, CR91a, QP91b]. Enfin mentionnons un nouveau et prometteur modèle d'expansion [dM95].

10
Compilation vers C

Encore une fois, ce chapitre traitera de compilation, mais avec de nouvelles techniques (et notamment celle des environnements plats) et surtout un nouveau langage cible : C. Il abordera quelques-uns des problèmes de ce mariage qui, s'il apporte quelques joies comme l'obtention gratuite d'optimisations de compilation de bas niveau ou la libre disposition d'immenses bibliothèques, procure néanmoins quelques désagréments comme la non-garantie de la récursion terminale ou la difficulté de couplage avec le GC.

Compiler vers un langage de haut niveau tel que C est intéressant à plus d'un titre : le langage cible étant plus riche, on peut espérer une traduction ressemblant plus à l'original qu'à un vague salmigondis linéaire. Comme C existe sur pratiquement toutes les machines, le code produit a des chances d'être portable. De plus, toutes les optimisations que peut procurer un tel compilateur sont automatiquement et implicitement disponibles. C'est particulièrement important dans le cas de C qui possède des compilateurs effectuant de nombreuses optimisations sur l'allocation de registres, le réordonnancement de code ou le choix de modes d'adressage élaborés, toutes choses que l'on peut donc ignorer pour ne se concentrer que sur le seul langage source. D'un autre côté, un langage de haut niveau comme langage cible apporte certaines contraintes philosophiques et pragmatiques. Il est souvent conçu pour un certain type ou style de programmes sans suspecter que ces programmes puissent être engendrés par d'autres programmes. En conséquence, des limites comme *(i)* 32 arguments au plus dans les appels fonctionnels, *(ii)* moins de 16 niveaux de blocs lexicaux, etc., si elles sont presque supportables pour des utilisateurs normaux, ne le sont plus dès lors qu'il s'agit de programmes mécaniquement engendrés. Il n'est pas rare qu'un programme doté de quelques macros voie sa taille multipliée par 5, 10 ou 20 quand il est traduit en C, ce qui peut poser quelques problèmes au compilateur visé peu habitué à de tels monstres. D'autre part, le modèle d'exécution du langage cible peut n'entretenir que peu de rapports avec celui du langage source, ce qui peut limiter ou compliquer la traduction. C a été conçu comme le langage d'écriture d'un système d'exploitation (UN⋆X) et s'est donc muni d'une politique de gestion explicite de la mémoire qui permet tous les excès et notamment des pointeurs fous (au sens d'un mécanisme dont on a perdu le contrôle), ce qui n'est pas le cas de Lisp qui est un langage sûr. D'autre part, C ne favorise pas l'écriture de programmes fonctionnels ou récursifs car l'appel

de fonction y est notoirement dispendieux ; sa lenteur dégoûte les programmeurs qui, ne l'employant pas, justifient le peu d'empressement des implanteurs à l'améliorer, confortant par là même la lenteur qui dégoûte les programmeurs, etc.

Quoi qu'il en soit, la compilation vers C est à la mode, en témoignent Kyoto COMMON LISP [YH85] (raffiné depuis en AKCL par William Schelter), WCL [Hen92b], CLICC [Hof93], EcoLisp [Att95], Scheme->C [Bar89], et plus près de nous Sqil [Sén91], Ilog Talk [ILO94] et Bigloo [Ser94]. Ce que nous allons présenter ne rivalise pas avec ces derniers et ne constitue que l'ossature d'un tel compilateur mais cela suffit à montrer un grand nombre de points intéressants. Une solution simple voire triviale eût consisté à changer le générateur de code-octets vu au chapitre 7 pour lui faire engendrer du C. **[☞ p. 213]** Chaque instruction-octet aurait pu être expansée en quelques instructions C appropriées. Toutefois, nous prendrons un nouveau chemin, complet, utilisant la technique des environnements plats déjà mentionnée. **[☞ p. 194]** Ce nouveau compilateur sera partitionné en passes réalisées pour la plupart grâce à l'emploi d'arpenteurs (*code-walker*) spécialisés. Cette dernière technique repose sur un emploi systématique d'objets pour représenter et transformer le code à compiler dont nous pensons qu'il illuminera ce dernier chapitre par son élégance et son extensibilité.

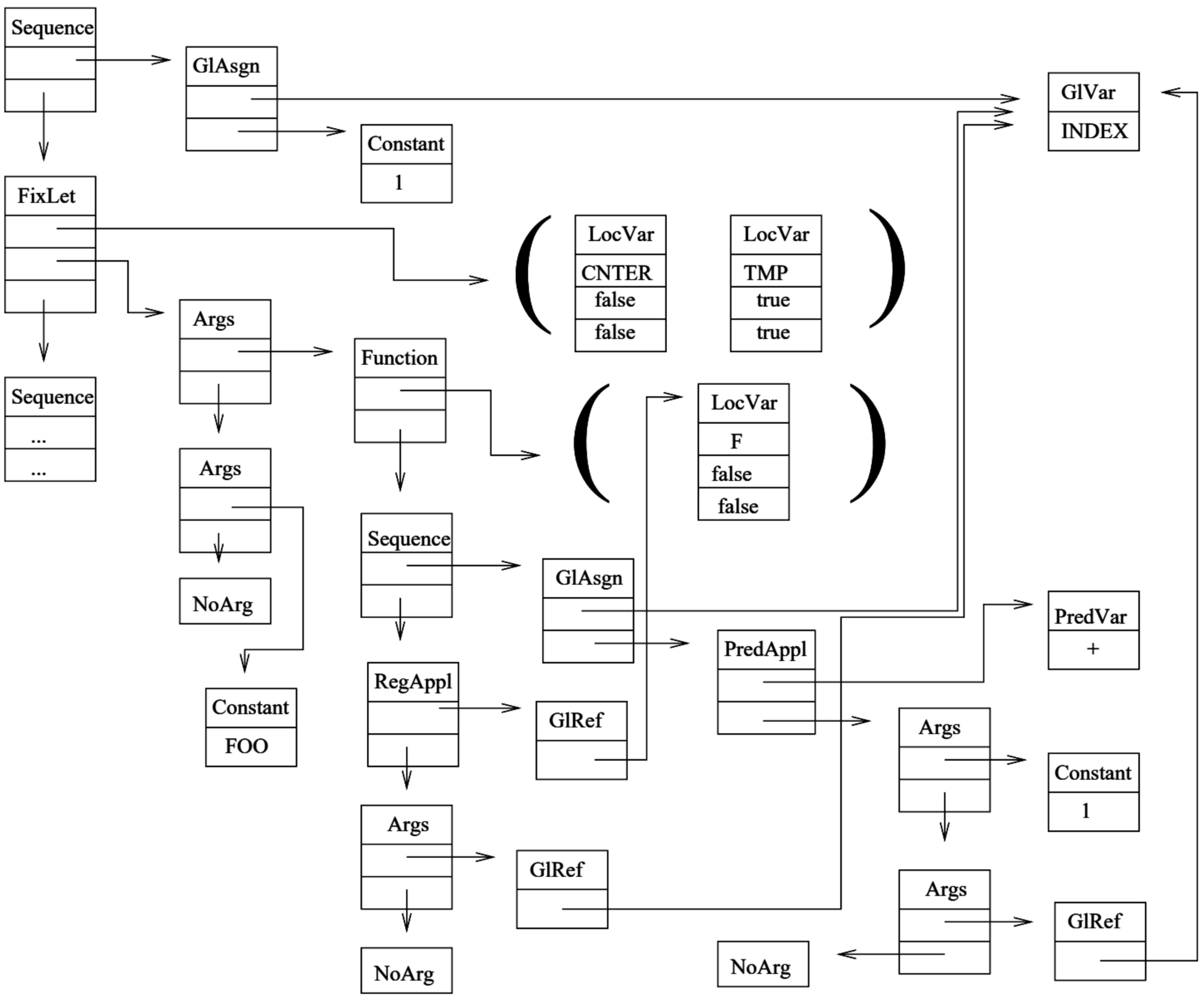


FIGURE 10.1 – Code objectifié

10.1 Objectification

Le convertisseur de programmes en objets a déjà été présenté au paragraphe 9.11.1. **[☞ p. 328]** Il prend soin des éventuelles macroexpansions qui peuvent s'y trouver et retourne une instance de la classe `Program`. Pour pallier le caractère un peu abrupt de la précédente description, qu'il nous soit permis de montrer, à la figure 10.1, un résultat de cette phase d'objectification. Nous utiliserons, pour ce faire, l'expression suivante qui contient au moins une apparition de toutes les facettes étudiées. Les noms des classes ont été abrégés, une partie, seulement, de la figure a été dessinée et les listes sont enserrées entre de gigantesques parenthèses. Cet exemple nous reservira par la suite.

```
(begin
  (set! index 1)
  ((lambda (cnter . tmp)
     (set! tmp (cnter (lambda (i) (lambda x (cons i x)))))
     (if cnter (cnter tmp) index) )
   (lambda (f)
     (set! index (+ 1 index))
     (f index) )
   'foo ) ) → (2 3)
```

10.2 Arpentage

L'arpentage (ou *code walking* [Cur89, Wat93]) est une technique importante qui consiste à parcourir l'arbre représentant le programme à compiler tout en l'enrichissant d'annotations diverses préparant la phase ultime qui est la génération de code. Suivant la nature de ce que l'on cherche à établir, plusieurs schémas de parcours et diverses formes de collecte d'informations peuvent être souhaités : il n'existe pas d'arpenteur universel ! Les évaluateurs de ce livre sont des arpenteurs (un peu) particuliers, de même qu'en sont les fonctions de précompilation telle `meaning` au chapitre sur l'interprétation rapide. **[☞ p. 198]**

Nous n'allons définir qu'une unique fonction d'arpentage modifiant systématiquement l'arbre fourni. Ce sera au demeurant un excellent exemple de *méta-méthode*. La fonction `update-walk!` prend en argument une fonction générique, un objet de classe `Program` et d'éventuels arguments supplémentaires. Elle remplace chaque champ de l'objet contenant une instance de la classe `Program` par le résultat de l'application de la fonction générique à la valeur de ce champ. L'objet initial est retourné en valeur. Pour cela, chaque objet est inspecté ; les champs de sa classe sont tour à tour extraits, vérifiés et récursivement analysés si de la classe `Program`. C'est pour être arpentable que la classe `Arguments` qui représente les arguments d'une `Application`, hérite de la classe `Program` !

```
(define (update-walk! g o . args)
  (for-each (lambda (field)
              (let ((vf (field-value o field)))
                (when (Program? vf)
                  (let ((v (if (null? args) (g vf)
                               (apply g vf args) )))
                    (set-field-value! o v field) ) ) ) ) )
```

```
                (Class-fields (object->class o)) )
    o )
```

La programmation peut sembler simpliste mais est d'une extrême commodité, comme nous allons le prouver dans les sections qui suivent et qui vont métamorphoser notre programme initial avec une remarquable économie de moyens. Il est évident que nombre des passes de transformations que nous allons opérer pourraient être combinées et ainsi accélérées. Nous y avons renoncé afin de mieux séparer les effets de ces transformations.

10.3 Introduction de boîtes

Nous allons supprimer toutes les affectations locales au profit de fonctions opérant sur des boîtes. Cette transformation a déjà été vue et employée dans ce livre. **[☞ p. 108]** Cette transformation va aussi nous servir de premier exemple d'arpentage de code.

Ce que l'on souhaite, c'est remplacer toutes les affectations sur des variables locales par des écritures dans des boîtes. Il faut aussi assurer que la lecture des variables ainsi affectées est aussi transformée en une lecture de boîtes. Compte tenu de l'interface de l'arpenteur, il nous faut fournir une fonction générique effectuant ce travail. Par défaut, cette fonction générique invoque récursivement l'arpenteur sur l'objet ayant servi à la discrimination. C'est cette interaction entre fonction générique et arpenteur qui fait la force de cette union.

```
(define-generic (insert-box! (o Program))
  (update-walk! insert-box! o) )
```

Nous allons introduire trois nouveaux types de nœuds syntaxiques pour représenter les programmes ainsi transformés. Nous les détaillerons lors des transformations qui les nécessiteront.

```
(define-class Box-Read Program (reference))
(define-class Box-Write Program (reference form))
(define-class Box-Creation Program (variable))
```

Fort heureusement pour nous, nous avions pris soin, lors de l'objectification, de repérer toutes les variables locales modifiables grâce à leur champ `mutable?`. Ainsi donc, transformer une lecture de variable modifiable en une instance de `Box-Read` est simple.

```
(define-method (insert-box! (o Local-Reference))
  (if (Local-Variable-mutable? (Local-Reference-variable o))
      (make-Box-Read o)
      o ) )
```

La transformation des affectations est tout aussi aisée. Une affectation locale est transformée en une instance de `Box-Write`. Cependant, compte tenu de la structure de l'algorithme d'arpentage, il ne faut pas oublier d'appeler récursivement l'arpenteur sur la forme fournissant la nouvelle valeur de la variable affectée.

```
(define-method (insert-box! (o Local-Assignment))
  (make-Box-Write (Local-Assignment-reference o)
                  (insert-box! (Local-Assignment-form o)) ) )
```

Une fois tous les accès (en écriture ou lecture) aux variables modifiables transformés pour passer à travers des boîtes, il ne reste plus qu'à créer ces dites boîtes. Les

variables modifiables locales ne peuvent être créées que par des formes `lambda` ou `let`, c'est-à-dire des nœuds `Function` ou `Fix-Let` [1]. La technique est d'insérer en tête du corps de ces formes, une « mise en boîte » appropriée. **[☞ p. 108]** Voici donc comment spécialiser la fonction `insert-box!` sur ces deux types de nœuds introduisant potentiellement des variables locales modifiables. Elles recourrent toutes les deux à une sous-fonction qui insère en tête de leur corps autant d'instances de `Box-Creation` qu'il y a de variables locales modifiables pour lesquelles il s'agit d'allouer des boîtes.

```
(define-method (insert-box! (o Function))
  (set-Function-body!
   o (insert-box!
      (boxify-mutable-variables (Function-body o)
                                (Function-variables o) ) ) )
  o )
(define-method (insert-box! (o Fix-Let))
  (set-Fix-Let-arguments! o (insert-box! (Fix-Let-arguments o)))
  (set-Fix-Let-body!
   o (insert-box!
      (boxify-mutable-variables (Fix-Let-body o)
                                (Fix-Let-variables o) ) ) )
  o )
(define (boxify-mutable-variables form variables)
  (if (pair? variables)
      (if (Local-Variable-mutable? (car variables))
          (make-Sequence
           (make-Box-Creation (car variables))
           (boxify-mutable-variables form (cdr variables)) )
          (boxify-mutable-variables form (cdr variables)) )
      form ) )
```

La transformation est désormais complète et a été spécifiée en seulement quatre méthodes focalisées sur les seules formes importantes pour la transformation. Le résultat peut se contempler à la figure 10.2 qui ne représente que les quelques sous-parties de la figure précédente qui ont changé.

10.4 Élimination des fonctions imbriquées

Le langage C n'autorise pas la définition de fonctions à l'intérieur de fonctions ; autrement dit, les `lambda` à l'intérieur des `lambda` ne peuvent être instantanément traduites. Il va donc falloir éliminer ces cas de figure de manière à ramener le programme à compiler à une simple suite de fonctions closes, c'est-à-dire de fonctions sans variable libre. Encore une fois, la Nature est bonne et a gratifié le genre humain d'une telle transformation. On la nomme `lambda`-*lifting* car elle consiste à faire migrer les formes `lambda` vers l'extérieur de sorte qu'il ne reste plus de `lambda` internes. De nombreuses variantes sont possibles suivant les qualités que l'on désire conserver ou promouvoir [WS94, KH89, CH94].

L'évaluation d'une forme `lambda` conduit à la synthèse d'une fermeture, une sorte d'enregistrement qui enclôt son environnement de définition. Lorsqu'une fermeture est invoquée, une fonction spéciale (que nous avions nommée `invoke`) sait faire en

1. Si l'on n'avait pas identifié les applications réductibles, on aurait eu une méthode de moins à spécifier.

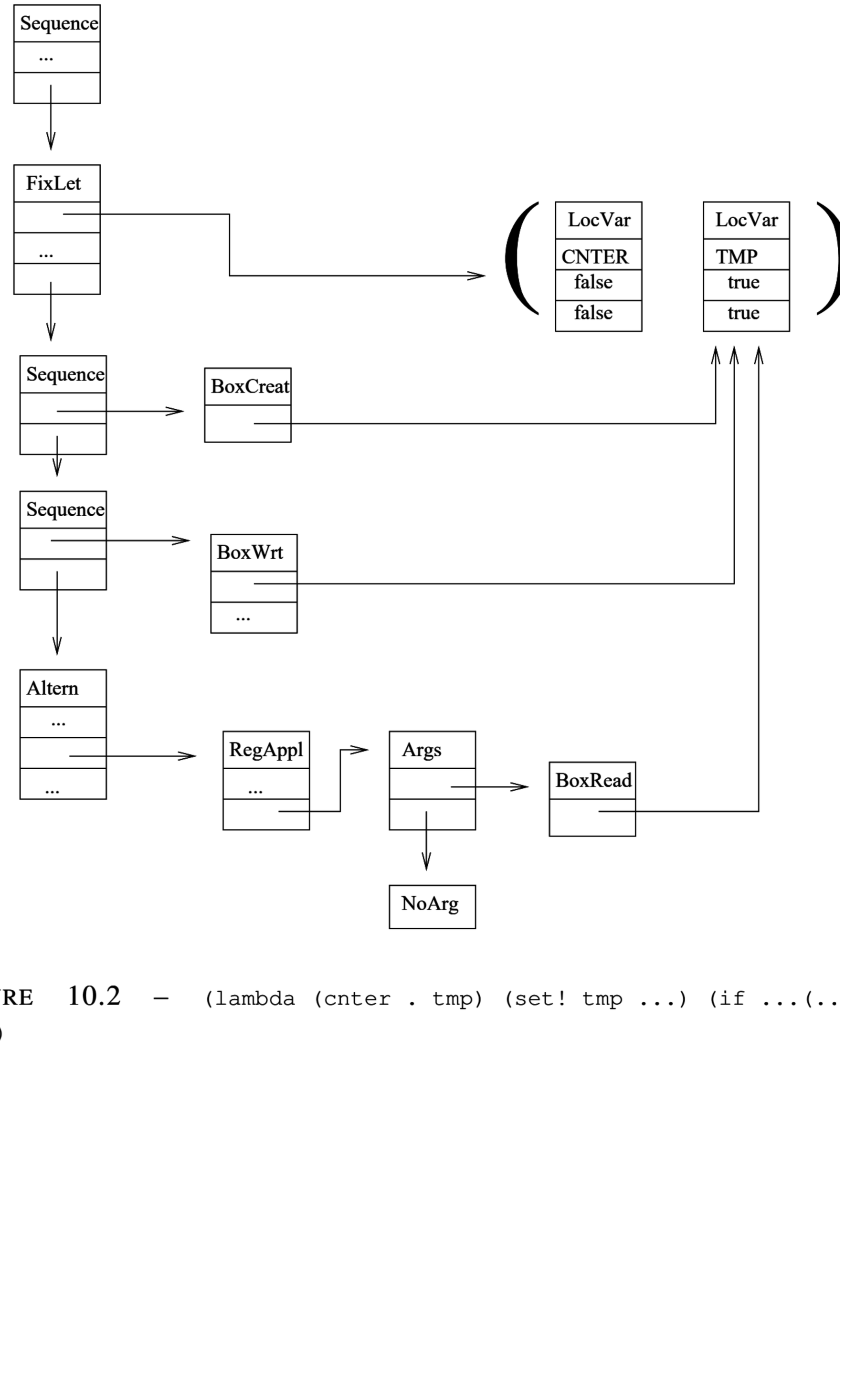


FIGURE 10.2 – `(lambda (cnter . tmp) (set! tmp ...) (if ...(...tmp) ...))`

sorte d'évaluer le corps de cette fermeture tout en lui fournissant les moyens de retrouver les valeurs des variables libres présentes dans ce corps. En fait, seul l'invoqueur sait comment invoquer les fermetures. On peut donc remplacer ces fermetures par des enregistrements sans perturber le reste du programme à condition que toute application fonctionnelle passe par l'invoqueur générique `invoke` qui saura les traiter.

Donnons un petit exemple illustrant la variante intitulée *OO-lifting* en [Que94], une fois de plus sur l'habituel cobaye, la fonction factorielle :

```
(define (fact n k)
  (if (= n 0) (k 1)
      (fact (- n 1) (lambda (r) (k (* n r)))) ) )
```

On peut la traduire pour éliminer la forme `lambda` interne qui enclôt les variables `n` et `k`, en :

```
(define-class Fact-Closure Object (n k))
(define-method (invoke (f Fact-Closure) r)
  (invoke (Fact-Closure-k f) (* (Fact-Closure-n f) r)) )
(define (fact n k)
  (if (= n 0) (invoke k 1)
      (fact (- n 1) (make-Fact-Closure n k)) ) )
```

L'essence de la transformation est de remplacer la fabrication de la fermeture par l'allocation d'un objet (`make-Fact-Closure`) contenant les variables libres du corps de la fermeture, ici `n` et `k`. Une classe particulière est associée à cet objet (ici `Fact-Closure`) afin qu'`invoke` puisse déterminer quelle méthode employer lorsque cet objet est invoqué.

La base de la transformation est de calculer pour chaque abstraction (`Function`) l'ensemble des variables libres présentes dans son corps. Une nouvelle classe raffinant `Function` permettra de stocker cette information : `Flat-Function`. La collection des variables libres sera représentée par des instances de la classe `Free-Environment` et sera terminée par `No-Free`. Ces deux classes sont respectivement semblables mais distinctes des classes `Arguments` et `No-Argument` quoique dérivant de `Program` car représentant des termes évaluables. On identifiera aussi les références à ces variables libres par une classe particulière de référence : `Free-Reference`.

```
(define-class Flat-Function Function (free))
(define-class Free-Environment Program (first others))
(define-class No-Free Program ())
(define-class Free-Reference Reference ())
```

Pour opérer cette transformation, l'arpentage sera encore mis à contribution. La fonction `lift!` sert d'interface au traitement entier. La fonction générique `lift-procedures!` sert d'argument à l'arpenteur ; elle prend elle-même deux arguments supplémentaires qui sont l'abstraction `flatfun` dans laquelle stocker les variables libres ainsi que la liste des variables liées courantes dans la variable `vars`. Par défaut, l'arpenteur s'appelle récursivement sur toutes les instances de `Program`.

```
(define (lift! o)
  (lift-procedures! o #f '()) )
(define-generic (lift-procedures! (o Program) flatfun vars)
  (update-walk! lift-procedures! o flatfun vars) )
```

Trois méthodes seulement seront nécessaires pour la transformation. La première identifie les références aux variables libres, les transforme et les collecte dans l'abstraction courante. La fonction `adjoin-free-variable!` ajoute une variable, si elle n'y est

pas déjà, au sein des variables libres déjà identifiées dans l'abstraction courante.

```
(define-method (lift-procedures! (o Local-Reference) flatfun vars)
  (let ((v (Local-Reference-variable o)))
    (if (memq v vars)
        o (begin (adjoin-free-variable! flatfun o)
                 (make-Free-Reference v) ) ) ) )
(define (adjoin-free-variable! flatfun ref)
  (when (Flat-Function? flatfun)
    (let check ((free* (Flat-Function-free flatfun)))
      (if (No-Free? free*)
          (set-Flat-Function-free!
           flatfun (make-Free-Environment
                    ref (Flat-Function-free flatfun) ) )
          (unless (eq? (Reference-variable ref)
                       (Reference-variable
                        (Free-Environment-first free*) ) )
            (check (Free-Environment-others free*)) ) ) ) ) ) )
```

La forme `Fix-Let` crée de nouvelles liaisons qui peuvent en masquer d'autres. Il faut donc ajouter les variables de `Fix-Let` aux variables liées courantes avant que d'analyser son corps. Il est là très utile de conserver les applications réductibles telles quelles car il serait très coûteux de fabriquer ces fermetures à l'exécution.

```
(define-method (lift-procedures! (o Fix-Let) flatfun vars)
(set-Fix-Let-arguments!
 o (lift-procedures! (Fix-Let-arguments o) flatfun vars) )
(let ((newvars (append (Fix-Let-variables o) vars)))
  (set-Fix-Let-body!
   o (lift-procedures! (Fix-Let-body o) flatfun newvars) )
  o ) )
```

Enfin, le cas le plus complexe est le traitement d'une abstraction. Le corps de l'abstraction est analysé et une instance de `Flat-Function` est allouée afin de servir de réceptacle aux variables libres qui y seront découvertes. Comme les variables libres trouvées peuvent aussi être libres dans l'abstraction englobante, l'arpentage est relancé sur cette liste de variables libres qui a été, fort intelligemment, codée comme une sous-classe de `Program`.

```
(define-method (lift-procedures! (o Function) flatfun vars)
(let* ((localvars (Function-variables o))
       (body   (Function-body o))
       (newfun (make-Flat-Function
                localvars body (make-No-Free) )) )
  (set-Flat-Function-body!
   newfun (lift-procedures! body newfun localvars) )
  (let ((free* (Flat-Function-free newfun)))
    (set-Flat-Function-free!
     newfun (lift-procedures! free* flatfun vars) ) )
  newfun ) )
```

Comme d'habitude, montrons sur l'exemple courant et dans la figure 10.3, l'effet partiel de cette transformation.

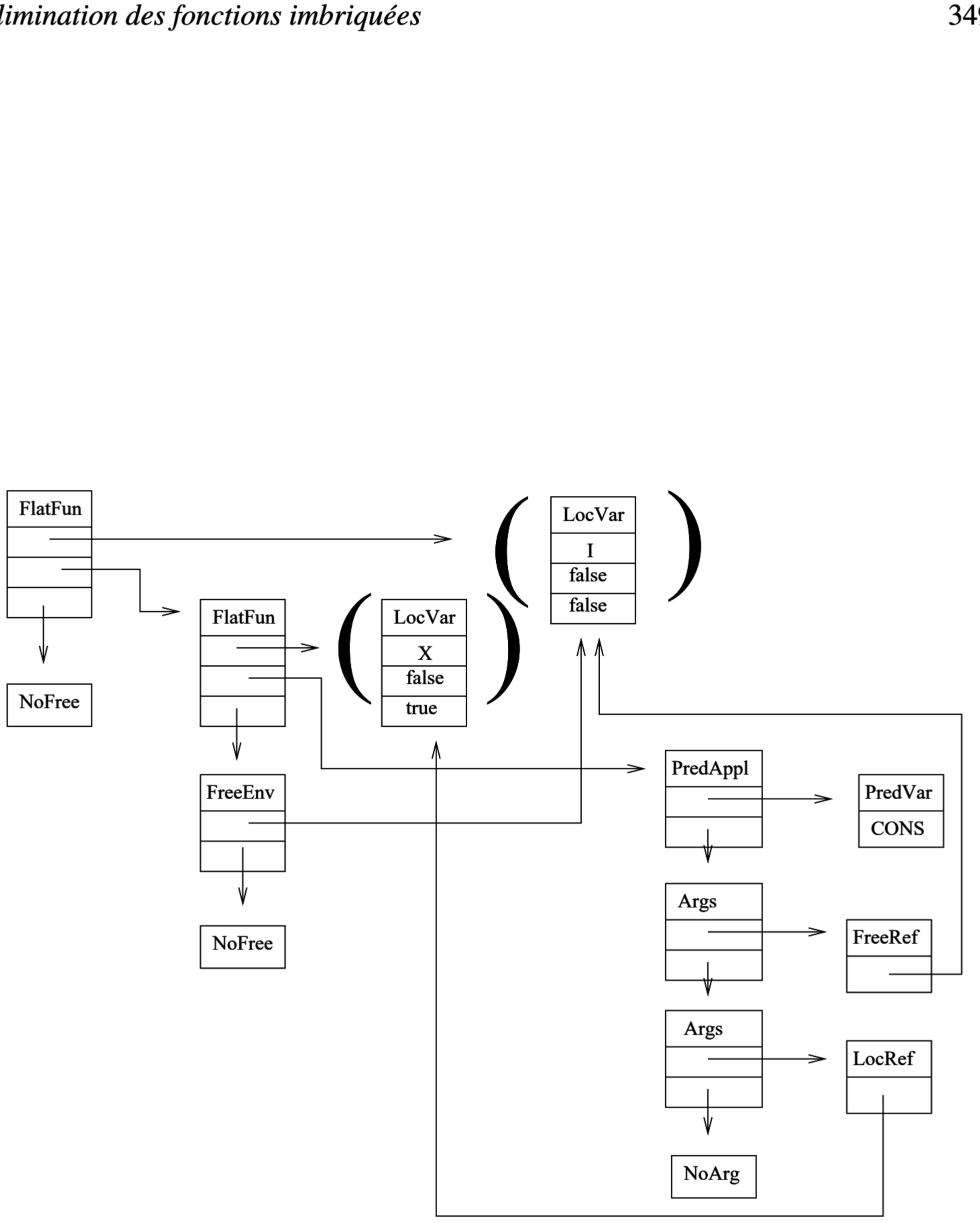


FIGURE 10.3 – `(lambda (i) (lambda x ...))`

10.5 Collecte des citations et des fonctions

La transformation précédente avait laissé les fonctions en place puisque syntaxiquement il y avait encore des `lambda` (mais closes c'est-à-dire sans variable libre) à l'intérieur des `lambda`. Mais c'était temporiser pour mieux sauter ! L'arpentage suivant va extraire d'un programme les citations et les définitions de fonctions afin de les faire apparaître au plus haut niveau. Il ne nous faudra que deux méthodes particulières pour ce faire.

La fonction `extract-things!` opèrera la transformation d'un programme en une instance de `Flattened-Program` qui est une spécialisation de la classe `Program` munie de trois champs supplémentaires : `form` contient le programme à évaluer ; `quotations` est la liste des citations tandis que `definitions` est la liste des définitions de fonctions. Les citations seront effectuées par des références à des variables globales d'une nouvelle classe : `Quotation-Variable`. Les fonctions seront maintenant repérées par leur numéro d'ordre, un simple entier : un `index`. Enfin, la création de fermeture sera traduite par un nouveau nœud syntaxique : `Closure-Creation`. Beaucoup de choses en même temps qui vont sous peu s'éclairer.

```
(define-class Flattened-Program Program
  (form quotations definitions) )
(define-class Quotation-Variable Variable (value))
(define-class Function-Definition Flat-Function (index))
(define-class Closure-Creation Program (index variables free))
```

L'arpentage proprement dit est le résultat de l'interaction entre `extract-things!` et la fonction générique `extract!`. Afin d'éliminer tout recours à une variable globale (par exemple pour compiler deux programmes en parallèle), les résultats de l'arpentage sont insérés dans l'objet final qui est passé en argument supplétif de la fonction générique.

```
(define (extract-things! o)
  (let ((result (make-Flattened-Program o '() '())))
    (set-Flattened-Program-form! result (extract! o result))
    result ) )
(define-generic (extract! (o Program) result)
  (update-walk! extract! o result) )
```

Les citations sont simplement accumulées dans le champ des citations du programme final et remplacées par des références à des variables globales initialisées avec ces citations.

```
(define-method (extract! (o Constant) result)
  (let* ((qv* (Flattened-Program-quotations result))
         (qv  (make-Quotation-Variable (length qv*)
                                       (Constant-value o) )) )
    (set-Flattened-Program-quotations! result (cons qv qv*))
    (make-Global-Reference qv) ) )
```

Quant aux abstractions, on les recherche dans les nœuds de classe `Flat-Function` qui sont dans le même temps transformés en instances de `Closure-Creation`. On pourrait aussi imaginer de partager de mêmes abstractions, c'est-à-dire faire d'`ad-join-definition!` une mémo-fonction. Un point qui peut sembler étrange est de conserver dans la fabrication de la fermeture la liste des variables de l'abstraction originale. Nous la conservons pour faciliter le calcul de l'arité de la fermeture lorsque

nous engendrerons du C.

```
(define-method (extract! (o Flat-Function) result)
  (let* ((newbody   (extract! (Flat-Function-body o) result))
         (variables (Flat-Function-variables o))
         (freevars  (let extract ((free (Flat-Function-free o)))
                      (if (Free-Environment? free)
                          (cons (Reference-variable
                                 (Free-Environment-first free) )
                                (extract
                                 (Free-Environment-others free) ) )
                          '() ) ))
         (index (adjoin-definition!
                 result variables newbody freevars )) )
    (make-Closure-Creation
     index variables (Flat-Function-free o) ) ) )
(define (adjoin-definition! result variables body free)
  (let* ((definitions (Flattened-Program-definitions result))
         (newindex (length definitions)) )
    (set-Flattened-Program-definitions!
     result (cons (make-Function-Definition
                   variables body free newindex )
                  definitions ) )
    newindex ) )
```

Toujours pour illustrer les résultats de cette transformation, on trouvera quelques extraits choisis de notre exemple à la figure 10.4.

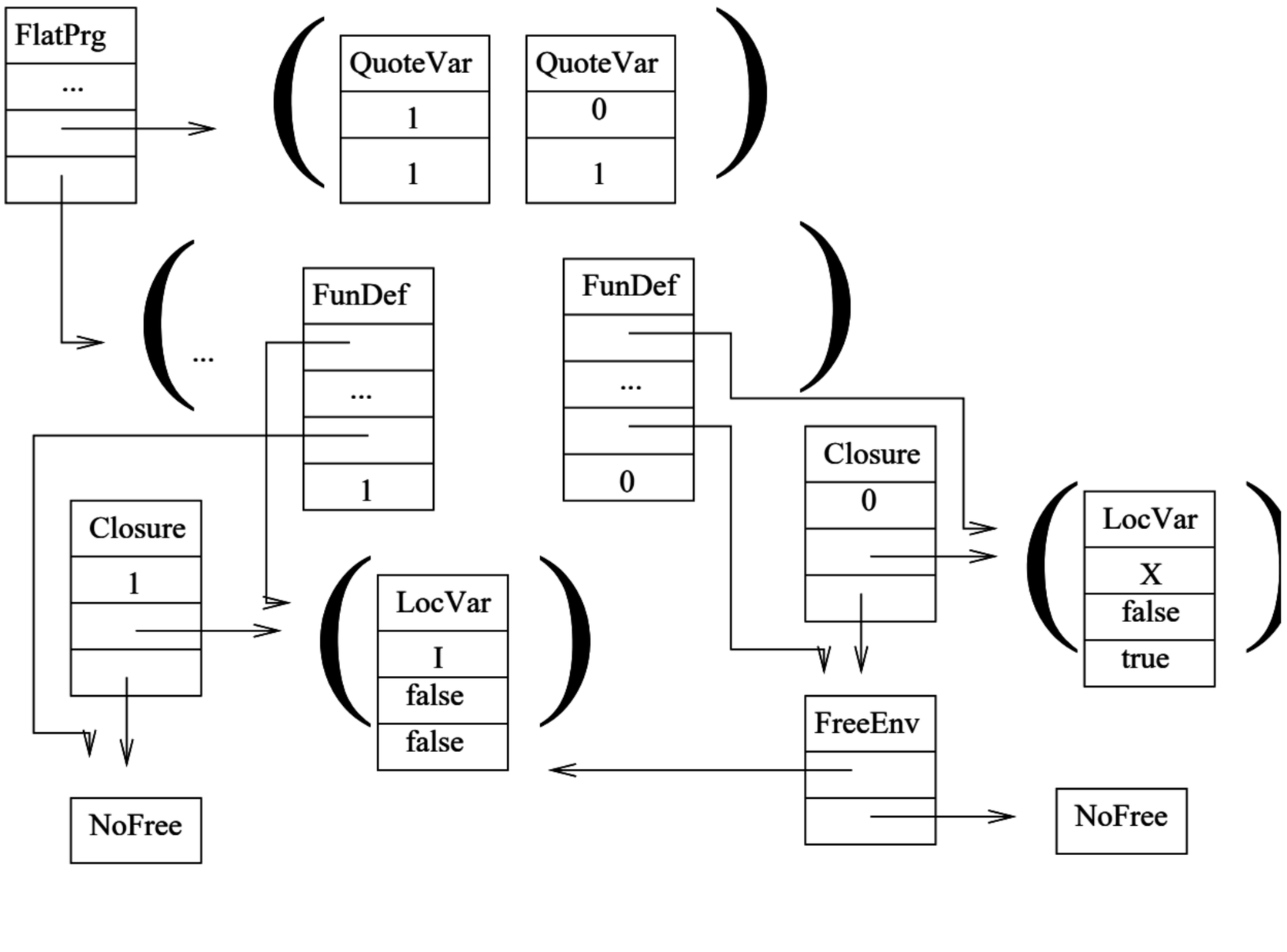


FIGURE 10.4 – `(begin (set! ...) ((lambda ...) ...))`

Enfin, et ce n'est utile que pour un peu simplifier la fonction `main` du C que nous engendrerons ainsi que la prochaine transformation qui s'appliquera uniformément à toutes les définitions fonctionnelles, nous allons convertir le programme entier en l'application d'une fermeture zéroadique : π est transformé en `((lambda () π))`.

```
(define (closurize-main! o)
```

```
(let ((index (length (Flattened-Program-definitions o))))
  (set-Flattened-Program-definitions!
   o (cons (make-Function-Definition
            '() (Flattened-Program-form o) '() index )
           (Flattened-Program-definitions o) ) )
  (set-Flattened-Program-form!
   o (make-Regular-Application
      (make-Closure-Creation index '() (make-No-Free))
      (make-No-Argument) ) )
  o ) )
```

10.6 Collecte de variables temporaires

Nous avons un avantage décisif par rapport au lecteur car nous savons déjà ce que nous cherchons à engendrer. Par souci d'originalité, nous avons choisi de convertir les expressions de Scheme en expressions de C. Ce choix peut paraître bizarre car C est un langage plutôt porté sur les instructions. Notre choix permet toutefois de mieux respecter la structure de Scheme. Un problème alors est que les nœuds de type `Fix-Let` ne peuvent être traduits en C, car C ne possède pas d'expressions permettant d'introduire des variables locales [2]. Notre solution passera par la collecte de toutes les variables locales des `Fix-Let` qui seront introduites fonction C par fonction C. Ce point s'éclairera bientôt. **[☞ p. 364]**

Nous allons donc introduire un nouvel arpentage dont le but est de recenser (et de renommer) toutes les variables locales des nœuds de type `Fix-Let` internes. Cette opération consistant à supprimer d'éventuels conflits de noms est une α-conversion (ou *alphatization*). Les variables temporaires seront accumulées dans une nouvelle spécialisation de `Function-Definition`, la classe `With-Temp-Function-Definition`.

```
(define-class With-Temp-Function-Definition Function-Definition
  (temporaries) )
```

La fonction `gather-temporaries!` met en œuvre la transformation. Elle utilisera la fonction générique `collect-temporaries!` de concert avec l'arpenteur. Le second argument de cette fonction générique sera l'endroit où stocker les variables locales temporaires tandis que le troisième argument mémorise la liste associant les anciens noms des variables avec leur nouveau nom afin que les renommages puissent être effectués.

```
(define (gather-temporaries! o)
  (set-Flattened-Program-definitions!
   o (map (lambda (def)
            (let ((flatfun (make-With-Temp-Function-Definition
                            (Function-Definition-variables def)
                            (Function-Definition-body def)
                            (Function-Definition-free def)
                            (Function-Definition-index def)
                            '() )))
              (collect-temporaries! flatfun flatfun '()) ) )
          (Flattened-Program-definitions o) ) )
  o )
(define-generic (collect-temporaries! (o Program) flatfun r)
```

2. `gcc` toutefois procure une telle extension sous la forme syntaxique `({ ... })`.

```
  (update-walk! collect-temporaries! o flatfun r) )
```

Trois méthodes suffiront à combler nos vœux. Les références locales sont renommées, si besoin est, mais il ne faut point omettre aussi de renommer les variables mises en boîtes.

```
(define-method (collect-temporaries! (o Local-Reference) flatfun r)
  (let* ((variable (Local-Reference-variable o))
         (v (assq variable r)) )
    (if (pair? v) (make-Local-Reference (cdr v)) o) ) )
(define-method (collect-temporaries! (o Box-Creation) flatfun r)
  (let* ((variable (Box-Creation-variable o))
         (v (assq variable r)) )
    (if (pair? v) (make-Box-Creation (cdr v)) o) ) )
```

La méthode la plus complexe est, bien sûr, celle portant sur `Fix-Let`. Elle s'invoque récursivement sur ses arguments puis renomme ses variables locales grâce à la fonction `new-renamed-variable`, ajoute ces nouvelles variables à la définition fonctionnelle courante et enfin s'invoque récursivement sur son corps dans le nouvel environnement de substitution approprié.

```
(define-method (collect-temporaries! (o Fix-Let) flatfun r)
  (set-Fix-Let-arguments!
   o (collect-temporaries! (Fix-Let-arguments o) flatfun r) )
  (let* ((newvars (map new-renamed-variable
                       (Fix-Let-variables o) ))
         (newr (append (map cons (Fix-Let-variables o) newvars) r)) )
    (adjoin-temporary-variables! flatfun newvars)
    (set-Fix-Let-variables! o newvars)
    (set-Fix-Let-body!
     o (collect-temporaries! (Fix-Let-body o) flatfun newr) )
    o ) )
(define (adjoin-temporary-variables! flatfun newvars)
  (let adjoin ((temps (With-Temp-Function-Definition-temporaries
                       flatfun ))
               (vars newvars) )
    (if (pair? vars)
        (if (memq (car vars) temps)
            (adjoin temps (cdr vars))
            (adjoin (cons (car vars) temps) (cdr vars)) )
        (set-With-Temp-Function-Definition-temporaries!
         flatfun temps ) ) ) )
```

Le renommage des variables s'appuie sur une nouvelle classe de variables et sur un compteur les numérotant séquentiellement.

```
(define-class Renamed-Local-Variable Variable (index))
(define renaming-variables-counter 0)
(define-generic (new-renamed-variable (variable)))
(define-method (new-renamed-variable (variable Local-Variable))
  (set! renaming-variables-counter (+ renaming-variables-counter 1))
  (make-Renamed-Local-Variable
   (Variable-name variable) renaming-variables-counter ) )
```

10.6.1 Point d'orgue

L'expression Scheme suivante tente de suggérer textuellement le résultat final des métamorphoses qu'a subies notre exemple courant, c'est-à-dire :

1. l'introduction de boîtes ;
2. l'élimination des fonctions imbriquées ;
3. la collecte des citations et des fonctions ;
4. le ramassage des variables temporaires.

```
(define quote_5 'foo)                          ; citation collectée
(define-class Closure_0 Object ())             ; abstraction (lambda (f) ...)
(define-method (invoke (self Closure_0) f)
  (begin
    (set! index (+ 1 index))
    (invoke f index) ) )                       ; appel calculé
(define-class Closure_1 Object (i))            ; abstraction (lambda x ...)
(define-method (invoke (self Closure_1) . x)
  (cons (Closure_1-i self)                     ; variable close  i
        x ) )
(define-class Closure_2 Object ())             ; abstraction (lambda (i) ...)
(define-method (invoke (self Closure_2) i)
  (make-Closure_1 i) )                         ; allocation de fermeture
(define-class Closure_3 Object ())             ; abstraction (lambda () programme)
(define-method (invoke (self Closure_3))
  ((lambda (cnter_1 tmp_2)                     ; renommage variables locales
     (set! index 1)
     (set! cnter_1 (make-Closure_0))           ; initialisation variable temporaire
     (set! tmp_2 (cons quote_5 '()))
     (set! tmp_2 (make-box tmp_2))             ; mise en boîte
     (box-write! tmp_2                         ; variable modifiable
                 (invoke cnter_1 (make-Closure_2)) )
     (if cnter_1 (invoke cnter_1 (box-read tmp_2)) index) ) ) )
(invoke (make-Closure_3))                      ; invocation programme principal
```

10.7 Génération de C

Nous allons enfin aborder les rivages enchantés de la génération de code et en l'occurrence de code C. Tout est désormais prêt, même si de nombreuses explications sont encore nécessaires. L'auteur n'est pas un maître en C et doit ses connaissances en la matière à la lecture de [ISO90, HS91, CEK+89]. Il suppose que vous en avez vous aussi entendu parler et ne s'appesantira pas sur ce que c'est exactement.

L'arbre syntaxique abstrait est complet et n'attend plus que d'être décompilé, comme par un imprimeur embellissant (*pretty-printer*), vers le langage C. La génération est simple vue de haut niveau. La fonction `compile->C` prend une Sexpression, y applique (dans le bon ordre) l'ensemble des transformations ci-dessus décrites, enfin engendre le programme équivalent sur le port de sortie `out`.

```
(define (compile->C e out)
  (set! g.current '())
  (let ((prg (extract-things! (lift! (Sexp->object e)))))
    (gather-temporaries! (closurize-main! prg))
    (generate-C-program out e prg) ) )
(define (generate-C-program out e prg)
  (generate-header out e)
  (generate-global-environment out g.current)
  (generate-quotations out (Flattened-Program-quotations prg))
```

```
(generate-functions out (Flattened-Program-definitions prg))
(generate-main out (Flattened-Program-form prg))
(generate-trailer out)
prg )
```

Comme tout programme C et plus généralement comme beaucoup d'êtres animés, celui-ci comporte un début, un milieu et une fin. Afin de laisser une trace, l'expression compilée est imprimée joliment (par la fonction `pp` qui n'est pas standard en Scheme) en commentaire. Une directive pour le préprocesseur de C, permet d'inclure un fichier d'en-tête standard `scheme.h` qui définit tout ce qui sera nécessaire pour le reste de la compilation. Nous emploierons aussi à partir de maintenant la fonction usuelle mais non standard `format` pour imprimer du code C.

```
(define (generate-header out e)
  (format out "/* Compiler to C $Revision: 4.3 $ ~%")
  (pp e out)                                  ; DEBUG
  (format out "  */~%~%#include \"scheme.h\"~%") )
(define (generate-trailer out)
  (format out "~%/* End of generated code. */~%") )
```

Las ! Les choses plus complexes commencent alors . . . Le résultat compilé de notre exemple courant apparaît page 369, il peut être utile de s'y référer avant que de poursuivre la lecture des quelques pages qui suivent.

10.7.1 Environnement global

Les variables globales du programme à compiler tombent dans deux catégories : les variables globales prédéfinies, comme `car` ou `+`, et les variables globales modifiables. Les variables prédéfinies sont supposées immuables et apportées par une bibliothèque de fonctions qui sera reliée au programme pour en faire un exécutable. On doit par contre engendrer l'environnement global modifiable, c'est-à-dire exploiter le contenu de `g.current`, où ont été accumulées les variables globales modifiables qui apparaissaient libres dans le programme soumis au compilateur.

La génération à laquelle nous allons procéder dépend du fait que nous compilons un programme entier et non un fragment de programme en vue d'une compilation séparée (qui pose d'autres problèmes que nous n'aborderons point ici, faute de place). **[☞ p. 248]**

Afin de simplifier le code C engendré, nous utiliserons des macros en C afin de rendre ce code plus lisible. Par exemple, la déclaration d'une variable globale s'effectuera en C grâce à la macro `SCM_DefineGlobalVariable`. Le second argument de cette macro est la chaîne de caractères correspondant au nom original en Scheme qui peut être utilisée pour fournir des informations à la mise au point [3].

```
(define (generate-global-environment out gv*)
  (when (pair? gv*)
    (format out "~%/* Global environment: */~%")
    (for-each (lambda (gv) (generate-global-variable out gv))
              gv* )) )
(define (generate-global-variable out gv)
  (let ((name (Global-Variable-name gv)))
```

3. On remarquera que la fonction `read` utilisée pour lire les exemples à compiler dans ce chapitre convertit tous les noms des symboles en majuscules.

```
(format out "SCM_DefineGlobalVariable(~A,\"~A\");~%"
        (IdScheme->IdC name) name ) ) )
```

Nous identifions là un premier problème : les identificateurs de Scheme ne sont pas tous des identificateurs légaux pour C. C'est l'objet de la fonction `IdScheme->IdC` que d'effectuer cette conversion (ou *mangling*). Le principal problème est d'éliminer les caractères illégaux tout en assurant un schéma de traduction de nom qui soit relativement inversible de manière, lorsque l'on rencontre un identificateur en C, que l'on puisse supputer ce que fut le nom original de la variable en Scheme. Notre solution (d'autres existent) est de traduire les caractères saugrenus en caractères normaux et de vérifier si le nom finalement obtenu n'a pas déjà été obtenu afin d'éviter les doublons. Ce code n'est pas très intéressant mais, pour ne rien dissimuler, voici le détail de ces fonctions. La variable `Scheme->C-names-mapping` mémorise toutes les traductions de noms. Elle est prédéfinie avec quelques traductions imposées. D'autres peuvent y être ajoutées, que nous omettons ici.

```
(define Scheme->C-names-mapping
  '( (*         . "TIMES")
     (<         . "LESSP")
     (pair?     . "CONSP")
     (set-cdr!  . "RPLACD")
     ) )
(define (IdScheme->IdC name)
  (let ((v (assq name Scheme->C-names-mapping)))
    (if (pair? v) (cdr v)
        (let ((str (symbol->string name)))
          (let retry ((Cname (compute-Cname str)))
            (if (Cname-clash? Cname Scheme->C-names-mapping)
                (retry (compute-another-Cname str))
                (begin (set! Scheme->C-names-mapping
                             (cons (cons name Cname)
                                   Scheme->C-names-mapping ) )
                       Cname ) ) ) ) ) ) )
```

Lorsqu'il y a conflit, on synthétise un nouveau nom en suffixant le nom original par un indice toujours croissant.

```
(define (Cname-clash? Cname mapping)
  (let check ((mapping mapping))
    (and (pair? mapping)
         (or (string=? Cname (cdr (car mapping)))
             (check (cdr mapping)) ) ) ) )
(define compute-another-Cname
  (let ((counter 1))
    (lambda (str)
      (set! counter (+ 1 counter))
      (compute-Cname (format #f "~A_~A" str counter)) ) ) )
(define (compute-Cname str)
  (define (mapcan f l)
    (if (pair? l)
        (append (f (car l)) (mapcan f (cdr l)))
        '() ) )
  (define (convert-char char)
    (case char
      ((#\_)              '(#\_ #\_))
      ((#\?)              '(#\p))
      ((#\!)              '(#\i))
```

```
      ((#\<)              '(#\l))
      ((#\>)              '(#\g))
      ((#\=)              '(#\e))
      ((#\- #\/ #\* #\:) '())
      (else               (list char)) ) )
  (let ((cname (mapcan convert-char (string->list str))))
    (if (pair? cname) (list->string cname) "weird") ) )
```

On remarquera que dans les noms engendrés ne peut apparaître un blanc souligné (ou *underscore*) isolé. Nous utilisons cette caractéristique pour, par exemple les noms des variables temporaires, sans risque de confusion aucun. On remarquera aussi que ce schéma n'est pas capable de traduire le nom `1+` mais comme il n'est pas imposé que ce dernier soit un identificateur légal en Scheme (quoique pouvant apparaître par macroexpansion) nous ne nous en soucierons pas.

10.7.2 Citations

Les citations doivent être traduites en un fragment de C permettant de les retrouver à l'exécution. Nous innovons ici en présentant un schéma où elles ne seront traduites que par des déclarations de données à l'exclusion de tout code exécutable. On assure aussi que les citations prendront le moins possible de place, c'est-à-dire que les sous-expressions communes auront été identifiées pour être partagées.

La fonction `generate-quotations` assure globalement la traduction des citations. Pour simplifier ce qui suit, nous ignorerons le cas des vecteurs, des grands entiers ou rationnels, des nombres en virgule flottante et des caractères qui se traitent sans problème supplémentaire.

```
(define (generate-quotations out qv*)
  (when (pair? qv*)
    (format out "~%/* Quotations: */~%")
    (scan-quotations out qv* (length qv*) '()) ) )
```

La production proprement dite est le fait de `scan-quotations` qui analyse les valeurs citées telles qu'elles apparaissent dans les instances de `Quotation-Variable`. Chaque fois que c'est possible, ces valeurs sont partagées, ce que détecte le prédicat `already-seen-value?`.

```
(define (scan-quotations out qv* i results)
  (when (pair? qv*)
    (let* ((qv       (car qv*))
           (value    (Quotation-Variable-value qv))
           (other-qv (already-seen-value? value results)) )
      (cond (other-qv
             (generate-quotation-alias out qv other-qv)
             (scan-quotations out (cdr qv*) i (cons qv results)) )
            ((C-value? value)
             (generate-C-value out qv)
             (scan-quotations out (cdr qv*) i (cons qv results)) )
            ((symbol? value)
             (scan-symbol out value qv* i results) )
            ((pair? value)
             (scan-pair out value qv* i results) )
            (else (generate-error "Unhandled constant" qv)) ) ) ) )
(define (already-seen-value? value qv*)
  (and (pair? qv*)
```

```
        (if (equal? value (Quotation-Variable-value (car qv*)))
            (car qv*)
            (already-seen-value? value (cdr qv*)) ) ) )
```

Toutes les citations seront identifiées en C par des lexèmes préfixés par `thing`. Détecter un partage possible, c'est simplement identifier deux lexèmes entre eux ce que l'on réalisera par une macro C que met en œuvre `generate-quotation-alias`. Afin de faciliter la lecture du programme engendré, on fait apparaître en commentaire la valeur partagée.

```
(define (generate-quotation-alias out qv1 qv2)
  (format out "#define thing~A thing~A /* ~S */~%"
          (Quotation-Variable-name qv1)
          (Quotation-Variable-name qv2)
          (Quotation-Variable-value qv2) ) )
```

Lorsque les valeurs sont immédiates (ce que teste `C-value?`), on les traduit sans plus attendre en C (ce que réalise `generate-C-value`). Les valeurs immédiates sont la liste vide, les booléens, les petits entiers et les chaînes de caractères. Toutes ces valeurs sont traduites en les objets C adaptés et l'on admettra pour l'instant qu'il existe des macros C appropriées pour définir des chaînes de caractères, des entiers, les deux booléens et la liste vide. Toutes ces entités C sont préfixées par `SCM_`. Les *afficionados* remarqueront que les valeurs délimitant les petits nombres les restreignent à 16 bits en complément à 1.

```
(define *maximal-fixnum* 16384)
(define *minimal-fixnum* (- *maximal-fixnum*))
(define (C-value? value)
  (or (null? value)
      (boolean? value)
      (and (integer? value)
           (< *minimal-fixnum* value)
           (< value *maximal-fixnum*) )
      (string? value) ) )
(define (generate-C-value out qv)
  (let ((value (Quotation-Variable-value qv))
        (index (Quotation-Variable-name qv)) )
    (cond ((null? value)
           (format out "#define thing~A SCM_nil /* () */~%"
                   index ) )
          ((boolean? value)
           (format out "#define thing~A ~A /* ~S */~%"
                   index (if value "SCM_true" "SCM_false") value ) )
          ((integer? value)
           (format out "#define thing~A SCM_Int2fixnum(~A)~%"
                   index value ) )
          ((string? value)
           (format out "SCM_DefineString(thing~A_object,\"~A\");~%"
                   index value )
           (format out "#define thing~A SCM_Wrap(&thing~A_object)~%"
                   index index ) ) ) ) )
```

Lorsque les valeurs sont composites : paires pointées ou symboles, on les décompose pour déterminer si d'éventuels partages sont possibles, afin de pouvoir les reconstruire à partir de leurs composants. Un symbole n'est reconstruit qu'à partir de la chaîne de caractères qui le nomme. Ces chaînes sont créées préalablement aux symboles.

```
(define (scan-symbol out value qv* i results)
  (let* ((qv    (car qv*))
         (str   (symbol->string value))
         (strqv (already-seen-value? str results)) )
    (cond (strqv (generate-symbol out qv strqv)
                 (scan-quotations out (cdr qv*) i (cons qv results)) )
          (else
           (let ((newqv (make-Quotation-Variable
                         i (symbol->string value) )))
             (scan-quotations out (cons newqv qv*)
                              (+ i 1) results ) ) ) ) ) )

(define (generate-symbol out qv strqv)
  (format out "SCM_DefineSymbol(thing~A_object,thing~A);  /* ~S */~%"
          (Quotation-Variable-name qv)
          (Quotation-Variable-name strqv)
          (Quotation-Variable-value qv) )
  (format out "#define thing~A SCM_Wrap(&thing~A_object)~%"
          (Quotation-Variable-name qv)
          (Quotation-Variable-name qv) ) )
```

En ce qui concerne les paires, on commence par engendrer leur `car` puis leur `cdr` en recherchant les éventuels partages (ce qui peut possiblement survenir entre `car` et `cdr`). La fonction `scan-pair` s'inspire d'un style par continuation.

```
(define (scan-pair out value qv* i results)
  (let* ((qv  (car qv*))
         (d   (cdr value))
         (dqv (already-seen-value? d results)) )
    (if dqv
        (let* ((a   (car value))
               (aqv (already-seen-value? a results)) )
          (if aqv
              (begin
                (generate-pair out qv aqv dqv)
                (scan-quotations out (cdr qv*) i (cons qv results)) )
              (let ((newaqv (make-Quotation-Variable i a)))
                (scan-quotations out (cons newaqv qv*)
                                 (+ i 1) results ) ) ) )
        (let ((newdqv (make-Quotation-Variable i d)))
          (scan-quotations
           out (cons newdqv qv*) (+ i 1) results ) ) ) ) )
(define (generate-pair out qv aqv dqv)
(format out
 "SCM_DefinePair(thing~A_object,thing~A,thing~A); /* ~S */~%"
 (Quotation-Variable-name qv)
 (Quotation-Variable-name aqv)
 (Quotation-Variable-name dqv)
 (Quotation-Variable-value qv) )
(format out "#define thing~A SCM_Wrap(&thing~A_object)~%"
        (Quotation-Variable-name qv)
        (Quotation-Variable-name qv) ) )
```

Nous allons maintenant nous intéresser à un exemple et ainsi détailler la représentation des données.

10.7.3 Déclaration des données

Soit le programme : `(quote ((#F #T) (FOO . "FOO") 33 FOO . "FOO"))`. La compilation telle que nous l'avons ci-dessus décrite, conduira à ce qui suit. Quelques commentaires élucident les valeurs construites. Contemplez !

```
/* Source expression:
'((#F #T) (FOO . "FOO") 33 FOO . "FOO")  */
#include "scheme.h"
/* Quotations */
SCM_DefineString(thing4_object,"FOO");
#define thing4 SCM_Wrap(&thing4_object)
SCM_DefineSymbol(thing5_object,thing4);         /* FOO */
#define thing5 SCM_Wrap(&thing5_object)
SCM_DefinePair(thing3_object,thing5,thing4);    /* (FOO . "FOO") */
#define thing3 SCM_Wrap(&thing3_object)
#define thing6 SCM_Int2fixnum(33)
SCM_DefinePair(thing2_object,thing6,thing3);    /* (33 FOO . "FOO") */
#define thing2 SCM_Wrap(&thing2_object)
SCM_DefinePair(thing1_object,thing3,thing2);
                                /* ((FOO . "FOO") 33 FOO . "FOO") */
#define thing1 SCM_Wrap(&thing1_object)
#define thing9 SCM_nil                          /* () */
#define thing10 SCM_true                        /* #T */
SCM_DefinePair(thing8_object,thing10,thing9);   /* (#T) */
#define thing8 SCM_Wrap(&thing8_object)
#define thing11 SCM_false                       /* #F */
SCM_DefinePair(thing7_object,thing11,thing8);   /* (#F #T) */
#define thing7 SCM_Wrap(&thing7_object)
SCM_DefinePair(thing0_object,thing7,thing1);
                            /* ((#F #T) (FOO . "FOO") 33 FOO . "FOO") */
#define thing0 SCM_Wrap(&thing0_object)
                    ...
```

Le premier objet créé est la chaîne de caractères `"FOO"`. On utilise pour cela la macro `SCM_DefineString` qui prend en premier argument le nom que l'objet portera en C et, en second argument, la dite chaîne de caractères. Similairement, un symbole est créé par la macro `SCM_DefineSymbol` qui prend en premier argument le nom que l'objet portera en C et, en second argument, la chaîne de caractères nommant ce symbole. Toujours similairement, une paire est créée par la macro `SCM_DefinePair` qui prend en premier argument le nom que l'objet portera en C et, en second et troisième arguments, les contenus des champs `car` et `cdr`.

Les objets prédéfinis comme les booléens ou la liste vide ne sont bien sûr pas créés à nouveau mais référencés sous les noms `SCM_true`, `SCM_false` et `SCM_nil`.

Nous n'indiquerons que plus loin **[☞ p. 371]** les représentations exactes des valeurs de Scheme. La compilation est dans une large mesure indépendante de la représentation particulière qui est adoptée. Il nous suffit pour l'heure de savoir que les directives de création `SCM_Define`...n'allouent que des objets et que les valeurs légales les référençant sont obtenues par conversion de leur adresse par `SCM_Wrap`. Quant aux petits entiers, ils font l'objet d'une conversion par `SCM_Int2fixnum`. C'est pourquoi, après toute allocation d'objet, on définit la valeur C le représentant

sous un nom débutant par `thing`. Par exemple, `thing4` désigne la chaîne de caractères `"FOO"` tandis que `thing6` est le petit entier `33`. Plus précisément, `thing4` est le pointeur sur l'objet `thing4_object`, qui est la chaîne de caractères proprement dite.

On remarquera le partage de la chaîne de caractères `"FOO"` et de la Sexpression `(FOO . "FOO")` au sein des objets `thing4` et `thing3`.

10.7.4 Compilation des expressions

La compilation des expressions de Scheme vers C forme bien sûr le gros du compilateur. Là encore nous innoverons car nous traduirons expression vers expression. Nous y gagnerons une certaine clarté du résultat qui respecte mieux la structure du programme source. La perspicacité des choix est cependant un problème bien délicat puisque ce choix est légèrement déviant par rapport à la philosophie de C qui préfère plutôt les instructions aux expressions. Ce n'est pas très gênant sauf lorsqu'il s'agit de mettre au point et d'utiliser, par exemple, `gdb`, le débogueur symbolique de la *Free Software Foundation*. En effet, en mode pas à pas, l'exécution se fait instruction par instruction, ce qui est un grain trop grand pour la génération choisie. Si l'on avait voulu compiler vers des instructions plutôt que vers des expressions, on aurait adopté une technique semblable à celle du compilateur vers du code-octet. **[☞ p. 213]**

La compilation des expressions est effectuée par la fonction générique `->C`. Le premier argument est l'expression à compiler tandis que le second est le port de sortie sur lequel écrire le C résultant. Comme l'on écrit dans un fichier, il faut faire très attention de produire le code séquentiellement sans retour arrière bref, de l'engendrer en une seule passe. **[☞ p. 221]**

```
(define-generic (->C (e Program) out))
```

Contrairement aux précédentes transformations, nous n'allons pas utiliser notre arpenteur générique car il n'y a pas de traitement par défaut mais bien une méthode de génération de code par type de nœud syntaxique. Il ne nous reste donc plus qu'à énumérer ces méthodes. Elles seront assez simples puisque se décalquant assez systématiquement des constructions C équivalentes.

Le langage C a des notions syntaxiques fortes avec précédence, associativité à sens variable, etc. Comme cela est cependant souvent conseillé, nous userons de parenthèses. On y gagnera un air *à la Lisp* de fort bon aloi pour les lispiens et sûrement malencontreux pour les habitués de C. Ce parenthésage sera spécifié par la macro `between-parentheses`.

```
(define-syntax between-parentheses
  (syntax-rules ()
    ((between-parentheses out . body)
     (let ((out out))
       (format out "(")
       (begin . body)
       (format out ")") ) ) ) )
```

Compilation des références aux variables

Plusieurs types de variables existent. En général, nous assimilons une variable Scheme à une variable C. La méthode appropriée sera sous-traitée à la fonction `refe-`

`rence->C` qui, en général, la sous-traite immédiatement à la fonction `variable->C`. Ces indirections permettent de plus facilement spécialiser ces comportements.

```
(define-method (->C (e Reference) out)
  (reference->C (Reference-variable e) out) )
(define-generic (reference->C (v Variable) out))
(define-method (reference->C (v Variable) out)
  (variable->C v out) )
(define-generic (variable->C (variable) out))
```

De façon générale, une variable se traduit par son nom en C, sauf quand elle a été renommée ou qu'elle désigne une citation.

```
(define-method (variable->C (variable Variable) out)
  (format out (IdScheme->IdC (Variable-name variable))) )
(define-method (variable->C (variable Renamed-Local-Variable) out)
  (format out "~A_~A"
          (IdScheme->IdC (Variable-name variable))
          (Renamed-Local-Variable-index variable) ) )
(define-method (variable->C (variable Quotation-Variable) out)
  (format out "thing~A" (Quotation-Variable-name variable)) )
```

Un cas particulier existe cependant pour les variables globales non prédéfinies, les variables libre du programme compilé, dont aucune analyse ne nous a dit si elles étaient initialisées ou pas. On doit donc vérifier cela explicitement grâce à la macro `SCM_CheckedGlobal`. **[☞ Ex. 10.2]**

```
(define-method (reference->C (v Global-Variable) out)
  (format out "SCM_CheckedGlobal")
  (between-parentheses out
    (variable->C v out) ) )
```

Enfin, il reste le cas des variables libres pour lesquelles nous ferons encore une fois appel à une macro appropriée : `SCM_Free`.

```
(define-method (->C (e Free-Reference) out)
  (format out "SCM_Free")
  (between-parentheses out
    (variable->C (Free-Reference-variable e) out) ) )
```

Compilation des affectations

Les affectations se traitent encore plus simplement car il n'y en a que deux types : les affectations de variables globales et les écritures dans les boîtes. L'affectation d'une variable globale se traduit par l'affectation en C de la variable globale correspondante.

```
(define-method (->C (e Global-Assignment) out)
  (between-parentheses out
    (variable->C (Global-Assignment-variable e) out)
    (format out "=")
    (->C (Global-Assignment-form e) out) ) )
```

Compilation des boîtes

En ce qui concerne les boîtes, il n'y a que trois opérations concernées. Une macro C permet d'accéder, en lecture ou écriture, au contenu d'une boîte : `SCM_Content`,

et une fonction de la bibliothèque prédéfinie, `SCM_allocate_box`, permet d'allouer une boîte lorsque nécessité s'en fait sentir.

```
(define-method (->C (e Box-Read) out)
  (format out "SCM_Content")
  (between-parentheses out
    (->C (Box-Read-reference e) out) ) )
(define-method (->C (e Box-Write) out)
  (between-parentheses out
    (format out "SCM_Content")
    (between-parentheses out
      (->C (Box-Write-reference e) out) )
    (format out "=")
    (->C (Box-Write-form e) out) ) )
(define-method (->C (e Box-Creation) out)
  (variable->C (Box-Creation-variable e) out)
  (format out "= SCM_allocate_box")
  (between-parentheses out
    (variable->C (Box-Creation-variable e) out) ) )
```

Alternative

L'alternative de Scheme se traduit en l'expression alternative de C, à savoir la construction ternaire $\pi_0 ? \pi_1 : \pi_2$. Comme toute valeur différente de Faux est considérée comme Vrai, on teste explicitement ce cas. Bien sûr, on parenthèse sans restriction le tout !

```
(define-method (->C (e Alternative) out)
  (between-parentheses out
    (boolean->C (Alternative-condition e) out)
    (format out "~%? ")
    (->C (Alternative-consequent e) out)
    (format out "~%: ")
    (->C (Alternative-alternant e) out) ) )
(define-generic (boolean->C (e) out)
  (between-parentheses out
    (->C e out)
    (format out " != SCM_false") ) )
```

On voit là une première source d'inefficacité par rapport à un vrai compilateur. Dans le cas d'un prédicat comme `(if (pair? x) ...)`, il est inefficace que l'appel à `pair?` retourne un booléen de Scheme que l'on compare à `SCM_false`, il serait préférable que `pair?` retournât directement un booléen de C. On peut donc spécialiser la fonction `boolean->C` pour reconnaître et traiter particulièrement les appels aux prédicats prédéfinis. On peut aussi unifier les booléens de C et de Scheme en adoptant la convention que Faux est représenté par `NULL`. On peut enfin effectuer une analyse de types (ou *type recovery*) afin de supprimer ces lourdes comparaisons [Shi91, Ser93, WC94].

Séquences

Les séquences sont traduites en séquences de C qui se notent $(\pi_1, \ldots, \pi_n)$.

```
(define-method (->C (e Sequence) out)
  (between-parentheses out
```

```
(->C (Sequence-first e) out)
(format out ",~%")
(->C (Sequence-last e) out) ) )
```

10.7.5 Compilation des applications fonctionnelles

Les applications fonctionnelles avaient été rangées en plusieurs catégories : les applications fonctionnelles normales, les applications fonctionnelles closes (dont le terme fonctionnel est une abstraction), enfin les applications fonctionnelles où la fonction invoquée est une fonction valeur d'une variable prédéfinie. Ces trois types d'application sont matérialisées par trois classes de nœuds syntaxiques : `Fix-Let`, `Regular-Application` et `Predefined-Application`.

Une application fonctionnelle normale $(f\ x_1\ \ldots x_n)$ sera compilée en l'expression C `SCM_invoke(`$f,n,x_1,\ldots,x_n$`)`, où n est le nombre d'arguments passés à la fonction et où `SCM_invoke` est la fonction spécialisée d'invocation. Pour aider la lecture, les petites arités (inférieure à quatre) utiliseront des macros C spécialisées :

```
#define SCM_invoke0(f)       SCM_invoke(f,0)
#define SCM_invoke1(f,x)     SCM_invoke(f,1,x)
#define SCM_invoke2(f,x,y)   SCM_invoke(f,2,x,y)
#define SCM_invoke3(f,x,y,z) SCM_invoke(f,3,x,y,z)
```

Les autres arités utiliseront le protocole direct :

```
(define-method (->C (e Regular-Application) out)
  (let ((n (number-of (Regular-Application-arguments e))))
    (cond ((< n 4)
           (format out "SCM_invoke~A" n)
           (between-parentheses out
             (->C (Regular-Application-function e) out)
             (->C (Regular-Application-arguments e) out) ) )
          (else
           (format out "SCM_invoke")
           (between-parentheses out
             (->C (Regular-Application-function e) out)
             (format out ",~A" n)
             (->C (Regular-Application-arguments e) out) ) ) ) ) )
```

Les applications closes seront traduites en une séquence de C correspondant à leur corps et justifieront a posteriori la collecte de variables temporaires que nous avions précédemment effectuée. Cette collecte est d'ailleurs équivalente à une fusion des blocs temporaires (ou *frame coalescing*), technique de compilation où l'on cherche à limiter le nombre d'allocations au profit de blocs d'activation plus grands [AS94, PJ87]. Si l'on suppose que toutes les variables temporaires des formes `let` ont déjà été allouées, alors la liaison de ces variables à leur valeur n'est plus qu'une affectation. On pourrait traduire cela, en termes lispiens, par la règle de réécriture imagée suivante[4] où l'on fusionne les `let` internes au profit d'un unique `let` englobant associé à des renommages locaux. Cette transformation n'est pas licite dans le cas général car si $\ldots_1$ retourne multiplement et si π_2 capturait la variable `x`, alors cet `x` renommé en `x1` serait partagé. **[☞ p. 181]** La transformation est cependant correcte ici car `x1` n'est

4. On pourrait, bien sûr, chercher à partager encore plus les emplacements réservés aux variables. C'est le cas des variables `x1` et `x2` si aucune confusion n'est à craindre.

pas modifiée (la variable est immuable même si `set!` est présent) et nous utilisons une technique d'environnement plat où les valeurs sont recopiées.

```
(begin ...1                          (let (x1 x2) ...1
  (let ((x π1))                          (set! x1 π1)
    π2 )                                 π2[x→x1]
  ...2                          ⇒        ...2
  (let ((x π3))                          (set! x2 π3)
    π4 ) )                               π4[x→x2] )
```

Nous n'avons donc qu'à affecter chaque variable locale avec sa valeur d'initialisation. Comme toutes les variables locales ont été identifiées et renommées, il n'y a pas de confusion possible, ni de portée bafouée. Une fonction générique particulière, `bindings->C`, s'occupe de traduire les liaisons.

```
(define-method (->C (e Fix-Let) out)
  (between-parentheses out
    (bindings->C (Fix-Let-variables e) (Fix-Let-arguments e) out)
    (->C (Fix-Let-body e) out) ) )
(define-generic (bindings->C variables (arguments) out))
(define-method (bindings->C variables (e Arguments) out)
  (variable->C (car variables) out)
  (format out "=")
  (->C (Arguments-first e) out)
  (format out ",~%")
  (bindings->C (cdr variables) (Arguments-others e) out) )
(define-method (bindings->C variables (e No-Argument) out)
  (format out "") )
```

Enfin le cas des applications fonctionnelles, où la fonction est la valeur d'une variable prédéfinie et donc parfaitement connue, va faire l'objet d'une sorte d'intégration (ou *inlining*) où la fonction sera directement appelée sans l'entremise de `SCM_invoke`. Les fonctions prédéfinies sont écrites en C et figurent dans la bibliothèque qui doit être liée au programme compilé. Les appeler directement est équivalent mais bien sûr plus efficace que de passer par `SCM_invoke`.

Engendrer l'appel direct dépend de l'implantation de la primitive, nous supposerons donc que la description fonctionnelle associée à la variable prédéfinie possède un champ, `generator`, contenant le générateur d'appel approprié. Cette fonction de génération recevra le nœud correspondant à l'application et se chargera d'appeler récursivement `->C` sur les arguments grâce à la fonction générique `arguments->C`.

```
(define-method (->C (e Predefined-Application) out)
  ((Functional-Description-generator
    (Predefined-Variable-description
     (Predefined-Application-variable e) ) ) e out ) )
(define-generic (arguments->C (e) out))
(define-method (arguments->C (e Arguments) out)
  (->C (Arguments-first e) out)
  (->C (Arguments-others e) out) )
(define-method (arguments->C (e No-Argument) out)
  #t )
```

10.7.6 Environnement prédéfini

La compilation des applications à des fonctions prédéfinies impose que le compilateur connaisse ces fonctions. Nous définirons donc une macro `defprimitive` à cet usage.

```
(define-class Functional-Description Object
  (comparator arity generator) )

(define-syntax defprimitive
  (syntax-rules ()
    ((defprimitive name Cname arity)
     (let ((v (make-Predefined-Variable
               'name (make-Functional-Description
                      = arity
                      (make-predefined-application-generator
                       'Cname ) ) )))
       (set! g.init (cons v g.init))
       'name ) ) ) )
```

La définition d'une fonction primitive passe donc par la mention de son nom en Scheme, son nom en C et son arité. Les appels engendrés ont tous la forme d'un appel fonctionnel en C, c'est-à-dire un nom suivi d'une flopée d'arguments entre parenthèses, séparés par des virgules. La fonction `make-predefined-application-generator` est un créateur de tels générateurs.

```
(define (make-predefined-application-generator Cname)
  (lambda (e out)
    (format out "~A" Cname)
    (between-parentheses out
      (arguments->C (Predefined-Application-arguments e) out) ) ) )
```

Donnons quelques exemples, comme les habituelles `cons`, `car`, `+` ou `=`, que voici :

```
(defprimitive cons "SCM_cons" 2)
(defprimitive car "SCM_car" 1)
(defprimitive + "SCM_Plus" 2)
(defprimitive = "SCM_EqnP" 2)
```

On notera, sans toutefois percevoir encore la différence, que `cons` se compile en un appel à la fonction C `SCM_cons` tandis que `+` se compile en un appel à la macro C `SCM_Plus`. **[☞ p. 375]** La capitalisation faisant la différence entre fonctions et macros. Ce *distinguo* ne modifie pas qu'un peu la vitesse d'exécution et la taille du code C résultant.

10.7.7 Compilation des fonctions

Les abstractions ont été remplacées grâce aux transformations précédentes par des allocations de fermetures ou nœuds de classe `Closure-Creation`. Ces fermetures seront toutes allouées par la fonction `SCM_close` figurant dans la bibliothèque d'exécution prédéfinie. Cette fonction prendra en premier argument l'adresse de la fonction C (convenablement typée grâce à la macro C `SCM_CfunctionAddress`) correspondant au corps de l'abstraction, l'arité de la fermeture créée en second argument, suivie du nombre de valeurs closes préfixant ces valeurs mêmes. Les valeurs closes sont tout simplement obtenues par traduction vers C du champ `free` de la description de la fermeture. L'arité des fonctions prenant un nombre fixe d'arguments sera

codée par ce nombre même. En revanche, lorsqu'une fonction a une variable pointée, elle doit au minimum prendre un certain nombre d'arguments, disons i, son arité sera alors représentée par $-i-1$. Ainsi la fonction `list` a une arité de -1.

```
(define-method (->C (e Closure-Creation) out)
  (format out "SCM_close")
  (between-parentheses out
    (format out "SCM_CfunctionAddress(function_~A),~A,~A"
            (Closure-Creation-index e)
            (generate-arity (Closure-Creation-variables e))
            (number-of (Closure-Creation-free e)) )
    (->C (Closure-Creation-free e) out) ) )
(define (generate-arity variables)
  (let count ((variables variables)(arity 0))
    (if (pair? variables)
        (if (Local-Variable-dotted? (car variables))
            (- (+ arity 1))
            (count (cdr variables) (+ 1 arity)) )
        arity ) ) )
(define-method (->C (e No-Free) out)
  #t )
(define-method (->C (e Free-Environment) out)
  (format out ",")
  (->C (Free-environment-first e) out)
  (->C (Free-environment-others e) out) )
```

Les abstractions elles-mêmes ont été regroupées dans l'objet représentant le programme tout entier, une instance de `Flattened-Program`, en une liste de fonctions sans variables libres décrites par des instances de `With-Temp-Function-Definition`. Chacune de ces fonctions conduit à la génération d'une fonction équivalente en C.

```
(define (generate-functions out definitions)
  (format out "~%/* Functions: */~%")
  (for-each (lambda (def)
              (generate-closure-structure out def)
              (generate-possibly-dotted-definition out def) )
            (reverse definitions) ) )
```

Afin de rendre le code engendré plus lisible, nous utiliserons encore des macros de C afin de masquer les détails fins de représentation. Pour chaque fermeture seront engendrées une fonction et une structure de données décrivant où sont placées les variables encloses dans l'objet de type fermeture. Tous les noms qui seront engendrés pour représenter ces objets seront formés du radical `function_` et d'un indice qui figure déjà dans le champ `index` de l'objet `Function-Definition`[5].

La représentation de la fermeture est définie par la macro `SCM_DefineClosure`, dont le premier argument est le nom de la fonction C associée et dont le deuxième argument est la liste des noms des variables capturées et séparés par des points-virgules.

```
(define (generate-closure-structure out definition)
  (format out "SCM_DefineClosure(function_~A, "
          (Function-Definition-index definition) )
  (generate-local-temporaries (Function-Definition-free definition)
                              out )
```

5. Que les fonctions soient nommées `function_`i montrent bien qu'elles sont indépendantes du nom des variables dans lesquelles elles sont stockées. Il serait bien sûr plus agréable que les fonctions associées à des variables globales non modifiables soient nommées d'après elles.

```
(format out ");~%") )
```

La fonction `generate-possibly-dotted-definition` engendre la définition de la fonction C en tenant compte de son arité. La fonction est décrite par la macro `SCM_DeclareFunction` qui prend en argument le nom de la fonction engendrée. Ses variables sont décrites par les macros `SCM_DeclareLocalVariable` ou `SCM_DeclareLocalDottedVariable` qui prennent le nom de la variable et son rang dans la liste des variables. Ce rang n'est important que pour calculer la liste liée à une variable pointée. Le corps de la fonction ne pose pas de problème, il est obtenu par simple application de la fonction `->C` mais est précédé par un `return` nécessaire aux fonctions C.

```
(define (generate-possibly-dotted-definition out definition)
  (format out "~%SCM_DeclareFunction(function_~A) {~%"
          (Function-Definition-index definition) )
  (let ((vars (Function-Definition-variables definition))
        (rank -1) )
    (for-each
     (lambda (v)
       (set! rank (+ rank 1))
       (cond ((Local-Variable-dotted? v)
              (format out "SCM_DeclareLocalDottedVariable(") )
             ((Variable? v)
              (format out "SCM_DeclareLocalVariable(") ) )
       (variable->C v out)
       (format out ",~A);~%" rank) )
     vars )
    (let ((temps (With-Temp-Function-Definition-temporaries
                  definition )))
      (when (pair? temps)
        (generate-local-temporaries temps out)
        (format out "~%") ) )
    (format out "return ")
    (->C (Function-Definition-body definition) out)
    (format out ";~%}~%~%") ) )
```

Les listes de variables sont converties vers C au moyen de la fonction utilitaire `generate-local-temporaries` :

```
(define (generate-local-temporaries temps out)
  (when (pair? temps)
    (format out "SCM ")
    (variable->C (car temps) out)
    (format out "; ")
    (generate-local-temporaries (cdr temps) out) ) )
```

10.7.8 Initialisation du programme

Nous avions à compiler un programme Scheme entier ; il ne nous reste donc plus qu'à indiquer comment former un programme C entier. C'est à cette fin que nous avions procédé à la mise en fermeture de l'expression initiale lors de la phase d'aplatissement des fonctions. **[☞ p. 351]** La seule décision importante et contestable est que nous engendrons un appel à la fonction `SCM_print` qui imprimera la valeur de l'expression compilée. Ce n'est point nécessaire et de nombreux programmes font des choses intéressantes sans pour autant polluer leur flux de sortie (par exemple, `cc`).

C'est simplement plus commode pour les petits programmes que nous compilons.

```
(define (generate-main out form)
  (format out "~%/* Expression: */~%void main(void) {~%")
  (format out "  SCM_print")
  (between-parentheses out
    (->C form out) )
  (format out ";~%  exit(0);~%}~%") )
```

Bien que nous n'ayons encore rien dit de la représentation des données, ni décrit la bibliothèque initiale d'exécution, le compilateur est désormais complet et opérationnel. Nous allons donc pouvoir traduire en C notre exemple courant. Nous nous sommes cependant permis d'user de quelques renfoncements gauches (d'ajouter de belles espaces au fer à gauche) pour en améliorer la lecture. Voici donc la traduction complète de cet exemple qui, d'ailleurs, réapparaît en commentaire dans les lignes 2 à 9. **[☞ p. 343]**

---

o/chap10ex.c

```
/* Compiler to C $Revision: 4.0 $
(BEGIN
  (SET! INDEX 1)
  ((LAMBDA
     (CNTER . TMP)
     (SET! TMP (CNTER (LAMBDA (I) (LAMBDA X (CONS I X)))))
     (IF CNTER (CNTER TMP) INDEX))
   (LAMBDA (F) (SET! INDEX (+ 1 INDEX)) (F INDEX))
   'FOO))  */
#include "scheme.h"
/* Global environment: */
SCM_DefineGlobalVariable(INDEX,"INDEX");
/* Quotations: */
#define thing3 SCM_nil /* () */
SCM_DefineString(thing4_object,"FOO");
#define thing4 SCM_Wrap(&thing4_object)
SCM_DefineSymbol(thing2_object,thing4);        /* FOO */
#define thing2 SCM_Wrap(&thing2_object)
#define thing1 SCM_Int2fixnum(1)
#define thing0 thing1 /* 1 */
/* Functions: */
SCM_DefineClosure(function_0, );
SCM_DeclareFunction(function_0) {
  SCM_DeclareLocalVariable(F,0);
  return ((INDEX=SCM_Plus(thing1,
                          SCM_CheckedGlobal(INDEX))),
          SCM_invoke1(F,
                      SCM_CheckedGlobal(INDEX)));
}
SCM_DefineClosure(function_1, SCM I; );
SCM_DeclareFunction(function_1) {
  SCM_DeclareLocalDottedVariable(X,0);
  return SCM_cons(SCM_Free(I),
                  X);
}
SCM_DefineClosure(function_2, );
SCM_DeclareFunction(function_2) {
  SCM_DeclareLocalVariable(I,0);
```

```
  return SCM_close(SCM_CfunctionAddress(function_1),-1,1,I);
}
SCM_DefineClosure(function_3, );
SCM_DeclareFunction(function_3) {
  SCM TMP_2; SCM CNTER_1;
  return ((INDEX=thing0),
          (CNTER_1=SCM_close(SCM_CfunctionAddress(function_0),1,0),
           TMP_2=SCM_cons(thing2,
                          thing3),
           (TMP_2= SCM_allocate_box(TMP_2),
            ((SCM_Content(TMP_2)=
              SCM_invoke1(CNTER_1,SCM_close(SCM_CfunctionAddress
                                            (function_2),1,0))),
             ((CNTER_1 != SCM_false)
              ? SCM_invoke1(CNTER_1,
                            SCM_Content(TMP_2))
              : SCM_CheckedGlobal(INDEX))))));
}
/* Expression: */
void main(void) {
  SCM_print(SCM_invoke0(SCM_close(SCM_CfunctionAddress(function_3),
                                  0,0)));
  exit(0);
}
/* End of generated code. */
```

Nous en donnons aussi la forme expansée, c'est-à-dire débarrassée de ses macros C (à l'exception des `va_list` que nous ne saurions voir), des fonctions `function_1` et `function_2`. Les autres expansions sont du même tonneau.

```
struct function_1 {
SCM(*behavior) (void);
long            arity;
SCM             I;
};
SCM function_1(struct function_1 * self_,
           unsigned long size_,
           va_list arguments_)
{
SCM             X = SCM_list(size_ - 0, arguments_);
return SCM_cons(((*self_).I), X);
}
struct function_2 {
SCM(*behavior) (void);
long            arity;
};
SCM function_2(struct function_2 * self_,
           unsigned long size_,
           va_list arguments_)
{
SCM             I = va_arg(arguments_, SCM);
return SCM_close(((SCM(*) (void)) function_1), -1, 1, I);
}
```

Compilé par un compilateur conforme à ISO-C, par exemple, `gcc` avec les bibliothèques d'exécution appropriée et décrite *infra*, nous obtiendrons un (très petit)

exécutable produisant (très rapidement [6]) la valeur souhaitée [7].

```
% gcc -ansi -pedantic chap10ex.c scheme.o schemelib.o
% time a.out
(2 3)
0.010u 0.000s 0:00.00 0.0% 3+5k 0+0io 0pf+0w
% size a.out
text    data    bss     dec     hex
28672   4096    32      32800   8020
```

10.8 Représentation des données

Ce n'est pas la moindre des coquetteries du précédent compilateur que d'avoir été présenté indépendamment de la représentation en C qu'ont les valeurs de Scheme. Un jeu de macros C réunies dans le fichier `scheme.h`, voilà cet aspect que nous allons maintenant éclaircir. Point n'est besoin encore de rappeler que le but de cet exercice n'est pas de délivrer le plus performant des compilateurs, mais bien plutôt d'esquisser, de suggérer et de détailler diverses techniques ! Nous ne nous encombrerons pas des problèmes de gestion de mémoire : il n'y aura pas de GC et toutes les allocations utiliseront la fonction `malloc`, ce qui rendra particulièrement simple l'adaptation d'un GC conservatif comme celui de Hans Boehm [BW88].

Les valeurs de Lisp/Scheme sont caractérisées, par le fait qu'il est toujours possible de s'enquérir de leur type : il est donc nécessaire qu'à chaque valeur soit associée cette information. Rendre le coût de cette association le plus négligeable possible est la source de nombreux tourments pour les implanteurs. Les valeurs sont aussi de taille variable puisqu'elles peuvent contenir un nombre arbitraire de valeurs. Le procédé consiste à manipuler ces valeurs par leur adresse qui, elle, a une taille fixe. Les objets seront alloués en mémoire et leur type sera encodé dans le mot qui les précède. L'inconvénient, par rapport à un langage typé statiquement où les types n'existent plus à l'exécution, est que les valeurs simples comme le peuvent être les petits entiers ne sont plus directement manipulables puisque l'on doit suivre un pointeur pour les obtenir. Des solutions existent pour pallier ce problème. Dans beaucoup de machines, les adresses sont des multiples de quatre ce qui laisse deux bits de poids faible libres dans une adresse. On les détourne, par exemple, pour y coder la nature de l'objet pointé. Et si la valeur référencée est un entier, alors autant le mettre à la place de l'adresse. C'est ce schéma que nous utiliserons pour les entiers, ce qui les rend plus efficaces mais qui les tronque d'un bit limitant ainsi leur dynamique. De nombreux autres schémas ont été inventés et sont répertoriés en [Gud93].

Pour résumer, voir la figure 10.5 : une valeur de Scheme sera représentée par une valeur de C telle que si son bit de poids faible est égal à 1, c'est un entier codé dans les bits restants : on parle d'entier 31 bits pour une machine à mots de 32 bits. Si le bit de poids faible est égal à 0, alors c'est l'adresse du premier champ d'une valeur allouée ;

---

6. Le temps qui figure ici correspond surtout au temps de chargement du programme et quasiment pas à son temps d'exécution qui est négligeable. On le verra plus loin, itérer ce calcul 10 000 fois donne un temps d'environ 1 seconde.

7. Ces temps ont été mesurés sur une station Sony News 3200 dotée d'un processeur Mips R3000.

le type de cette valeur apparaît dans le mot qui précède cette adresse[8] [DPS94a]. Une telle valeur apparaît comme un vrai pointeur pour C référençant directement les valeurs intéressantes de l'objet. Le type des valeurs de Scheme en C, celui que l'on manipulera tout le temps, sera nommé `SCM` :

```
typedef union SCM_object          *SCM;
```

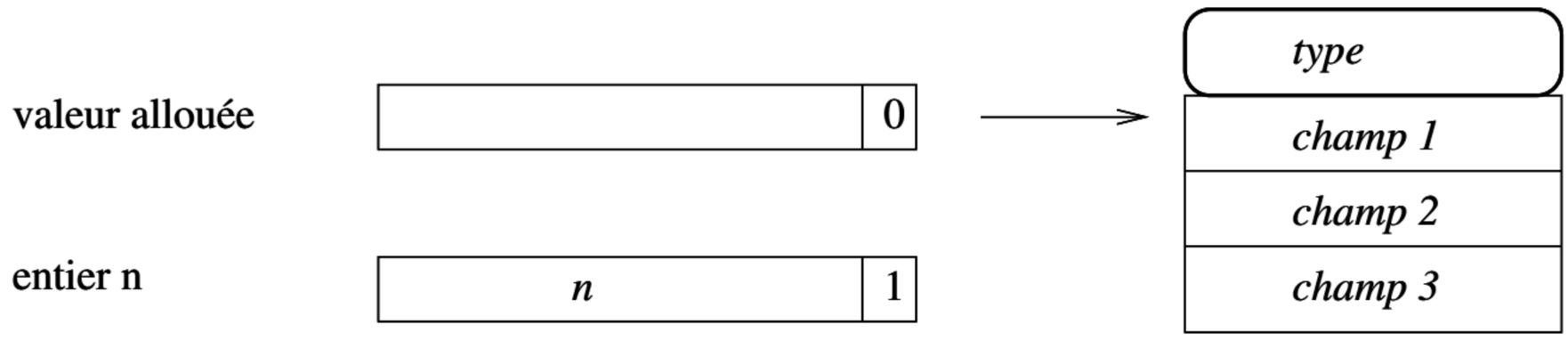


FIGURE 10.5 – Représentation des valeurs de Scheme en C

La macro `SCM_FixnumP` permet de distinguer un petit entier d'un pointeur. Petit entier et valeur peuvent être convertis l'un en l'autre grâce à `SCM_Fixnum2int` et `SCM_Int2fixnum`. L'entier 37 de Scheme est donc représenté par l'entier 75 en C.

```
#define SCM_FixnumP(x)     ((unsigned long)(x) & (unsigned long)1)
#define SCM_Fixnum2int(x)  ((long)(x)>>1)
#define SCM_Int2fixnum(i)  ((SCM)(((i)<<1) | 1))
```

Lorsqu'une valeur de `SCM` n'est pas un petit entier, c'est un pointeur sur un objet décrit par `SCM_object`. C'est une union des divers types possibles. Certains types y manquent comme les nombres flottants, les vecteurs, les ports d'entrée/sortie. On y trouve cependant les types essentiels que sont les paires (avec le `cdr` en tête pour rappeler que c'est d'abord une liste et ensuite parce que l'accès au `cdr` est un peu plus fréquent que celui au `car` [Cla79]).

```
union SCM_object {
  struct SCM_pair {
    SCM cdr;
    SCM car;
  } pair;
  struct SCM_string {
    char Cstring[8];
  } string;
  struct SCM_symbol {
    SCM pname;
  } symbol;
  struct SCM_box {
    SCM content;
  } box;
  struct SCM_subr {
    SCM (*behavior)();
    long arity;
  } subr;
  struct SCM_closure {
    SCM (*behavior)();
    long arity;
    SCM environment[1];
  } closure;
  struct SCM_escape {
    struct SCM_jmp_buf *stack_address;
  } escape;
};
```

Les objets de première classe figurant parmi les variantes de `SCM_object` sont préfixés par leur type qui n'apparaît qu'après moult magie C. Le type sera représenté par un type explicitement énuméré : `SCM_tag` (laissant de nombreux bits libres pour, par exemple, le GC), il sera stocké dans un champ de type `SCM\_header` qui aura

8. On notera que `NUL`, le pointeur nul de C généralement implanté comme `0L`, n'est pas un pointeur légal pour notre compilateur car il n'y a souvent pas de mot à l'adresse -1.

la même taille qu'un `SCM`, c'est ce qui justifie le membre `SCM ignored` dans l'union définissant `SCM_header`. Enfin les objets préfixés de leur type seront décrits par la structure `SCM_unwrapped_object`.

```
enum SCM_tag {
  SCM_NULL_TAG          =0xaaa0,
  SCM_PAIR_TAG          =0xaaa1,
  SCM_BOOLEAN_TAG       =0xaaa2,
  SCM_UNDEFINED_TAG     =0xaaa3,
  SCM_SYMBOL_TAG        =0xaaa4,
  SCM_STRING_TAG        =0xaaa5,
  SCM_SUBR_TAG          =0xaaa6,
  SCM_CLOSURE_TAG       =0xaaa7,
  SCM_ESCAPE_TAG        =0xaaa8
};
union SCM_header {
  enum SCM_tag tag;
  SCM ignored;
};
union SCM_unwrapped_object {
  struct SCM_unwrapped_immediate_object {
    union SCM_header header;
  } object;
  struct SCM_unwrapped_pair {
    union SCM_header header;
    SCM cdr;
    SCM car;
  } pair;
struct SCM_unwrapped_string {
  union SCM_header header;
  char Cstring[8];
} string;
struct SCM_unwrapped_symbol {
  union SCM_header header;
  SCM pname;
} symbol;
struct SCM_unwrapped_subr {
  union SCM_header header;
  SCM (*behavior)(void);
  long arity;
} subr;
struct SCM_unwrapped_closure {
  union SCM_header header;
  SCM (*behavior)(void);
  long arity;
    SCM environment[1];
  } closure;
  struct SCM_unwrapped_escape {
    union SCM_header header;
    struct SCM_jmp_buf
                *stack_address;
  } escape;
};
```

Les macros suivantes permettent de convertir les références entre elles ainsi que d'extraire le type d'un objet. En fait, seule la bibliothèque d'exécution a besoin de distinguer entre `SCM` et `SCMref`.

```
typedef union SCM_unwrapped_object *SCMref;
```

```
#define SCM_Wrap(x)         ((SCM) (((union SCM_header *) x) + 1))
#define SCM_Unwrap(x)       ((SCMref) (((union SCM_header *) x) - 1))
#define SCM_2tag(x)         ((SCM_Unwrap((SCM) x))->object.header.tag)
```

Enfin les addresses de fonctions écrites en C seront, sauf lorsqu'appliquées, typées comme retournant un `SCM` et ne prenant aucun argument. La macro `SCM_Cfunction-Address` permet d'obtenir cette conversion :

```
#define SCM_CfunctionAddress(Cfunction) ((SCM (*)(void)) Cfunction)
```

10.8.1 Déclaration des valeurs

Un jeu de macros existe pour allouer statiquement des valeurs de Scheme. Elles ont des noms préfixés par `SCM_Define`. La définition d'une paire pointée passe par l'allocation d'une structure décrite par `SCM_unwrapped_pair`. La définition des symboles est similaire. À chaque fois, l'objet est d'abord créé et un pointeur sur cet objet est créé (et converti par `SCM_Wrap` en une référence légale).

```
#define SCM_DefinePair(pair,car,cdr) \
  static struct SCM_unwrapped_pair pair = {{SCM_PAIR_TAG}, cdr, car }
#define SCM_DefineSymbol(symbol,pname) \
  static struct SCM_unwrapped_symbol symbol = \
        {{SCM_SYMBOL_TAG}, pname }
```

La définition des chaînes est un peu plus complexe puisque C ne sait pas initialiser des structures de données de taille variable. Il faut aussi faire attention en C lorsque l'on manipule des chaînes de ne point confondre leur contenu et leur adresse ainsi que de ne pas négliger l'ultime caractère (nul) par lequel elles s'achèvent. La solution prise est de bâtir une définition de structure appropriée par chaîne définie. Les chaînes de Scheme seront représentées par des chaînes en C sans aucune fioriture ajoutée.

```
#define SCM_DefineString(Cname,string) \
  struct Cname##_struct {              \
    union SCM_header header;           \
    char Cstring[1+sizeof(string)];};  \
  static struct Cname##_struct Cname = \
    {{SCM_STRING_TAG}, string }
```

Un certain nombre de valeurs prédéfinies doivent exister. Leur existence et leur type seuls sont utiles, pas leur contenu. Elles seront donc définies par des objets immédiats. On notera l'appel à `SCM_Wrap` pour convertir une adresse en un `SCM`.

```
#define SCM_DefineImmediateObject(name,tag) \
  struct SCM_unwrapped_immediate_object name = {{tag}}
SCM_DefineImmediateObject(SCM_true_object,SCM_BOOLEAN_TAG);
SCM_DefineImmediateObject(SCM_false_object,SCM_BOOLEAN_TAG);
SCM_DefineImmediateObject(SCM_nil_object,SCM_NULL_TAG);
#define SCM_true        SCM_Wrap(&SCM_true_object)
#define SCM_false       SCM_Wrap(&SCM_false_object)
#define SCM_nil         SCM_Wrap(&SCM_nil_object)
```

Les booléens de Scheme ne sont point ceux de C, il faut donc parfois convertir ces derniers en ces premiers. C'est l'objet de `SCM_2bool` :

```
#define SCM_2bool(i) ((i) ? SCM_true : SCM_false )
```

On introduit aussi quelques macros supplémentaires pour reconnaître et désosser des valeurs. Les prédicats qui suivent ont un nom suffixé par `P` pour rappeler qu'ils retournent un booléen de C.

```
#define SCM_Car(x)      (SCM_Unwrap(x)->pair.car)
#define SCM_Cdr(x)      (SCM_Unwrap(x)->pair.cdr)
#define SCM_NullP(x)    ((x)==SCM_nil)
#define SCM_PairP(x)    \
  ((!SCM_FixnumP(x)) && (SCM_2tag(x)==SCM_PAIR_TAG))
#define SCM_SymbolP(x) \
  ((!SCM_FixnumP(x)) && (SCM_2tag(x)==SCM_SYMBOL_TAG))
#define SCM_StringP(x) \
  ((!SCM_FixnumP(x)) && (SCM_2tag(x)==SCM_STRING_TAG))
#define SCM_EqP(x,y)    ((x)==(y))
```

Bien sûr des macros comme `SCM_Car` ou `SCM_Cdr` sont à employer dans des contextes sûrs lorsqu'il est connu que la valeur est une paire. Ce n'est pas le cas des macros arithmétiques suivantes qui testent explicitement que leurs arguments sont des petits entiers. Cette programmation interdit de tirer parti d'un contexte où l'on saurait que les arguments sont du bon type puisqu'elles effectuent toujours les tests de type[9]. Rappelons que notre étalage de variantes est avant tout pédagogique ; une recherche d'efficacité conduirait à adopter partout la meilleure organisation.

```
#define SCM_Plus(x,y)                                          \
 ( ( SCM_FixnumP(x) && SCM_FixnumP(y) )                        \
   ? SCM_Int2fixnum( SCM_Fixnum2int(x) + SCM_Fixnum2int(y) )   \
   : SCM_error(SCM_ERR_PLUS) )
#define SCM_GtP(x,y)                                           \
 ( ( SCM_FixnumP(x) && SCM_FixnumP(y) )                        \
   ? SCM_2bool( SCM_Fixnum2int(x) > SCM_Fixnum2int(y) )        \
   : SCM_error(SCM_ERR_GTP) )
```

...

10.8.2 Variables globales

Obtenir la valeur d'une variable globale modifiable passait par la macro `SCM_CheckedGlobal` permettant de vérifier si la variable était initialisée ou pas. Nous emploierons donc une valeur en C pour matérialiser la non-initialisation en Scheme : il s'agit de `SCM_undefined`. Les variables globales modifiables sont donc simplement initialisées (pour C) avec cette valeur indiquant que (pour Scheme) ce n'est pas initialisé.

```
#define SCM_CheckedGlobal(Cname) \
  ((Cname != SCM_undefined) ? Cname \
   : SCM_error(SCM_ERR_UNINITIALIZED))
#define SCM_DefineInitializedGlobalVariable(Cname,string,value) \
  SCM Cname = SCM_Wrap(value)
#define SCM_DefineGlobalVariable(Cname,string) \
  SCM_DefineInitializedGlobalVariable(Cname,string,\
                                      &SCM_undefined_object)
#define SCM_undefined   SCM_Wrap(&SCM_undefined_object)
SCM_DefineImmediateObject(SCM_undefined_object,SCM_UNDEFINED_TAG);
```

Un très important travail est de lier les valeurs prédéfinies aux variables globales par lesquelles les programmes peuvent y accéder. Par exemple, si l'on suppose que la variable globale `NIL` a pour valeur la liste vide `()`, il faut encore lier la variable C, `NIL`, à la valeur C, `SCM_nil_object`. C'est le but du fichier `schemelib.c` qui contient ces définitions et notamment :

9. Ou alors, c'est que le compilateur C retire de lui-même ces tests superflus, ce qui n'est généralement pas le cas.

```
SCM_DefineInitializedGlobalVariable(NIL,"NIL",&SCM_nil_object);
SCM_DefineInitializedGlobalVariable(F,"F",&SCM_false_object);
SCM_DefineInitializedGlobalVariable(T,"T",&SCM_true_object);
```

Mais au delà du folklore concernant ces trois dernières variables, il demeure primordial que `CAR`, `CONS` et les autres soient aussi disponibles. Ce sera chose faite grâce à la macro `SCM_DefinePredefinedFunctionVariable`, dont voici la définition accompagnée de quelques exemples :

```
#define SCM_DefinePredefinedFunctionVariable(subr,string,arity,Cfunc) \
  static struct SCM_unwrapped_subr subr##_object = \
    {{SCM_SUBR_TAG}, Cfunc, arity}; \
  SCM_DefineInitializedGlobalVariable(subr,string,&(subr##_object))
SCM_DefinePredefinedFunctionVariable(CAR,"CAR",1,SCM_car);
SCM_DefinePredefinedFunctionVariable(CONS,"CONS",2,SCM_cons);
SCM_DefinePredefinedFunctionVariable(EQN,"=",2,SCM_eqnp);
SCM_DefinePredefinedFunctionVariable(EQ,"EQ?",2,SCM_eqp);
```

Les variables globales modifiables sont mises dans des boîtes ; toutes les lectures et écritures passent par la macro `SCM_Content` ainsi définie :

```
#define SCM_Content(e) ((e)->box.content)
```

10.8.3 Définition des fonctions

Il ne reste plus qu'à expliquer comment sont représentées les fonctions du programme. La représentation des fonctions doit être soigneusement étudiée car c'est d'elle que dépend la plus ou moins grande coopération entre Scheme et C. Les fonctions primitives d'arité fixe sont représentées par des objets référençant des fonctions C de même arité. Ainsi, la fonction `cons` pourra-t-elle être invoquée depuis C sous la forme simple et mnémonique `SCM_cons(`x`,`y`)`.

De première classe en Scheme, les fonctions posent des problèmes bien plus graves : elles peuvent prendre un nombre quelconque d'arguments ; elles peuvent être calculées enfin, elles peuvent prendre leurs arguments de façon spéciale si invoquées via `apply`. Inventer un schéma efficace et satisfaisant toutes ces contraintes est loin d'être trivial. Nous nous autoriserons plus de latitude en en montrant plusieurs à l'œuvre et nous adopterons pour les fonctions non primitives un schéma original et régulier sinon véloce. **[☞ Ex. 10.1]**

Les fermetures sont représentées par des objets dont le premier champ est un pointeur sur une fonction C ; le second champ indique l'arité de la fonction tandis que les champs supplémentaires contiennent les variables closes. Pour chaque type de fermeture, la macro `SCM_DefineClosure` définit une structure C appropriée.

```
#define SCM_DefineClosure(struct_name,fields) \
  struct struct_name {                \
    SCM (*behavior)(void);            \
    long arity;                       \
    fields }
```

Lorsqu'une fermeture est invoquée par `SCM_invoke`, elle se recevra elle-même en premier argument afin d'extraire d'elle-même ses variables closes. Lorsqu'une fonction est appelée avec un nombre incorrect d'arguments, il est nécessaire de signaler une erreur : on supposera que la fonction `SCM_invoke` se charge de cette vérification qui n'incombe donc pas à l'appelé. Toutefois, lorsqu'une fonction reçoit

un nombre variable d'arguments, il faut lui indiquer combien elle en trouvera, car nul moyen linguistique permet sans a priori de le déterminer en C. On supposera donc que les fonctions C associées aux fermetures de Scheme prennent le nombre d'arguments qu'elles vont recevoir en second argument. Ces arguments suivront et seront fournis par le mécanisme dit de `varargs` en C traditionnel maintenant nommé `stdarg` [10]. Comme nous l'avons déjà fait remarquer, ce n'est pas le meilleur mécanisme envisageable : celui adopté par les primitives est plus efficace.

Les noms des variables C mises en œuvre ci-après (à savoir `self_`, `size_` et `arguments_`) sont à usage interne et ne servent qu'à la communication avec les macros définissant les variables locales. Le seul cas intéressant est celui d'une variable pointée qui doit empaqueter ses arguments en une liste ; c'est l'objet de la fonction `SCM_list` qui, justement, a la bonne interface.

```
#define SCM_DeclareFunction(Cname) \
  SCM Cname (struct Cname *self_, unsigned long size_, \
             va_list arguments_)
#define SCM_DeclareLocalVariable(Cname,rank) \
  SCM Cname = va_arg(arguments_,SCM)
#define SCM_DeclareLocalDottedVariable(Cname,rank) \
  SCM Cname = SCM_list(size_-rank,arguments_)
#define SCM_Free(Cname) ((*self_).Cname)
```

Une des beautés de ce schéma est que l'accès aux variables closes s'exprime simplement puisque celles-ci nomment les membres de la structure associée à la fermeture.

10.9 Bibliothèque d'exécution

La bibliothèque d'exécution est une suite de fonctions écrites en C qu'il faut relier à son programme afin qu'il y puise les ressources qui lui manquent encore. Cette bibliothèque est squelettique car lui font défaut de nombreuses fonctions utiles mais non fondamentales comme `string-ref` ou `close-output-port`. Nous n'allons décrire que les principales fonctions représentatives et ignorerons les autres (et notamment `SCM_print` qui apparaissait dans les `main` engendrés).

10.9.1 Allocation

Il n'y a pas de gestionnaire de mémoire et donc pas de glaneur de cellules (GC) car cela nous entrainerait trop loin. On consultera pour en savoir plus [Spi90, Wil92] et on laissera en exercice le soin d'adapter le GC de Boehm [BW88] à ce qui suit. La fonction d'allocation la plus évidente est `cons`. On y alloue un objet préfixé de son type que l'on renseigne de même que ses champs `car` et `cdr` ; enfin l'on convertit son adresse en un `SCM` que l'on retourne en valeur.

```
SCM SCM_cons (SCM x, SCM y) {
  SCMref cell = (SCMref) malloc(sizeof(struct SCM_unwrapped_pair));
```

10. Je ne suis pas assez spécialiste de C pour déterminer si le codage par `stdarg` est réellement portable. On verra, dans `SCM_invoke`, que tous les `va_start` exécutent bien le `va_end` associé au même niveau lexical (en C) mais il demeure des emplois de `va_arg` (par exemple dans `SCM_apply`) qui ne sont pas dans la portée lexicale associée à ces `va_start` : c'est peut-être osé !

```
  if (cell == (SCMref) NULL) SCM_error(SCM_ERR_CANT_ALLOC);
  cell->pair.header.tag = SCM_PAIR_TAG;
  cell->pair.car = x;
  cell->pair.cdr = y;
  return SCM_Wrap(cell);
}
```

Les fermetures sont allouées par la fonction `SCM_close` qui, elle, prend un nombre variable d'arguments, ce qui justifie a posteriori le choix d'utiliser le mécanisme d'argument multiples de C. Il s'agit donc d'allouer un objet, portant le type `SCM_CLOSURE_TAG`, et de renseigner les autres champs en fonction du nombre d'arguments reçus.

```
SCM
SCM_close (SCM (*Cfunction)(void),
           long arity,
           unsigned long size, ... ) {
  SCMref result = (SCMref) malloc(sizeof(struct SCM_unwrapped_closure)
                                  + (size-1)*sizeof(SCM) );
  unsigned long i;
  va_list args;
  if (result == (SCMref) NULL) SCM_error(SCM_ERR_CANT_ALLOC);
  result->closure.header.tag = SCM_CLOSURE_TAG;
  result->closure.behavior = Cfunction;
  result->closure.arity = arity;
  va_start(args,size);
  for ( i=0 ; i<size ; i++ ) {
    result->closure.environment[i] = va_arg(args,SCM);
  }
  va_end(args);
  return SCM_Wrap(result);
}
```

10.9.2 Fonctions sur les paires

Les fonctions `car` et `set-cdr!` sont simples mais ne doivent point être confondues avec les macros de noms presque semblables. Ces fonctions sont ici sûres en ce sens qu'elles testent la nature de leurs arguments avant de s'appliquer. En général, il y a la fonction sûre et la fonction non sûre qui n'est substituée à la première que lorsque le compilateur a pu assurer que c'était possible. Bien que Lisp soit non typé, les programmes sont tels que presque deux fois sur trois les tests de types peuvent être supprimés [Hen92a, WC94, Ser94] : les mentalités sont telles que les gens typent dans leur tête, ce dont peut profiter un compilateur malin.

```
SCM SCM_car (SCM x) {
  if ( SCM_PairP(x) ) {
    return SCM_Car(x);
  } else return SCM_error(SCM_ERR_CAR);
}
SCM SCM_set_cdr (SCM x, SCM y) {
  if ( SCM_PairP(x) ) {
    SCM_Unwrap(x)->pair.cdr = y;
    return x;
  } else return SCM_error(SCM_ERR_SET_CDR);
}
```

Il est bon aussi d'indiquer ce qu'est la fonction `list` qui, elle, prend un nombre variable d'arguments et donc les reçoit préfixés par leur nombre. Pour sa programmation interne, on remarquera que nous avons encore péché par récursivité !

```
SCM SCM_list (unsigned long count, va_list arguments) {
  if ( count == 0 ) {
    return SCM_nil;
  } else {
    SCM arg = va_arg(arguments,SCM);
    return SCM_cons(arg,SCM_list(count-1,arguments));
  }
}
```

Le mécanisme de rattrapage d'erreur est inexistant ; on se contente de signaler les erreurs à l'aide de la macro C `SCM_error` qui indique, outre un code représentatif de l'erreur (c'est la mode en C), la ligne et le fichier erronés.

```
#define SCM_error(code) SCM_signal_error(code,__LINE__,__FILE__)
SCM SCM_signal_error (unsigned long code,
                      unsigned long line,
                      char *file ) {
  fflush(stdout);
  fprintf(stderr,"Error %u, Line %u, File %s.\n",code,line,file);
  exit(code);
}
```

10.9.3 Invocation

Les deux fonctions à la fois importantes et délicates traitant de ce sujet sont `SCM_invoke` et `apply`. Cette dernière ne nous surprend pas parce qu'elle a à connaître le protocole d'appel des fonctions pour s'y conformer.

Tous les appels fonctionnels autres qu'intégrés (c'est-à-dire transformés en un appel direct à la macro ou à la fonction C concernée) passent par `SCM_invoke`. Cette dernière prend en premier argument la fonction à appeler, en second argument le nombre d'arguments fournis et, juste après, ces dits arguments. Comme dans les interprètes précédents, la fonction `SCM_invoke` est plus ou moins générique. Elle analyse la nature de son premier argument qui doit être un objet invoquable : une fonction primitive ou pas (ou un échappement). Elle extrait ensuite de l'objet à appliquer, l'arité de celui-ci et la compare au nombre d'arguments reçus. Enfin elle fournit ces arguments à cette fonction suivant le protocole approprié. Ici nous avons trois protocoles différents, d'autres existent comme de suffixer les arguments par `NUL` plutôt que de les préfixer par leur nombre (c'est la technique qu'emploie Bigloo) :

— Les primitives d'arité fixe (comme `SCM_cons`) : elles sont appelées, directement, avec leurs arguments. L'appel fonctionnel peut être schématisé comme étant du type $f(x, y)$.
— Les primitives d'arité quelconque (i.e., `SCM_list`) : celles-ci doivent nécessairement connaître le nombre des arguments fournis. Ce nombre sera passé en premier argument. L'appel fonctionnel est donc du type $f(n, x_1, x_2, \ldots, x_n)$.
— Les fermetures : elles doivent non seulement (quand elles sont d'arité variable) connaître le nombre d'arguments reçus mais en plus récupérer leurs variables closes qui sont en fait stockées dans la fermeture même. Le type de l'appel fonctionnel alors réalisé sera $f(f, n, x_1, x_2, \ldots, x_n)$.

On pourrait, bien sûr, raffiner, unifier ou encore éliminer certains de ces protocoles. Voici donc la fonction `SCM_invoke` qui, quoique imposante par sa masse, est de structure éminemment régulière. Nous avons supprimé la partie concernant les continuations qui fera l'objet d'un prochain paragraphe.

```
SCM SCM_invoke(SCM function, unsigned long number, ...) {
  if ( SCM_FixnumP(function) ) {
    /* Cannot apply a number! */
    return SCM_error(SCM_ERR_CANNOT_APPLY);
  } else {
switch SCM_2tag(function) {
case SCM_SUBR_TAG: {
  SCM (*behavior)(void) = (SCM_Unwrap(function)->subr).behavior;
  long arity = (SCM_Unwrap(function)->subr).arity;
  SCM result;
  if ( arity >= 0 ) {          /* Fixed arity subr */
    if ( arity != number ) {
      return SCM_error(SCM_ERR_WRONG_ARITY); /* Wrong arity! */
    } else {
      if ( arity == 0) {
        result = behavior();
      } else {
        va_list args;
        va_start(args,number);
        { SCM a0 ;
          a0 = va_arg(args,SCM);
          if ( arity == 1 ) {
        result = ((SCM (*)(SCM)) *behavior)(a0);
      } else {
        SCM a1 ;
        a1 = va_arg(args,SCM);
        if ( arity == 2 ) {
          result = ((SCM (*)(SCM,SCM)) *behavior)(a0,a1);
        } else {
          SCM a2 ;
          a2 = va_arg(args,SCM);
          if ( arity == 3 ) {
            result =
              ((SCM (*)(SCM,SCM,SCM)) *behavior)(a0,a1,a2);
          } else {
/* No fixed arity subr with more than 3 variables */
            return SCM_error(SCM_ERR_INTERNAL);
          }
        }
      }
      va_end(args);
    }
    return result;
  }
} else {                       /* Nary subr */
  long min_arity = SCM_MinimalArity(arity) ;
  if ( number < min_arity ) {
    /* Too less arguments! */
    return SCM_error(SCM_ERR_MISSING_ARGS);
  } else {
    va_list args;
```

```
        SCM result;
        va_start(args,number);
        result = ((SCM (*)(unsigned long,va_list))
                  *behavior)(number,args);
        va_end(args);
        return result;
      }
    }
}
case SCM_CLOSURE_TAG: {
  SCM (*behavior)(void) =
    (SCM_Unwrap(function)->closure).behavior;
long arity = (SCM_Unwrap(function)->closure).arity ;
SCM result;
va_list args;
va_start(args,number);
if ( arity >= 0 ) {
  if ( arity != number ) { /* Wrong arity! */
    return SCM_error(SCM_ERR_WRONG_ARITY);
  } else {
    result = ((SCM (*)(SCM,unsigned long,va_list)) *behavior)
              (function,number,args);
  }
} else {
  long min_arity = SCM_MinimalArity(arity) ;
  if ( number < min_arity ) {
    /* Too less arguments! */
    return SCM_error(SCM_ERR_MISSING_ARGS);
        } else {
          result = ((SCM (*)(SCM,unsigned long,va_list)) *behavior)
                    (function,number,args);
        }
      }
      va_end(args);
      return result;
    }
    default: {
      SCM_error(SCM_ERR_CANNOT_APPLY);    /* Cannot apply! */
    }
    }
  }
}
```

La fonction `apply` est tout aussi imposante par sa masse. Ce qui est intéressant, ici, c'est que `apply` étant une fonction primitive d'arité fixe, elle reçoit ses arguments dans une instance de `va_list` dont la dernière position contient une liste qu'il faut explorer. Le problème de l'interface des variables multiples de C est que l'on ne peut construire de nouvelles instances de `va_list` car c'est un type privé de l'implantation du compilateur C. Il ne reste donc plus qu'à prendre son mal en patience et discriminer toutes les arités pour engendrer tous les appels plausibles. L'ennui est que l'on ne peut les énumérer tous : on doit donc contraindre `apply` à ne pouvoir transmettre qu'un nombre borné d'arguments. Le cas était prévu en COMMON LISP, puisqu'il y existe une constante indiquant le nombre maximal d'arguments permis : `call-arguments-limit` dont la valeur doit être au moins 50. Je me suis limité à qua-

torze, dont seulement les quatres premières possibilités sont déployées ci-dessous [11] !

```
SCM SCM_apply (unsigned long number, va_list arguments) {
  SCM args[31];
  SCM last_arg;
  SCM fun = va_arg(arguments,SCM);
  unsigned long i;
  for ( i=0 ; i<number-1 ; i++  ) {
    args[i] = va_arg(arguments,SCM);
  }
  last_arg = args[--i];
  while ( SCM_PairP(last_arg) ) {
    args[i++] = SCM_Car(last_arg);
    last_arg = SCM_Cdr(last_arg);
  }
  if ( ! SCM_NullP(last_arg) ) {
    SCM_error(SCM_ERR_APPLY_ARG);
  }
  switch ( i ) {
  case 0: return SCM_invoke(fun,0);
  case 1: return SCM_invoke(fun,1,args[0]);
  case 2: return SCM_invoke(fun,2,args[0],args[1]);
  case 3: return SCM_invoke(fun,3,args[0],args[1],args[2]);
  case 4: return SCM_invoke(fun,4,args[0],args[1],args[2],args[3]);
                    ...
  default: return SCM_error(SCM_ERR_APPLY_SIZE);
  }
}
```

C ne garantit pas moins de 32 arguments aussi n'est-il pas besoin d'aller jusqu'à 50. Les puristes noteront le test (inutile ?) vérifiant que l'ultime `cdr` de la liste contenant les arguments finaux soumis à `apply` est bien vide.

Le problème essentiel du mécanisme d'invocation en C est qu'il ne préserve pas la propriété, imposée par Scheme, qu'un appel terminal se fasse à continuation constante. En conséquence, un programme traduit en C peut s'interrompre à cause d'un débordement de pile ce qui ne peut se produire dans une implantation conforme de Scheme. Certains compilateurs de Scheme vers C (Scheme->C, Bigloo) consomment beaucoup d'énergie pour éviter ces problèmes (fonctions récursives, boucles, etc.) mais ne peuvent trouver tous ces cas et sont donc faillibles sur certains styles de programmation. Une autre solution est de ne pas utiliser la pile de C et de manipuler explicitement les continuations de Scheme.

10.10 Avoir `call/cc` ou pas ?

Nous avons terminé la compilation d'un ensemble significatif de Scheme vers C. Malgré les très nombreuses fonctions manquantes, l'essentiel est présent sinon efficace. Les continuations, par contre, manquent. Comme pour le compilateur de code-octet, nous allons donner la définition de la fonction `call/ep` qui procure des continuations à durée de vie dynamique, comme en Lisp que nous traduirons par des `setjmp`/`longjmp`. Dans une seconde partie, nous nous attacherons à fournir de vraies continuations à la Scheme.

---

11. Si cette limitation vous semble absurde, testez donc sa valeur dans votre Lisp préféré.

10.10.1 La fonction `call/ep`

La fonction `call/ep` déjà décrite a la même interface que la fonction `call/cc` mais la continuation de première classe qu'elle synthétise ne peut être validement utilisée que pendant sa durée de vie dynamique. **[☞ p. 96]** Nous allons utiliser la même implantation qu'au chapitre 7 à savoir qu'on alloue un objet dans le tas pour représenter la continuation. Celle-ci pointera sur le `jmp_buf` dans la pile qui, lui-même, référencera la continuation. Voici les fonctions :

```
SCM SCM_allocate_continuation (struct SCM_jmp_buf *address) {
  SCMref continuation =
          (SCMref) malloc(sizeof(struct SCM_unwrapped_escape));
  if (continuation == (SCMref) NULL) SCM_error(SCM_ERR_CANT_ALLOC);
  continuation->escape.header.tag = SCM_ESCAPE_TAG;
  continuation->escape.stack_address = address;
  return SCM_Wrap(continuation);
}
struct SCM_jmp_buf {
  SCM back_pointer;
  jmp_buf jb;
};
SCM SCM_callep (SCM f) {
  struct SCM_jmp_buf scmjb;
  SCM continuation = SCM_allocate_continuation(&scmjb);
  scmjb.back_pointer = continuation;
  if ( setjmp(scmjb.jb) != 0 ) {
    return jumpvalue;
  } else {
    return SCM_invoke1(f,continuation);
  }
}
```

Lorsqu'une continuation est invoquée, c'est la fonction `SCM_invoke` qui s'en aperçoit et qui vérifie l'arité de cet appel avant de procéder à l'appel proprement dit. Le fragment qui suit est à insérer dans la fonction `SCM_invoke` précédemment décrite. Pour vérifier que la continuation est valide (et parce que l'on ne dispose pas de `unwind-protect` pour l'invalider), on teste si elle est bien appariée au `jmp_buf` dans la pile et si actuellement on est bien *au dessus* de ce `jmp_buf`. Cette dernière vérification demande malheureusement de connaître le sens de variation de la pile de C, connaissance encapsulée dans la macro `SCM_STACK_HIGHER`, dont la définition dépend de l'implantation mais dont la connaissance s'acquiert simplement grâce à un petit programme en C portable. Généralement, sous UN⋆X, la pile croît par adresses décroissantes aussi `SCM_STACK_HIGHER` n'est-elle autre que `<=`.

```
...  case SCM_ESCAPE_TAG: {
      if ( number == 1) {
        va_list args;
        va_start(args,number);
        jumpvalue = va_arg(args,SCM);
        va_end(args);
        { struct SCM_jmp_buf *address =
            SCM_Unwrap(function)->escape.stack_address;
            if ( SCM_EqP(address->back_pointer,function)
                && ( (void *) &address
                        SCM_STACK_HIGHER (void *) address ) ) {
              longjmp(address->jb,1);
```

```
            } else {    /* surely out of dynamic extent! */
              return SCM_error(SCM_ERR_OUT_OF_EXTENT);
            }
          }
        } else {
          /* Too less arguments! */
          return SCM_error(SCM_ERR_MISSING_ARGS);
        }
      }  ...
```

Ce que nous avions dit de l'efficacité de `call/ep` dans le cadre du compilateur octet ne tient plus ici car les `longjmp` de C sont notoirement lents et ralentissent énormément les programmes qui en feraient trop usage. L'intérêt est toutefois de montrer l'intégration de Lisp et de C que l'on peut réaliser.

10.10.2 La fonction `call/cc`

Mais les nostalgiques des vraies continuations restent sur leur faim. On sait que l'on a la possibilité de procurer `call/cc` comme une fonction magique, occulte et mystérieuse. Il faut malheureusement pour pouvoir l'écrire, connaître au moins un peu la structure de la pile de C car c'est elle qui contient ce que l'on doit capturer. Mais C ne procure aucun moyen portable d'introspection de la pile aussi est-il extrêmement difficile d'implanter des continuations de ce type en C portable sans d'encombrantes définitions conditionnelles. Il ne nous reste donc plus qu'à effectuer une transformation de programme faisant apparaître les continuations (CPS) et compiler le tout avec le précédent compilateur.

Explicitation des continuations

La transformation que nous allons présenter et qui utilise, une fois de plus, notre arpenteur favori est équivalente à celle présentée naguère. **[☞ p. 169]** Cette transformation est opérée sur l'expression initiale dès qu'objectifiée. Toutefois comme les formes `let` (les nœuds de classe `Fix-Let`) auront disparu dans l'affaire, nous les réintroduirons au moyen d'un second arpentage qui ne vise qu'à retrouver et convertir les formes closes, ce qui, au passage, supprimera les fameux redex administratifs. La transformation `letify` sera expliquée juste après CPS. Voici cependant la nouvelle version du compilateur :

```
(define (compile->C e out)
  (set! g.current '())
  (let* ((ee (letify (cpsify (Sexp->object e)) '()))
         (prg (extract-things! (lift! ee))) )
    (gather-temporaries! (closurize-main! prg))
    (generate-C-program out e prg) ) )
```

L'arpenteur explicitant les continuations se nomme `cpsify` et résulte de l'interaction de la fonction de parcours `update-walk!` et de la fonction générique `->CPS`. Deux nouvelles classes d'objets sont introduites : la classe des continuations (qui ne sert qu'à marquer de façon visible les abstractions jouant le rôle de continuations) et la classe des pseudo-variables servant de variables pour les continuations.

```
(define-class Continuation Function ())
(define-class Pseudo-Variable Local-Variable ())
```

```
(define (cpsify e)
  (let ((v (new-Variable)))
    (->CPS e (make-Continuation (list v)
                                (make-Local-Reference v) )) ) )
(define new-Variable
  (let ((counter 0))
    (lambda ()
      (set! counter (+ 1 counter))
      (make-Pseudo-Variable counter #f #f) ) ) )
```

La fonction `->CPS` prend l'objet à convertir en premier argument et la continuation courante en second argument. Par défaut, elle applique la continuation à l'objet. On aura remarqué que la continuation initiale, apparaissant dans `cpsify`, est l'identité $\lambda x.x$ objectifiée.

```
(define-generic (->CPS (e Program) k)
  (convert2Regular-Application k e) )
(define (convert2Regular-Application k . args)
  (make-Regular-Application k (convert2arguments args)) )
```

Il ne reste plus qu'à énoncer les méthodes appropriées. La séquence est simple : on convertit la première forme en reléguant la seconde dans la continuation.

```
(define-method (->CPS (e Sequence) k)
  (->CPS (Sequence-first e)
         (let ((v (new-Variable)))
           (make-Continuation
            (list v) (->CPS (Sequence-last e) k) ) ) ) )
```

L'alternative s'énonce aussi simplement mais peut dupliquer la continuation dans les deux branches de l'alternative.

```
(define-method (->CPS (e Alternative) k)
  (->CPS (Alternative-condition e)
         (let ((v (new-Variable)))
           (make-Continuation
            (list v) (make-Alternative
                      (make-Local-Reference v)
                      (->CPS (Alternative-consequent e) k)
                      (->CPS (Alternative-alternant e) k) ) ) ) ) )
```

Les affectations sont aussi simples. On convertit la valeur à affecter en prenant soin d'effectuer la modification dans la continuation.

```
(define-method (->CPS (e Box-Write) k)
  (->CPS (Box-Write-form e)
         (let ((v (new-Variable)))
           (make-Continuation
            (list v) (convert2Regular-Application
                      k (make-Box-Write
                         (Box-Write-reference e)
                         (make-Local-Reference v) ) ) ) ) ) )
(define-method (->CPS (e Global-Assignment) k)
  (->CPS (Global-Assignment-form e)
         (let ((v (new-Variable)))
           (make-Continuation
            (list v) (convert2Regular-Application
                      k (make-Global-Assignment
                         (Global-Assignment-variable e)
                         (make-Local-Reference v) ) ) ) ) ) )
```

Pour se simplifier la vie (et parce que l'objectification en fait trop), on défait les applications closes au profit d'applications de fermetures. Rappelons que les applications closes engendrées par la transformation seront identifiées par `letify`. Il y en aura d'ailleurs beaucoup car la transformation CPS naïve que nous programmons engendre de nombreux redex administratifs [SF92]. Ceux-ci ne sont toutefois pas gênants car, compilés correctement, ils disparaîtront naturellement : ils ne sont déplaisants qu'au regard et ne gênent que l'interprétation.

```
(define-method (->CPS (e Fix-Let) k)
  (->CPS (make-Regular-Application
          (make-Function (Fix-Let-variables e) (Fix-Let-body e))
          (Fix-Let-arguments e) )
         k ) )
```

Les fonctions se voient affublées d'un argument supplémentaire qui représente la continuation de leur appelant.

```
(define-method (->CPS (e Function) k)
  (convert2Regular-Application
   k (let ((k (new-Variable)))
       (make-Function (cons k (Function-variables e))
                      (->CPS (Function-body e)
                             (make-Local-Reference k) ) ) ) ) ) )
```

Les formes les plus complexes sont les applications fonctionnelles qui se traitent semblablement. Il faut calculer tous les arguments les uns après les autres avant de leur appliquer la fonction. Nous avons une fois de plus choisi l'ordre gauche-droite.

```
(define-method (->CPS (e Predefined-Application) k)
  (let* ((args (Predefined-Application-arguments e))
         (vars (let name ((args args))
                 (if (Arguments? args)
                     (cons (new-Variable)
                           (name (Arguments-others args)) )
                     '() ) ))
         (application
          (convert2Regular-Application
           k
           (make-Predefined-Application
            (Predefined-Application-variable e)
            (convert2arguments
             (map make-Local-Reference vars) ) ) ) ) )
    (arguments->CPS args vars application) ) )
(define-method (->CPS (e Regular-Application) k)
  (let* ((fun (Regular-Application-function e))
         (args (Regular-Application-arguments e))
         (varfun (new-Variable))
         (vars (let name ((args args))
                 (if (Arguments? args)
                     (cons (new-Variable)
                           (name (Arguments-others args)) )
                     '() ) ))
         (application
          (make-Regular-Application
           (make-Local-Reference varfun)
           (make-Arguments
            k (convert2arguments
               (map make-Local-Reference vars) ) ) ) ) )
(->CPS fun (make-Continuation
```

```
                  (list varfun)
                  (arguments->CPS args vars application) )) ) )
(define (arguments->CPS args vars appl)
  (if (pair? vars)
      (->CPS (Arguments-first args)
             (make-Continuation
              (list (car vars))
              (arguments->CPS (Arguments-others args)
                              (cdr vars)
                              appl ) ) )
```

Réintroduction de formes closes

La tranformation précédemment indiquée, `letify`, a pour charge d'identifier les formes closes et de les traduire en des formes `let` appropriées. Toutefois, nous allons profiter de ce polissage pour nettoyer le résultat de `->CPS` qui, lorsqu'elle traita l'alternative, dupliqua (par partage physique) dans les deux branches possibles de l'alternative, le même sous-arbre de syntaxe abstraite représentant la continuation. Nous n'avons donc plus un arbre de syntaxe abstraite mais un DAG. Pour éviter que des transformations ultérieures n'achoppent à cause de ces partages physiques occultes, la fonction `letify` va entièrement recopier le DAG de syntaxe abstraite en un pur arbre de syntaxe abstraite. Nous supposerons disposer de la fonction générique `clone`, **[☞ Ex. 11.2]** capable de dupliquer tout objet de MEROONET, et nous l'adapterons au cas des variables qu'il nous faudra renommer. Voici donc cette transformation qui entretient, avec `collect-temporaries!`, une certaine ressemblance pour le renommage des variables locales :

```
(define-generic (letify (o Program) env)
  (update-walk! letify (clone o) env) )
(define-method (letify (o Function) env)
  (let* ((vars (Function-variables o))
         (body (Function-body o))
         (new-vars (map clone vars)) )
    (make-Function
     new-vars
     (letify body (append (map cons vars new-vars) env)) ) ) )
(define-method (letify (o Local-Reference) env)
  (let* ((v (Local-Reference-variable o))
         (r (assq v env)) )
    (if (pair? r)
        (make-Local-Reference (cdr r))
        (letify-error "Disappeared variable" o) ) ) )
(define-method (letify (o Regular-Application) env)
  (if (Function? (Regular-Application-function o))
      (letify (process-closed-application
               (Regular-Application-function o)
               (Regular-Application-arguments o) )
              env )
      (make-Regular-Application
       (letify (Regular-Application-function o) env)
       (letify (Regular-Application-arguments o) env) )  ) )
(define-method (letify (o Fix-Let) env)
```

```
  (let* ((vars (Fix-Let-variables o))
         (new-vars (map clone vars)) )
    (make-Fix-Let
     new-vars
     (letify (Fix-Let-arguments o) env)
     (letify (Fix-Let-body o)
             (append (map cons vars new-vars) env) ) ) ) )
(define-method (letify (o Box-Creation) env)
  (let* ((v (Box-Creation-variable o))
         (r (assq v env)) )
    (if (pair? r)
        (make-Box-Creation (cdr r))
        (letify-error "Disappeared variable" o) ) ) )
(define-method (clone (o Pseudo-Variable))
  (new-Variable) )
```

Bibliothèque d'exécution

Il est probablement difficile de se figurer où conduit la transformation précédente. Le but était de rendre les continuations apparentes, c'est-à-dire `call/cc` triviale à implanter. Une continuation sera représentée par une fermeture dont le comportement est d'oublier la continuation courante pour restaurer celle qu'elle représente. Voici la définition de `SCM_callcc`, on y notera qu'étant une fonction normale, elle est appelée avec un premier argument qui est la continuation de l'appelant et c'est pourquoi son arité est, ici, de deux ! La fonction `SCM_invoke_continuation` apparaît d'abord pour aider le compilateur C qui aime découvrir les choses dans le bon ordre.

```
SCM SCM_invoke_continuation (SCM self, unsigned long number,
                             va_list arguments) {
  SCM current_k = va_arg(arguments,SCM);
  SCM value     = va_arg(arguments,SCM);
  return SCM_invoke1(SCM_Unwrap(self)->closure.environment[0],value);
}
SCM SCM_callcc (SCM k, SCM f) {
  SCM reified_k =
     SCM_close(SCM_CfunctionAddress(SCM_invoke_continuation),
               2, 1, k );
  return SCM_invoke2(f,k,reified_k);
}
SCM_DefinePredefinedFunctionVariable(CALLCC,"CALL/CC",2,SCM_callcc);
```

La bibliothèque des fonctions prédéfinies en C n'a pas à être changée ; par contre leur protocole d'appel est quelque peu modifié en raison du grouillement des continuations. Si l'on écrit `(let ((f car)) (f '(a b)))`, notre stupide compilateur ne détecte pas que c'est en fait équivalent à `(car '(a b))` (et même directement à `(quote a)` par propagation de constantes). Il engendrera un appel calculé à la valeur de la variable globale `CAR` qui recevra ici une continuation en premier argument et une paire en second. La nouvelle valeur de `CAR` n'est en fait que `(lambda (k p) (k (car p)))` exprimée avec l'ancienne fonction `car`. Nous introduirons donc encore quelques macros C bien senties et nous redéfinirons la bibliothèque initiale avec une arité accrue de un. Pour ne point lasser, nous ne montrerons que quelques exemples. Les fonctions préfixées par `SCMq_` ne constituent que des interfaces aux fonctions simplement préfixées par `SCM_`.

```
#define SCM_DefineCPSsubr2(newname,oldname) \
  SCM newname (SCM k, SCM x, SCM y) {  \
    return SCM_invoke1(k,oldname(x,y)); \
  }
#define SCM_DefineCPSsubrN(newname,oldname)          \
  SCM newname (unsigned long number, va_list arguments) { \
    SCM k = va_arg(arguments,SCM);          \
    return SCM_invoke1(k,oldname(number-1,arguments));          \
}
SCM_DefineCPSsubr2(SCMq_gtp,SCM_gtp)
SCM_DefinePredefinedFunctionVariable(GREATERP,">",3,SCMq_gtp);
SCM_DefineCPSsubrN(SCMq_list,SCM_list)
SCM_DefinePredefinedFunctionVariable(LIST,"LIST",-2,SCMq_list);
```

Un dernier problème est celui de la fonction `apply` qui doit réarranger ses arguments pour remettre la continuation qu'elle reçoit en bonne position.

```
SCM SCMq_apply (unsigned long number, va_list arguments) {
  SCM args[32];
  SCM last_arg;
  SCM k   = va_arg(arguments,SCM);
  SCM fun = va_arg(arguments,SCM);
  unsigned long i;
  for ( i=0 ; i<number-2 ; i++  ) {
    args[i] = va_arg(arguments,SCM);
  }
  last_arg = args[--i];
  while ( SCM_PairP(last_arg) ) {
    args[i++] = SCM_Car(last_arg);
    last_arg = SCM_Cdr(last_arg);
  }
  if ( ! SCM_NullP(last_arg) ) {
    SCM_error(SCM_ERR_APPLY_ARG);
  }
  switch ( i ) {
  case 0: return SCM_invoke(fun,1,k);
  case 1: return SCM_invoke(fun,2,k,args[0]);
  case 2: return SCM_invoke(fun,3,k,args[0],args[1]);
  ...
   default: return SCM_error(SCM_ERR_APPLY_SIZE);
  }
}
```

Exemple

Reprenons donc notre exemple courant et montrons le code engendré dans lequel on retrouvera, nonobstant la syntaxe C, la saveur de CPS.

o/chap10kex.c

```
/* Compiler to C $Revision: 4.0 $
(BEGIN
  (SET! INDEX 1)
  ((LAMBDA
     (CNTER . TMP)
     (SET! TMP (CNTER (LAMBDA (I) (LAMBDA X (CONS I X)))))
     (IF CNTER (CNTER TMP) INDEX))
```

```
    (LAMBDA (F) (SET! INDEX (+ 1 INDEX)) (F INDEX))
    'FOO))  */
#include "scheme.h"
/* Global environment: */
SCM_DefineGlobalVariable(INDEX,"INDEX");
/* Quotations: */
#define thing3 SCM_nil /* () */
SCM_DefineString(thing4_object,"FOO");
#define thing4 SCM_Wrap(&thing4_object)
SCM_DefineSymbol(thing2_object,thing4);       /* FOO */
#define thing2 SCM_Wrap(&thing2_object)
#define thing1 SCM_Int2fixnum(1)
#define thing0 thing1 /* 1 */
/* Functions: */
SCM_DefineClosure(function_0, SCM I; );
SCM_DeclareFunction(function_0) {
  SCM_DeclareLocalVariable(v_25,0);
  SCM_DeclareLocalDottedVariable(X,1);
  SCM v_27; SCM v_26;
  return (v_26=SCM_Free(I),
          (v_27=X,
           SCM_invoke1(v_25,
                       SCM_cons(v_26,
                                v_27))));
}
SCM_DefineClosure(function_1, );
SCM_DeclareFunction(function_1) {
  SCM_DeclareLocalVariable(v_24,0);
  SCM_DeclareLocalVariable(I,1);
  return SCM_invoke1(v_24,
                     SCM_close(SCM_CfunctionAddress
                               (function_0),-2,1,I));
}
SCM_DefineClosure(function_2, SCM v_15; SCM CNTER; SCM TMP; );
SCM_DeclareFunction(function_2) {
  SCM_DeclareLocalVariable(v_21,0);
  SCM v_20; SCM v_19; SCM v_18; SCM v_17;
  return (v_17=(SCM_Content(SCM_Free(TMP))=v_21),
          (v_18=SCM_Free(CNTER),
           ((v_18 != SCM_false)
            ? (v_19=SCM_Free(CNTER),
               (v_20=SCM_Content(SCM_Free(TMP)),
                SCM_invoke2(v_19,
                            SCM_Free(v_15),
                            v_20)))
            : SCM_invoke1(SCM_Free(v_15),
                          SCM_CheckedGlobal(INDEX)))));
}
SCM_DefineClosure(function_3, );
SCM_DeclareFunction(function_3) {
  SCM_DeclareLocalVariable(v_15,0);
  SCM_DeclareLocalVariable(CNTER,1);
  SCM_DeclareLocalVariable(TMP,2);
  SCM v_23; SCM v_22; SCM v_16;
  return (v_16=TMP= SCM_allocate_box(TMP),
          (v_22=CNTER,
           (v_23=SCM_close(SCM_CfunctionAddress(function_1),2,0),
```

```
                SCM_invoke2(v_22,
                            SCM_close(SCM_CfunctionAddress
                                      (function_2),1,3,v_15,CNTER,TMP),
                            v_23))));
}
SCM_DefineClosure(function_4, );
SCM_DeclareFunction(function_4) {
  SCM_DeclareLocalVariable(v_8,0);
  SCM_DeclareLocalVariable(F,1);
  SCM v_11; SCM v_10; SCM v_9; SCM v_12; SCM v_14; SCM v_13;
  return (v_13=thing1,
          (v_14=SCM_CheckedGlobal(INDEX),
           (v_12=SCM_Plus(v_13,
                          v_14),
            (v_9=(INDEX=v_12),
             (v_10=F,
              (v_11=SCM_CheckedGlobal(INDEX),
               SCM_invoke2(v_10,
                           v_8,
                           v_11)))))));
}
SCM_DefineClosure(function_5, );
SCM_DeclareFunction(function_5) {
  SCM_DeclareLocalVariable(v_1,0);
  return v_1;
}
SCM_DefineClosure(function_6, );
SCM_DeclareFunction(function_6) {
  SCM v_5; SCM v_7; SCM v_6; SCM v_4; SCM v_3; SCM v_2; SCM v_28;
  return (v_28=thing0,
          (v_2=(INDEX=v_28),
           (v_3=SCM_close(SCM_CfunctionAddress(function_3),3,0),
            (v_4=SCM_close(SCM_CfunctionAddress(function_4),2,0),
             (v_6=thing2,
              (v_7=thing3,
               (v_5=SCM_cons(v_6,
                             v_7),
                SCM_invoke3(v_3,
                            SCM_close(SCM_CfunctionAddress
                                      (function_5),1,0),
                            v_4,
                            v_5)))))))));
}
/* Expression: */
void main(void) {
  SCM_print(SCM_invoke0(SCM_close(SCM_CfunctionAddress
                                  (function_6),0,0)));
  exit(0);
}
/* End of generated code. */
```

On y constate un accroissement de la taille du fichier produit en nombre de lignes (de 76 à 130), en nombre de fonctions engendrées par compilation (de 3 à 6) et enfin en nombre de variables locales utilisées (de 2 à 22). La taille de l'exécutable produit prend au passage quelque embonpoint. La durée de calcul s'allonge un peu : de l'ordre de 50% sur ma machine. Après avoir modifié `main` pour répéter le calcul 10 000 fois (sans

appel à `SCM_print`) et compilé avec `-O` comme niveau d'optimisation. La durée de calcul passe de 1,1 secondes à 1,7 secondes. Le calcul consomme dans les deux cas le même espace pour les continuations mais dans le premier cas effectuait ces allocations dans la pile (pour lequel le matériel et le compilateur C déploient (en principe) des trésors d'ingéniosité) tandis que, dans le second cas, ces allocations sont effectuées dans le tas et détruisent la localité des références. On peut bien sûr largement diminuer ces désavantages comme le montre [AS94].

```
% gcc -ansi -pedantic chap10kex.c scheme.o schemeklib.o
% time a.out
(2 3)
0.000u 0.020s 0:00.00 0.0% 3+3k 0+0io 0pf+0w
% size a.out
text    data    bss     dec     hex
32768   4096    32      36896   9020
```

La transformation qui vient d'être décrite n'annihile pas le problème de débordement de pile dû à la non préservation de la propriété de récursion terminale. **[☞ p. 382]** Il est en fait agravé même par la transformation qui effectue plus d'applications que naguère et qui, surtout, ne revient jamais de ces applications. Les fonctions s'appellent mais ne retournent jamais un résultat : la pile de C débordera donc au bout d'un certain temps si le calcul ne s'achève pas avant. Une solution est de faire un `longjmp` de temps en temps pour abaisser le niveau de la pile.

10.11 Interface avec C

Un des intérêts de notre petit compilateur est que la représentation des données, ainsi que l'adoption du protocole d'appel des fonctions de C, le rend particulièrement apte à convoler avec C. Ce couplage (pour *foreign interface*) est très important car il permet de détourner à son profit les très vastes bibliothèques de fonctions (utiles) écrites en C. Nous allons montrer un exemple d'un tel couplage choisi pour illustrer l'intérêt et les problèmes associés.

La fonction `system` pour UN⋆X prend une chaîne de caractères, la soumet à l'interprète de commandes `sh` puis retourne un entier correspondant à son code de retour. On supposera disposer d'une macro `defforeignprimitive` permettant de déclarer au compilateur l'interface que possède cette fonction `system` :

```
(defforeignprimitive system int ("system" string) 1)
```

Dûment prévenu, le compilateur saura lorsqu'il verra `(system` π`)` appeler la fonction `system` après avoir vérifié que son argument est une chaîne et convertira le résultat (un entier C) en un entier Scheme. L'identité de codage des chaînes C et Scheme rend leur conversion aisée mais d'autres types peuvent poser d'ardus problèmes [RM92, DPS94a] notamment les fonctions de renvoi (ou *call-back*).

Ce n'est malheureusement pas suffisant car il faut encore assurer que la fonction `system` puisse être appelée par calcul (ou par `apply`) ce qui impose qu'il y ait une valeur de Scheme représentant cette fonction. Nous laisserons ce délicat problème de côté.

10.12 Conclusions

Ce chapitre a présenté un compilateur de Scheme vers C. On pourrait adapter ce compilateur à la bibliothèque d'exécution que procurent implicitement des évaluateurs écrits en C tels que Bigloo [Ser94], SIOD [Car94] ou SCM [Jaf94]. On aura pu apprécier dans ce chapitre les problèmes de la compilation vers un langage de haut niveau, le fossé qui sépare Scheme de C mais aussi les bienfaits qui peuvent résulter d'une coopération raisonnable entre ces deux langages.

La mise en parallèle du compilateur vers C et du compilateur vers du code-octet est fortement conseillée. On pourra à volonté marier les techniques vues naguère aux nouvelles ici exposées et, par exemple, *(i)* changer la compilation des citations pour qu'elles soient lues par `read` ; *(ii)* s'affranchir de la pile de C (et des fonctions de C) au profit d'une pile de récursion dévolue à Scheme ; *(iii)* étendre le compilateur pour compiler des modules indépendants . . .**[☞ p. 213]** On pourra aussi mesurer le coût en taille de la bibliothèque d'exécution et en vitesse de calcul que représenterait l'adjonction d'une fonction telle que `load` ou, plus simplement, `eval`. Nous laisserons aussi en exercice le soin d'incorporer ces fonctionnalités et d'auto-amorcer l'évaluateur ainsi étoffé.

10.13 Exercices

Exercice 10.1 : L'invocation des fermetures pourrait emprunter la même technique que celle qui est utilisée pour les fonctions prédéfinies d'arité fixe, c'est-à-dire adopter un modèle que l'on pourrait qualifier de $f(f, x, y)$. Modifiez ce qu'il faut pour ainsi améliorer le compilateur.

Exercice 10.2 : L'accès aux variables globales serait plus efficace si l'on supprimait, à la lecture, le passage par le test `SCM_CheckedGlobal` qui vérifie si la variable est bien initialisée. Concevez une analyse d'initialisation cherchant à caractériser les références pour lesquelles on est sûr que la variable est initialisée.

Projet 10.3 : Le code produit par le compilateur de ce chapitre engendre du C conforme à la norme C de l'ISO [ISO90] ; modifiez ce qu'il faut pour engendrer du C K&R [KR78].

Projet 10.4 : Adaptez la génération de code de ce chapitre pour produire du C++ [Str86] plutôt que du C.

10.14 Lectures recommandées

La compilation vers C a fait l'objet de nombreux travaux ces dernières années et est excellement traitée dans les thèses de Nitsan Séniak [Sén91] et Manuel Serrano [Ser94]. Les mordus s'abîmeront dans les sources de Bigloo [Ser94].

11

Essence d'un système à objets

Objets, que serions-nous sans vous ! Ce chapitre décrit le système d'objets que nous avons utilisé au long de ce livre. Nous nous sommes volontairement limité à un ensemble restreint de caractéristiques afin de ne pas encombrer la description qui va suivre et de la limiter ainsi à sa *sustantificque mouelle* [Rab55]. Le système est appelé MEROON mais un tel système est complexe et nécessite beaucoup de soins si l'on souhaite qu'il soit à la fois efficace, portable et porté. Il en résulte un système doté d'une ossature fortement perturbée par ce souci de portabilité. Nous allons donc plutôt exposer une version réduite de MEROON que nous baptiserons MEROONET [1].

Lisp et les objets ont une longue histoire en commun puisque l'un des tout premiers langages à objets, Smalltalk 72, fut tout d'abord implanté en Lisp. Depuis cette date, Lisp, en tant qu'excellent langage de développement de langages, a servi de berceau à d'innombrables études sur les objets ; citons seulement Flavors (développé par Symbolics pour le système de fenêtrage des machines Lisp), qui a expérimenté l'héritage multiple, et Loops (créé au laboratoire Xerox Parc), qui a introduit la notion de fonctions génériques. Ces travaux culminent avec la définition de CLOS (COMMON LISP Object System) et de TEΛOΣ (the EULISP object system). Ces deux derniers systèmes unifient la majeure partie des caractéristiques des précédents systèmes d'objets tout en les immergeant dans le système de types du langage sous-jacent, satisfaisant ainsi à l'une des règles propres à ces systèmes : tout doit être objet.

Comparativement aux systèmes d'objets que l'on peut trouver dans les autres langages, deux caractéristiques particulières distinguent ceux de Lisp :

— Les fonctions génériques et leurs multi-méthodes (technique dite du *multiple dispatch*)
L'envoi de message, qui s'écrit `(`*message arguments*`...)`, ne se distingue syntaxiquement pas de l'appel de fonctions qui est `(`*fonction arguments*`...)`. Les multi-méthodes ne supposent pas que le receveur du message soit unique (même si c'est le plus souvent le cas) mais déterminent la méthode en fonction de la classe de tous les arguments significatifs. Imprimer, par exemple, un nombre en hexadécimal sur un flux n'est ainsi ni une méthode sur les nombres, ni une méthode sur les flux mais une méthode s'exerçant sur la conjonction

1. Ces systèmes sont tous deux disponibles par `ftp` anonyme selon les instructions données en page xix.

nombre × flux.

— La réflexivité
La réflexivité est la qualité dont sont dotés les systèmes qui, en bref, savent parler d'eux-mêmes. Ces systèmes introduisent notamment la notion de *méta-classes*. La classe de tout objet est elle-même un objet qui a donc nécessairement lui aussi une classe que l'on nomme méta-classe qui elle aussi est un objet et ainsi de suite. L'ensemble des comportements de ces méta-objets est nommé le *meta object protocol* (ou MOP) dont les caractéristiques contrôlent les possibilités du système d'objets. Plusieurs niveaux de réflexivité sont, par exemple, possibles depuis l'auto-description jusqu'à l'auto-modification. Il devient alors possible de spécifier la représentation physique des objets tout en conservant leurs propriétés d'héritage et par exemple s'accomoder d'un système d'archivage persistant ou d'une base de données. L'auto-description est aussi un point important pour l'introspection et notamment la conception d'outils de mise au point des programmes puisqu'ils permettent d'inspecter les objets, connaître leur structure pour par exemple les imprimer, les compiler, **[☞ p. 343]** les faire migrer d'une machine à une autre [Que94] etc.

C'est cette longue histoire que nous allons perpétuer en présentant MEROONET. Toutefois les restrictions ayant présidé à la construction de ce dernier ne doivent point être perçues comme attristantes au regard des excellentes qualités qui sont son apanage :

— Toutes les valeurs manipulables en Lisp ou Scheme peuvent être représentées par des objets de MEROONET y compris les vecteurs sans restriction d'héritage.
— MEROONET s'auto-décrit car les classes sont des objets à part entière dûment inspectable ; la régression infinie étant évitée comme enseigné en ObjVlisp [Coi87, BC87].
— La présence de fonctions génériques à la CLOS [BDG+88] mais sans multi-méthodes.
— Le code, enfin, est loin d'être inefficace.

Quelques implantations de systèmes à objets en Lisp ou Scheme ont déjà été décrites dans la littérature [AR88, Kes88, Coi87, MNC+89, Del89, KdRB92]. Le système MEROONET diffère d'elles sur quelques points :

— MEROONET adopte les fonctions génériques de Common Loops [BKK+86] mais pas les multi-méthodes.
— MEROONET procure des classes comme objets de première classe comme en ObjVlisp, ce qui autorise une remarquable auto-description.
— MEROONET abhorre l'héritage multiple ! Lorsque l'on saura ce que l'héritage multiple signifie et lorsque les problèmes notamment ceux posés en [DHHM92] seront résolus, MEROONET révisera (peut-être) son aversion.
— La représentation des objets sous forme de valeurs contiguës, c'est-à-dire de vecteurs, autorise moyennant de nouveaux descripteurs de champs la description de données importées d'autres langages [QC88, Que90a].

Ce qui va suivre va décrire MEROONET avec, en contrepoint, les raisons de certains choix d'implantation. Certains de ces choix sont dus au langage d'implantation, Scheme, et seraient à revoir si l'on faisait une implantation directe de MEROONET, en

C par exemple. Pour tenter de simplifier l'exposé, nous expliquerons MEROONET de bas en haut et nous introduirons progressivement les fonctions sitôt qu'elles seront nécessaires. La documentation précise d'emploi apparaîtra mélangée à l'implantation ; on pourra toutefois consulter le court mode d'emploi que nous ne répéterons point ici. **[☞ p. 82]**

11.1 Fondations

La première décision d'implantation concerne la représentation des objets adoptée par MEROONET par des suites contiguës de valeurs. Comme nous devons exprimer cela en Scheme, nous choisirons, pour ce faire, les vecteurs et nous réserverons le premier indice (l'indice zéro) pour y stocker l'identité de la classe, ce qui permettra de remonter d'un objet à sa classe ; c'est ce que l'on nomme le lien d'appartenance (ou *instantiation link*). Le plus évident serait qu'un objet pointe sur sa classe directe (c'est-à-dire la classe la plus précise à laquelle il appartient). Nous préférons associer à chaque classe un numéro et stocker ce numéro dans les objets. L'impression des objets par les moyens standard de Scheme en sera améliorée car les classes sont des grosses et complexes structures de données dont le détail n'importe qu'à peu de personnes et qui pourrait, de plus, contenir des circularités qui rendraient l'impression naturelle, par `display`, cryptique et illisible [2]. On verra un autre intérêt à utiliser de tels numéros lorsque nous détaillerons le mécanisme d'appel des fonctions génériques et le test d'appartenance à une classe.

Numéroter les classes impose donc de pouvoir convertir nombre en classe. Toutes les classes seront donc archivées dans un vecteur. Comme les vecteurs ne sont pas extensibles en Scheme [3] ; il y aura donc un nombre maximal de classes possibles et aucune détection explicite d'anomalie si le nombre de classes déborde cette taille. La variable `*class-number*` désigne le premier numéro libre dans le vecteur `*classes*` ; comme certaines classes sont prédéfinies au moment de l'amorçage, ce numéro sera quelque peu incrémenté. Fidèle à ne pas encombrer le code de vérifications scrupuleuses, nous ne testerons pas si un nombre correspond bien à une classe, etc.

```
(define *maximal-number-of-classes* 100)
(define *classes* (make-vector *maximal-number-of-classes* #f))
(define (number->class n)
  (vector-ref *classes* n) )
(define *class-number* 0)
```

Manipuler des classes par leur numéro est peu parlant aussi imposerons-nous qu'elles soient toutes nommées. Il n'y aura donc pas de classe anonyme, les classes étant vues comme des entités plutôt statiques, c'est-à-dire non créées dynamiquement. La fonction `->Class` permet de convertir un symbole en la classe qui porte ce nom. Fidèle, toujours, à ne pas grever le code de vérifications superflues, la fonction `->Class` retournera la dernière classe portant le nom recherché, ce qui aide un peu à la redéfinition de classes sans toutefois lui donner un sens vraiment précis : on se réfrénera

2. Pour certains GC et si toutes les valeurs de Scheme étaient des objets, un pointeur de moins par objet, à inspecter ou à mettre à jour, peut aussi être un gain.

3. Ils peuvent l'être en COMMON LISP.

donc de définir plusieurs fois une même classe sans raison.

```
(define (->Class name)
  (let scan ((index (- *class-number* 1)))
    (and (>= index 0)
         (let ((c (vector-ref *classes* index)))
           (if (eq? name (Class-name c))
               c (scan (- index 1)) ) ) ) ) )
```

En ce qui concerne les fonctions génériques, l'implantation exige qu'elles soient nommées et que l'on puisse accéder à leur structure interne ; nous les conserverons donc toutes en une liste `*generics*`. La fonction `->Generic` convertira, quant à elle, un nom en une fonction générique. Nous reverrons ces points par la suite.

```
(define *generics* (list))
(define (->Generic name)
  (let lookup ((l *generics*))
    (if (pair? l)
        (if (eq? name (Generic-name (car l)))
            (car l)
            (lookup (cdr l)) )
        #f ) ) )
```

11.2 Représentation des objets

Les objets de MEROONET sont des suites de valeurs contiguës. Comme nous souhaitons que toute valeur de Scheme soit assimilable à un objet définissable en MEROONET, nous devons prendre en compte la notion de vecteur[4] et ainsi introduisons-nous la notion de champ indexé qui au lieu de contenir une unique valeur en contient un certain nombre. Cette caractéristique existait déjà en Smalltalk [GR83] mais excluait alors que l'on puisse sainement hériter d'une classe possédant un champ indexé : MEROONET n'a pas ce genre de restrictions. Un problème subsiste pour les chaînes de caractères qui sont souvent, et pour des raisons d'efficacité, primitives (c'est-à-dire non construites) dans les langages de programmation. On pourrait cependant les représenter par des vecteurs de caractères, ce qui n'est pas totalement idiot si l'on songe que les soucis d'internationalisation poussent à l'adoption de jeux de caractères de plus en plus larges et représentés sur deux voire quatre octets. Si toutefois l'on souhaite encore des caractères d'un octet et comme Scheme ne procure pas de structures de données permettant de mélanger des répétitions de tels caractères (d'un octet) et des valeurs quelconques (des pointeurs), alors MEROONET ne saura pas simuler efficacement des chaînes de caractères.

MEROONET propose donc deux types de champs : les champs normaux (décrits par des objets de la classe `Mono-Field`) et les champs indexés (de la classe `Poly-Field`). Un champ normal sera implanté en une composante de vecteur tandis qu'un champ indexé sera représenté, un peu comme les chaînes en Pascal, c'est-à-dire par la suite des valeurs indexées en autant de composantes que nécessaire préfixées par le nombre de ces composantes. Pour fixer les idées, la classe des points pourra être définie comme :

---

4. et ainsi simuler les vecteurs par des objets de MEROONET, qui sont eux-mêmes représentés par des vecteurs. Le surcoût n'est que d'une seule coordonnée pour contenir le lien d'appartenance.

```
(define-class Point Object (x y))
```

En supposant que 7 soit le numéro associé à la classe `Point`, le point valeur de `(make-Point 11 22)` sera représenté par le vecteur `#(7 11 22)`. Un polygone peut être défini comme une suite de points représentant les différents segments que forment ses côtés ; on fera en outre hériter `Polygon` de `Point` pour fixer son point de référence.

```
(define-class Polygon Point ((* side)))
```

On voit ici la seconde syntaxe possible (parenthésée) de définition de champs. Un champ normal voit son nom préfixé par un signe égal tandis qu'un champ indexé sera étoilé [5]. Enfin si l'on souhaite des polygones colorés, on créera une nouvelle sous-classe :

```
(define-class ColoredPolygon Polygon ((= color)))
```

Chaque définition de classe est accompagnée d'une flopée de fonctions et notamment d'une fonction de création d'objets dont le nom est intelligemment formé par le préfixe `make-` et le nom même de la classe. La création d'un objet passe par la définition de tous ses champs. En particulier, la taille des champs indexés doit être définie : on préfixera donc les valeurs indexées devant figurer dans de tels champs par leur nombre. Soit à définir un triangle, c'est-à-dire un polygone à 3 côtés et, de surcroît, de couleur orange. On écrira ce qui suit et l'on contemplera la représentation sous forme de vecteurs :

```
(make- coloredPolygon
 11                                                          ; x
 22                                                          ; y
 3 (make-Point 44 55) (make-Point 66 77) (make-Point 88 99)  ; 3 sides
 'orange )                                                   ; color
→ #(9                ; { (Class-number ColoredPolygon-class)}
       11            ; x
       22            ; y
       3             ; number of sides
       #(7 44 55)    ; side[0]
       #(7 66 77)    ; side[1]
       #(7 88 99)    ; side[2]
       orange )      ; color
```

Les objets de MEROONET sont donc tous représentés par des vecteurs dont la première composante détient le numéro de la classe de l'objet. Nous utiliserons le mot « déplacement » (*offset*) lorsque nous parlerons de déplacement au sein du vecteur représentant un objet et d'« index » (*index*) lorsqu'il s'agira de manipuler un champ indexé. Puisqu'une composante est réservée pour le lien d'appartenance, le premier déplacement licite est donné par la constante `*starting-offset*`. La classe d'un objet peut être connue grâce à la fonction `object->class` [6]. Enfin on reconnaîtra si une valeur est un objet de MEROONET grâce à la fonction `Object?`. Cette fonction pose un grave problème puisque Scheme n'autorise pas la création de nouveaux types de données. Représenter les objets par des vecteurs les rend indistinguables les uns

---

5. L'étoile a été choisie pour rappeler la multiplicité comme dans la notation des expressions régulières à la Kleene.

6. C'est l'équivalent de la fonction `class-of` de COMMON LISP, EULISP et IS-Lisp. Nous l'avons nommé différemment afin de ne pas introduire de confusion et ainsi pouvoir, par exemple, mélanger plusieurs systèmes d'objets.

des autres. Le prédicat `Object?` reconnaît tous les objets de MEROONET mais malheureusement aussi les vecteurs d'entiers, etc. Pour en finir de ce prélude, voici les conventions concernant les noms de variables utilisées :

`o`, `o1`, `o2` ...	objet
`v` ...	valeur (entier, paire, fermeture etc.)
`i` ...	index

```
(define *starting-offset* 1)
(define (object->class o)
  (vector-ref *classes* (vector-ref o 0)) )
(define (Object? o)
  (and (vector? o)
       (integer? (vector-ref o 0)) ) )
```

11.3 Définition de classes

Pour définir une classe, il existe la forme `define-class`. Cette forme de définition prend trois arguments successivement :

— le nom de la classe à définir ;
— le nom de la super-classe ;
— la spécification de ses champs propres.

La classe qui sera créée héritera des champs figurant dans la définition de sa super-classe ainsi que de tous les comportements que sa super-classe peut avoir vis-à-vis de toutes les fonctions génériques existantes. On nomme ces deux types d'héritage : *héritage des champs* et *héritage des méthodes*.

La forme `define-class` a pour syntaxe :

`(define-class` nom-classe nom-super-classe `(` liste-des-champs `) )`

Un champ figurant dans la liste des champs peut être mentionné soit directement par son nom (c'est alors un champ normal), soit, dans une liste, préfixé par un signe indiquant sa nature : un signe égal si normal, une étoile si indexé.

Outre la définition de la classe, un certain nombre de fonctions d'accompagnement sont automatiquement créées pour votre plaisir immédiat :

— Un prédicat de reconnaissance des objets de cette classe. Son nom est formé par le nom de la classe suivi du suffixe usuel des prédicats en Scheme : le point d'interrogation.
— Un allocateur d'objets retournant de nouveaux membres de la classe. Aucun champ n'est initialisé ; leur valeur initiale est donc quelconque. Les tailles des champs indexés doivent cependant être précisées lors de l'allocation ; l'allocateur prend donc autant de tailles en arguments que la classe contient de champs indexés. Le nom de l'allocateur est formé du nom de la classe préfixé par `allocate-`.
— Un allocateur retournant de nouveaux membres de la classe et spécifiant la valeur initiale de tous les champs. Les valeurs constituant un champ indexé sont préfixées par la taille de ce champ. Le nom de cet allocateur est formé du nom de la classe préfixé par `make-`.

— Des sélecteurs de champ normaux ou indexés. Pour chaque champ de la classe est forgé un sélecteur en lecture ayant pour nom le nom de la classe suivi d'un tiret suivi du nom du champ. Le sélecteur en écriture a le nom du lecteur correspondant préfixé par `set-` et suffixé par l'usuel point d'exclamation soulignant les modificateurs physiques. Les sélecteurs de champs indexés prennent un argument supplémentaire : un index.
— La taille de chacun des champs indexés peut être connue grâce à une fonction de nom formé par le nom du sélecteur en lecture associé suffixé par `-length`.

Afin d'autoriser quelques opérations réflexives, la classe qui est créée et qui est membre de la classe `Class` est la valeur de la variable globale ayant pour nom le nom de la classe suffixé par `-class`. Pour rendre plus aisée l'énumération précédente, voici les descriptions des fonctions ou variables engendrées par la déclaration [7]

`(define-class ColoredPolygon Point (color))` :

```
(ColoredPolygon? o)                                  → a boolean
(allocate-ColoredPolygon sides-number)               → a polygon
(make-ColoredPolygon x y sides-number sides...color) → a polygon
(ColoredPolygon-x o)                                 → a value
(ColoredPolygon-y o)                                 → a value
(ColoredPolygon-side o index)                        → a value
(ColoredPolygon-color o)                             → a value
(set-ColoredPolygon-x! o value)                      → unspecified
(set-ColoredPolygon-y! o value)                      → unspecified
(set-ColoredPolygon-side! o value index)             → unspecified
(set-ColoredPolygon-color! o value)                  → unspecified
(ColoredPolygon-side-length o)                       → a length
ColoredPolygon-class                                 → a class
```

La création de classes passe par la forme spéciale `define-class`. Cette forme spéciale sera implantée comme une macro avec malheureusement tous les problèmes que cela comporte. Le premier de ces problèmes est que `define-class` est une macro possédant un état interne : l'arbre d'héritage, et que le système de macros de Scheme [CR91b] n'autorise pas ce genre de choses. Aussi allons-nous supposer disposer d'une nouvelle macro s'intitulant `define-meroonet-macro` dont le but est de définir des macros sans restriction sur la façon dont se passe l'expansion. On peut coder `define-meroonet-macro` dans toutes les implantations existantes mais pas en Scheme portable.

L'état interne de `define-class` est-il nécessaire ? Il l'est en MEROONET pour la raison suivante : lorsqu'une classe est définie, un certain nombre de fonctions d'accès à ses champs sont créées. Par exemple, pour la définition de la classe `Point`, dotée des champs `x` et `y`, les lecteurs `Point-x` et `Point-y` sont créés. Lorsque l'on définit la classe `Polygon`, MEROONET n'impose pas que le seul moyen d'obtenir les champs `x` et `y` (que les polygones héritent de `Point`) soient les fonctions `Point-x` et `Point-y` : MEROONET crée donc les fonctions `Polygon-x` et `Polygon-y`. On notera que la définition de la classe `Polygon` ne mentionne pas les champs hérités et que le seul moyen de les connaître passe par le nom de la super-classe. L'état interne qu'entretient `define-class` associe donc à chaque classe les noms (et types) de ses champs.

---

7. Nous avons pris comme convention de nommer les champs indexés par des mots au singulier. Il est plus naturel d'écrire `(ColoredPolygon-side` o i`)` pour obtenir le $i$ème côté d'un polygone et d'employer `side` plutôt que `sides` dans ce contexte.

On aurait pu éviter cet inconvénient en adoptant une autre convention pour nommer des sélecteurs comme, par exemple, que leur nom ne mentionne pas de classe. Supposons, pour illustrer ce cas, que le lecteur du champ `x` soit nommé `get-x`, alors la définition de la classe `Polygon` modifierait la valeur de `get-x` pour indiquer comment extraire le champ `x` des polygones. Le plus simple alors est de faire de `get-x` une fonction générique qui acquiert de nouvelles méthodes au fil des définitions de classes. Cette technique est celle qu'emploie CLOS où il est possible de mentionner, lors de la spécification d'un champ, la fonction générique à laquelle il faut ajouter la méthode de lecture de ce champ. Cette décision rend les fonctions génériques modifiables, ce qui peut être perçu comme un inconvénient notamment pour les optimisations statiques de compilation car il est difficile de fonder quoi que ce soit sur une valeur changeant sans restriction.

Nous avons pris inversement la décision de faire des sélecteurs de pures fonctions non modifiables, ce qui facilite leur intégration en ligne. Cela induit cependant de subtils problèmes comme la nature de la différence qui peut exister entre `Point-x` et `Polygon-x` ? Fondamentalement, ces fonctions extraient toutes deux le même champ `x` mais, plus précisément, la première extrait `x` d'un point tandis que la seconde l'extrait d'un polygone. Il semble donc normal que la forme `(Polygon-x (make-Point 11 22))` soit erronée et qu'en conséquence les sélecteurs `Point-x` et `Polygon-x` soient différents. Outre la précédente byzantinerie, l'énorme inconvénient de cette décision est le grand nombre de variables et de fonctions globales qu'elle consomme. C'est un problème de place en mémoire mais non en cerveau puisque tous les noms sont régulièrement formés et que seule la règle de dénomination est à retenir.

La connaissance des champs de la super-classe (maintenue par l'état interne de `define-class`) ne joue pas que sur la façon de nommer les sélecteurs, les allocateurs en ont aussi besoin afin que leur arité soit connue et que leur compilation soit efficace. Ces remarques seront reprises et amplifiées dans les sections suivantes.

Nous adopterons donc la solution suivante pour MEROONET : la définition d'une classe passe par la construction d'un objet, membre de la classe `Class` et son insertion dans la hiérarchie des classes, toutes actions effectuées par `register-class`. Les fonctions accompagnant cette classe sont, quant à elles, engendrées par la fonction d'expansion `Class-generate-related-names`. Conformément à la précédente description, nous définirons temporairement `define-class` comme suit :

```
(define-meroonet-macro (define-class name super-name
                                     own-field-descriptions )
  (let ((class (register-class name super-name
                               own-field-descriptions )))
    (Class-generate-related-names class) ) )
```

Cette programmation ne résiste pas à la compilation car il n'y a pas concordance des temps. La classe est créée puis insérée dans l'arbre d'héritage pendant la macroexpansion, tandis que les fonctions d'accompagnement sont créées lors de l'évaluation du code expansé. Supposons que l'on compile un fichier contenant la définition de la classe `Polygon`, le code résultant (que l'on peut voir comme un fichier avec le suffixe `.o` comme dans les compilateurs vers C, comme KCL [YH85], Scheme→C [Bar89] ou Bigloo [Ser94], ou avec un suffixe comme `.fasl`, comme dans beaucoup d'autres compilateurs) ne contient que le compilé de l'expansion, c'est-à-dire la définition des

fonctions d'accompagnement. La classe a bien été créée mais dans l'état mémoire du compilateur et en aucun cas dans la mémoire du système Lisp ou Scheme qui sera liée à ce fichier `.o` ou qui chargera dynamiquement le fichier `.fasl`. La classe s'est donc évanouie et a disparu lorsque le compilateur aura achevé son travail de compilation.

Il faut donc que la fabrication de la classe apparaisse dans l'expansion mais si elle n'apparaît que là, alors on ne peut plus demander à la classe d'engendrer les fonctions d'accompagnement puisque l'on a besoin, à la macroexpansion, de connaître les champs de la super-classe pour engendrer tous les sélecteurs prévus [8]. Il est donc nécessaire que la classe soit connue *de* la macroexpansion et qu'elle apparaisse aussi *dans* la macroexpansion. Cette double existence permet de compiler dans un même fichier les définitions d'une classe et de ses sous-classes comme par exemple, `Point` et `Polygon`. Pour éviter que MEROONET ne fabrique deux fois [9] la classe lorsqu'interprété, une fois à la macroexpansion puis à l'évaluation du code expansé, nous utiliserons l'astuce suivante. Lorsqu'une classe est définie pendant la macroexpansion, elle est stockée dans la variable globale `*last-defined-class*` puis, à l'évaluation, la classe n'est recréée que si `*last-defined-class*` est vide.

```
(define *last-defined-class* #f)
(define-meroonet-macro (define-class name supername
                                     own-field-descriptions )
  (set! *last-defined-class* #f)
  (let ((class (register-class name supername
                               own-field-descriptions )))
    (set! *last-defined-class* class)
    `(begin
      (if (not *last-defined-class*)
        (register-class ',name ',supername
                        ',own-field-descriptions ) )
      ,(Class-generate-related-names class) ) ) )
```

Autres difficultés

Toujours pour illustrer les difficultés des macros sur un exemple de la vie réelle à savoir MEROONET, on notera que l'ordre selon lequel les macroexpansions sont effectuées est important lorsque l'on a à utiliser des macros à état interne comme l'est `define-class`. En effet, considérez la forme composée suivante :

```
(begin (define-class Point Object (x y))
       (define-class Polygon Point ((* side))) )
```

Si l'expansion suit l'ordre gauche-droite alors la classe `Point` est définie et peut servir à l'expansion de la définition de la classe `Polygon`. Dans le cas contraire, la classe `Polygon` ne peut être construite car sa super-classe est encore inconnue. On peut tourner ce problème par un mécanisme (bien compliqué) différant la construction de la classe `Polygon` tant que la définition de `Point` n'est pas expansée. L'expansion de la définition de `Polygon` serait donc vide tandis que celle de `Point` contiendrait les deux définitions.

---

8. Rappelons qu'aucun dispositif linguistique ne permet, en Scheme et à l'exécution, de créer de nouvelles variables globales à nom calculé.

9. Aucun effort n'a été fait en MEROONET pour donner un sens précis à la redéfinition de classes, c'est pourquoi nous cherchons à l'éviter.

Comment alors compiler la définition de `ColoredPolygon` dans un fichier différent de celui qui contient les définitions de `Point` et de `Polygon` ? Il est nécessaire d'obtenir les champs de sa super-classe. MEROONET ne s'encombre point de ce problème et requiert donc que toutes les classes soient compilées ensemble. MEROON adopte une stratégie différente. Pour hériter d'une classe elle doit être définie soit, avant et dans le même fichier, soit avec l'option `:prototype`, ce qui l'insère dans la hiérarchie de classes sans toutefois engendrer de code équivalent [10]. On écrira alors :

```
(define-class Polygon Point ((* side)) :prototype)
(define-class ColoredPolygon Polygon (color))
```

Une autre solution encore est d'utiliser des modules et des mécanismes d'import/-export afin d'indiquer quels modules contiennent les informations pertinentes pour la compilation de tel autre. Cela revient à disposer d'une sorte de base de données, équivalent mimétique d'un état interne du compilateur projeté sur un système de fichiers.

11.4 Représentation des classes

Les classes de MEROONET sont représentées par des objets de MEROONET même. Cela facilite l'auto-description du système et l'écriture de méta-méthodes fondées sur la structure des classes plutôt que sur l'héritage. Par exemple la migration (de machine à machine) ou l'impression par défaut des objets de MEROONET peut se déduire de leur structure, c'est-à-dire de leur classe. Il existe donc une classe décrivant tous les objets de MEROONET : `Object`, qui agit comme racine de l'héritage. Les classes sont membres de la classe `Class` et les descripteurs de champs sont membres de `Mono-Field` ou `Poly-Field`, sous-classes toutes deux de `Field`. L'expansion d'une définition de classe utilise un grand nombre des fonctions accompagnant ces classes prédéfinies comme `Mono-Field?` ou `make-Poly-Field`, etc. Doit-on en déduire que lorsqu'une classe existe, on peut utiliser les fonctions l'accompagnant ? En termes plus précis, lorsque `Point` a été macroexpansé, peut-on utiliser la fonction `make-Point` ? Compte tenu de la précédente `define-class`, ce n'est pas possible puisque `make-Point` n'est pas créée à la macroexpansion mais à l'évaluation. Il en résulte que les méta-classes posent des problèmes délicats de compilation, que MEROONET n'abordera pas.

Minimalement, nous ne mettrons que le strict nécessaire dans les classes, à savoir leur nom, le numéro qui leur est associé, la liste de leurs champs, leur super-classe et la liste des numéros de leur sous-classes. Cette dernière information sera utile lorsque nous parlerons de fonctions génériques. On enregistre les numéros plutôt que les sous-classes elles-mêmes afin d'éviter une circularité de mauvais aloi au prix d'une petite décroissance d'efficacité. On peut condenser toutes ces informations en disant que `Class`, la classe des classes est définie par :

```
(define-class Class Object
   (name number fields super-class subclass-numbers) )
```

Les champs eux-mêmes seront aussi décrits comme des objets de MEROONET. Les champs sont définis par leur type, leur nom et la classe qui les a introduit. Pour ce der-

10. C'est en fait un peu plus compliqué en MEROON. L'option `:prototype` s'expanse en un test qui vérifie à l'évaluation que le prototype est conforme à la véritable classe qui doit alors exister.

nier champ, dont on verra l'intérêt notamment pour la fonction générale `field-value`, on stockera le numéro de la classe plutôt que la classe elle-même afin de rendre imprimable classes et champs. On aura donc :

```
(define-class Field Object (name defining-class-number))
(define-class Mono-Field Field ())
(define-class Poly-Field Field ())
```

Enfin la fonction suivante donnera l'illusion d'extraire d'un champ la classe l'ayant introduit, sans penser au numéro sous-jacent :

```
(define (Field-defining-class field)
  (number->class (careless-Field-defining-class-number field)) )
```

Bien sûr avant que MEROONET ne soit installé, on ne peut définir ces classes et pourtant elles sont nécessaires au fonctionnement même de MEROONET. Pour rompre ce problème d'auto-amorçage nous les créerons à la main comme indiqué ci-après :

```
(define Object-class
  (vector 1                                  ; it is a class
          'Object                            ; name
          0                                  ; class-number
          '()                                ; fields
          #f                                 ; no superclass
          '(1 2 3)                           ; subclass-numbers
          ) )
(define Class-class
  (vector
   1                                         ; it is also a class
   'Class                                    ; name
   1                                         ; class-number
   (list                                     ; fields
    (vector 4 'name                1)        ; offset= 1
    (vector 4 'number              1)        ; offset= 2
    (vector 4 'fields              1)        ; offset= 3
    (vector 4 'superclass          1)        ; offset= 4
    (vector 4 'subclass-numbers 1) )         ; offset= 5
   Object-class                              ; superclass
   '() ) )
(define Generic-class
  (vector
   1
   'Generic
   2
   (list
    (vector 4 'name                2)        ; offset= 1
    (vector 4 'default             2)        ; offset= 2
    (vector 4 'dispatch-table 2)             ; offset= 3
    (vector 4 'signature       2) )          ; offset= 4
    Object-class
    '() ) )
(define Field-class
  (vector
   1
   'Field
   3
   (list
    (vector 4 'name                   3) ; offset= 1
```

```
     (vector 4 'defining-class-number 3) ; offset= 2
     )
    Object-class
    '(4 5) ) )
(define Mono-Field-class
  (vector 1
          'Mono-Field
          4
          (careless-Class-fields Field-class)
          Field-class
          '() ) )
(define Poly-Field-class
  (vector 1
          'Poly-Field
          5
          (careless-Class-fields Field-class)
          Field-class
          '() ) )
```

Les classes sont ensuite installées comme il faut :

```
(vector-set! *classes* 0 Object-class)
(vector-set! *classes* 1 Class-class)
(vector-set! *classes* 2 Generic-class)
(vector-set! *classes* 3 Field-class)
(vector-set! *classes* 4 Mono-Field-class)
(vector-set! *classes* 5 Poly-Field-class)
(set! *class-number* 6)
```

Comme la gestion de MEROONET passe par l'entretien de la hiérarchie des classes, qui est représentée par des objets de MEROONET, quelques fonctions (comme les extracteurs de valeur `Class-number` ou `Class-fields`) sont nécessaires avant même d'être construites. Pour cela, nous introduirons leur équivalent de nom préfixé par `careless-`. Ce nom est justifié par leur définition qui ne vérifiera pas l'exacte nature de leur argument : MEROONET ne les utilise qu'à bon escient.

```
(define (careless-Class-name class)
  (vector-ref class 1) )
(define (careless-Class-number class)
  (vector-ref class 2) )
(define (careless-Class-fields class)
  (vector-ref class 3) )
(define (careless-Class-superclass class)
  (vector-ref class 4) )
(define (careless-Field-name field)
  (vector-ref field 1) )
(define (careless-Field-defining-class-number field)
  (vector-ref field 2) )
```

11.5 Les fonctions d'accompagnement

En ce qui concerne les fonctions d'accompagnement, nous utiliserons un utilitaire fort pratique : `symbol-concatenate`, pour former les nouveaux noms.

```
(define (symbol-concatenate . names)
  (string->symbol (apply string-append (map symbol->string names))) )
```

Plusieurs choix sont possibles pour engendrer les fonctions d'accompagnement. Le premier est d'engendrer directement leur code équivalent ; le second est d'engendrer une forme qui s'évaluera en une fermeture appropriée. Le second choix permet de factoriser et donc de partager le texte des fonctions sur lesquelles les fermetures sont construites mais empêche des optimisations fines du fait qu'il y aura plus d'appels à des fonctions calculées, non statiquement connues et donc non intégrables. Pour illustrer ce choix, voici comment serait défini l'allocateur de la classe `Polygon` dans ces deux cas :

```
(define allocate-Polygon
        (lambda (size)
          (let ((o (make-vector (+ 1 2 1 size))))
            (vector-set! o 0 (careless-Class-number Polygon-class))
            (vector-set! o 3 size)
            o )
(define allocate-Polygon (make-allocator Polygon-class))
```

Le premier choix permet de savoir statiquement que l'allocateur est une fonction unaire et que son corps n'use que de fonctions triviales, comme `make-vector`, et de lecture/écriture dans des vecteurs et qu'il peut donc être intégré en quelques instructions, voire même compilé en une macro en C si l'on compile vers celui-ci. À l'inverse, la seconde définition ne dit rien sur son arité et n'assure même pas que c'est une fonction qui sera liée à la variable `allocate-Polygon` [11]. Nous avons cependant privilégié ce second choix car le premier est un plus complexe à présenter et consomme plus de place en mémoire à l'exécution et à la compilation. Voici donc comment sont engendrées les fonctions d'accompagnement :

```
(define (Class-generate-related-names class)
  (let* ((name (Class-name class))
         (class-variable-name (symbol-concatenate name '-class))
         (predicate-name (symbol-concatenate name '?))
         (maker-name (symbol-concatenate 'make- name))
         (allocator-name (symbol-concatenate 'allocate- name)) )
    `(begin
       (define ,class-variable-name (->Class ',name))
       (define ,predicate-name (make-predicate ,class-variable-name))
       (define ,maker-name (make-maker ,class-variable-name))
       (define ,allocator-name (make-allocator ,class-variable-name))
       ,@(map (lambda (field)
                (Field-generate-related-names field class) )
              (Class-fields class) )
       ',(Class-name class) ) ) )
```

11.5.1 Prédicat

À toute classe est associée une fonction caractéristique, bref un prédicat. Ce prédicat répond Vrai sur tous les objets qui sont membres de cette classe, qu'ils soient membres directs de cette classe ou membres d'une quelconque de ses sous-classes. La célérité de ce prédicat est importante car, Scheme étant un langage sûr et sans typage statique, il faut sempiternellement vérifier les types ou les classes des objets manipulés. On peut aussi essayer de trouver à la compilation une estimation des types des

11. On ne sait même pas si, en Lisp, le calcul finit et si, en Scheme, il ne retourne qu'une unique valeur.

objets [12], factoriser les tests de types, se faire aider par des déclarations de l'utilisateur ou imposer que les programmes soient bien typés, par exemple dans les définitions de méthodes qui donnent certains renseignements sur la nature des arguments d'appel.

Le test d'appartenance à une classe est donc fondamental et repose sur le prédicat général `is-a?`. Pour ne pas entraver sa vélocité `is-a?` assume que l'objet et la classe sont bien un objet et une classe : il n'est donc pas question d'appliquer `is-a?` à n'importe quoi. En revanche, le prédicat associé à une classe sera moins restreint et testera tout d'abord si son argument est bien un objet de MEROONET. Enfin, nous introduirons un dernier prédicat qui sera utilisé pour son effet qui est de vérifier une appartenance et donner un message intelligible à défaut. Les erreurs n'étant pas standard en Scheme [13], les erreurs détectées par MEROONET font appel à la fonction `meroonet-error` qui n'est pas définie, ce qui est une manière portable d'obtenir une erreur !

```
(define (make-predicate class)
  (lambda (o) (and (Object? o)
                   (is-a? o class) )) )
(define (is-a? o class)
  (let up ((c (object->class o)))
    (or (eq? class c)
        (let ((sc (careless-Class-superclass c)))
          (and sc (up sc)) ) ) ) )
(define (check-class-membership o class)
  (if (not (is-a? o class))
    (meroonet-error "Wrong class" o class) ) )
```

Le prédicat `is-a?` est de complexité linéaire : on teste si l'objet a la classe souhaitée ou si sa super-classe est la classe souhaitée ou si sa super-super-classe . . . Ce n'est pas très gênant car c'est le premier cas qui est en général vrai à peu près une fois sur deux.

Un problème de régression infinie se présente toutefois dans `is-a?`. L'obtention de la super-classe passe par la fonction `careless-Class-superclass` et non directement par `Class-superclass`. La différence entre ces deux fonctions est que la première assume que son argument est une classe, ce qui est assuré dans le contexte de la fonction `is-a?`. Eussions-nous plutôt employé la seconde que `Class-superclass` aurait testé que son argument était bien une classe, ce qui aurait récursivement requis l'interception du prédicat `is-a?` même, grevant alors gravement l'efficacité de MEROONET.

11.5.2 Allocateur sans initialisation

MEROONET procure deux types d'allocateurs. Le premier crée des objets dont le contenu est quelconque tandis que le second crée des objets dont tous les champs sont initialisés. Les mêmes concepts existent en Lisp ou en Scheme : `cons` est l'allocateur avec initialisation des paires pointées, `vector` est l'allocateur avec initialisation des vecteurs. Un second allocateur existe pour les vecteurs : `make-vector` qui permet, sans second argument, de créer des vecteur de contenu non spécifié. MEROONET procurera ces deux types d'allocation.

---

12. La forme `define-method` donne notamment une indication sur la classe de la variable discriminante qui peut être mise à profit.

13. À l'instant d'écriture de cette phrase, c'est-à-dire pendant le règne du R^4RS [CR91b].

La notion de non-initialisation signifie que le contenu d'un objet non initialisé est quelconque. Deux sens au moins existent qui précisent cette notion. Pour des raisons de GC (et sauf pour les GC tolérant certaines ambiguïtés [BW88]), la non-initialisation signifie, en général, que l'objet est rempli d'entités connues de l'implantation. Que ces entités soient des valeurs de première classe ou non fonde donc les deux possibles sémantiques :

— Il existe une entité `#<uninitialized>` qui, lorsqu'elle apparaît dans un champ d'une structure de données, marque explicitement que ce champ ne contient aucune valeur. Lire un tel champ signale donc une erreur puisqu'il n'y a pas de valeur associée.

— Il n'existe pas de telle entité et donc les champs sont initialisés avec des valeurs normales mais quelconques, par exemple `nil` souvent en Lisp, `t` en Le-Lisp, `#f` dans beaucoup d'implantations de Scheme … Le contenu initial d'un champ est donc indéfini, c'est-à-dire non spécifié, bref quelconque, mais ce n'est pas une erreur que de lire ce champ, cela nuit à la portabilité de vos programmes.

Une troisième interprétation que nous qualifierions de « à la C » existe néanmoins : la lecture d'un champ non initialisé a des conséquences inconnues. Il est par conséquent plus que déconseillé de tenter de lire un tel champ. Remarquez que cette interprétation est encore moins spécifiée que les précédentes : il n'est par exemple pas obligatoire qu'une implantation détecte une erreur quand un tel champ est lu. Cela permet d'avoir une procédure de lecture n'effectuant aucun test sur la valeur lue qui, partant, est donc efficace. En contrepartie, lorsqu'un tel champ est lu, on peut obtenir l'infamant mais informatif « `Bus error, core dumped` » ou encore un résultat quelconque, sans rapport avec ce que l'on cherchait à calculer et que, probablement, l'on acceptera pour argent comptant. Le soin de la preuve que jamais, au grand jamais, l'on ne lit un champ non initialisé est laissé au programmeur.

La première interprétation requiert que la lecture d'un champ non initialisé signale une erreur, ce qui coûte donc un test explicite chaque fois qu'un champ susceptible d'être non initialisé est lu [Que93b]. Ce n'est pas le cas de la seconde qui se compare donc favorablement sur ce point avec C.

En termes d'allocation, la version à la C, est encore la plus efficace puisqu'il n'est pas besoin d'initialiser les champs ni même de les effacer. Paradoxalement, c'est l'allocation avec initialisation explicite qui est généralement plus efficace que l'allocation sans initialisation. En effet, allouer sans initialisation de la part de l'utilisateur requiert en fait d'initialiser avec `#<uninitialized>` de coût similaire à l'initialisation avec n'importe quelle valeur comme `#t`. Mais la plupart du temps, un champ qui n'est pas initialisé à l'allocation le sera dans les quelques instants qui suivent, ce qui, en définitive, double le travail. Alors que l'allocation avec initialisation explicite remplit définitivement tous les champs constants du premier coup.

La détection garantie qu'un champ est non initialisé selon la première interprétation existe en CLOS et est intéressante en Scheme si l'on réalise que les variables sont en fait représentées par des entités qui pourraient être décrites en MEROONET et pour lesquelles la détection de non-initialisation est obligatoire. **[☞ p. 56]** Comme nous avons décidé d'ignorer les formalistes, nous ne requerrons pas de MEROONET de traiter de la non-initialisation des champs et simplifierons ainsi son code : les champs

non initialisés seront remplis de `#f`.

On notera que la notion d'allocateur sans initialisation impose, pour être utile, que l'objet créé soit modifiable. C'est un point important car les objets immuables ont une nature plus proche des mathématiques que ceux qui ont un état interne, notamment en ce qui concerne l'égalité. **[☞ p. 116]** Les objets immuables se prêtent mieux à toutes sortes d'optimisation parce que leur contenu est garanti ne pas varier. Ainsi dans la fonction `make-allocator` ci-dessous et si le champ `number` de `Class` était immuable, serait-il possible de précalculer l'expression `(Class-number class)`, tout comme `(Class-fields class)` l'est, et ainsi clore directement sa valeur plutôt que de devoir le recalculer à chaque allocation.

Un allocateur sans initialisation explicite sera retourné par la fonction `make-allocator`. Celui-ci prend une classe et retourne une fonction qui prendra en arguments une liste d'entiers naturels spécifiant la taille de chacun des éventuels champs indexés figurant dans la définition de la classe. Le premier calcul consiste à déterminer la taille de la zone mémoire (du vecteur) à réserver (à allouer) ; pour cela, on parcourt récursivement la liste des champs et des tailles. Pour connaître la liste des champs, on extrait ceux-ci de la classe par `Class-fields`. La zone mémoire réservée ; il faut encore la structurer pour y placer les tailles des champs indexés : c'est l'objet d'une seconde boucle parcourant les champs et les tailles et maintenant un déplacement courant au sein de la zone mémoire allouée. Finalement, l'objet venant d'acquérir son squelette et son lien d'appartenance est retourné en valeur.

```
(define (make-allocator class)
  (let ((fields (Class-fields class)))
    (lambda sizes
      ;; compute the size of the instance to allocate
      (let ((room (let iter ((fields fields)
                             (sizes sizes)
                             (room *starting-offset*) )
                    (if (pair? fields)
                        (cond ((Mono-Field? (car fields))
                               (iter (cdr fields) sizes (+ 1 room)) )
                              ((Poly-Field? (car fields))
                               (iter (cdr fields) (cdr sizes)
                                     (+ 1 (car sizes) room) ) ) )
                        room ) )))
        (let ((o (make-vector room #f)))
          ;; setup instantiation link and skeleton of the instance
          (vector-set! o 0 (Class-number class))
          (let iter ((fields fields)
                     (sizes sizes)
                     (offset *starting-offset*) )
            (if (pair? fields)
                (cond ((Mono-Field? (car fields))
                       (iter (cdr fields) sizes (+ 1 offset)) )
                      ((Poly-Field? (car fields))
                       (vector-set! o offset (car sizes))
                       (iter (cdr fields) (cdr sizes)
                             (+ 1 (car sizes) offset) ) ) )
                o ) ) ) ) ) ) )
```

Plusieurs remarques peuvent être formulées. *(i)* Les allocateurs créés prennent en argument une liste de tailles et donc consomment des paires pointées dès qu'ils sont

associés à des classes possédant au moins un champ indexé : c'est un comble que de devoir allouer pour pouvoir allouer ! On verra une ébauche de solution par la suite. *(ii)* Les tailles superfétatoires sont purement et simplement ignorées sans compte-rendu d'erreur. *(iii)* L'allocation est un processus parcourant deux fois la liste des champs de la classe ce qui est assez coûteux surtout que cette liste n'étant pas variable, on sait dès la création de la classe si les membres directs de cette classe ont une taille fixe ou non. Ce point sera aussi corrigé par la suite. *(iv)* Si une taille n'est pas un entier naturel (non négatif donc), l'addition conduisant au calcul de la taille de la zone mémoire à allouer sera erronée et le message d'erreur cryptique. Pour être convivial, un test explicite devrait être effectué et un message d'erreur pertinent engendré. *(v)* Les champs ne peuvent être que des membres de `Mono-Field` ou `Poly-Field`. Il n'est pas possible ici d'ajouter ses propres classes de champs.

11.5.3 Allocateur avec initialisation

Les allocateurs avec initialisation sont créés de manière similaire mais nous améliorerons au passage l'efficacité des allocateurs d'objets de petite taille fixe. Les allocateurs sans initialisation de Scheme prennent, comme `make-vector` ou `make-string`, la taille de l'objet à allouer, suivie d'une éventuelle valeur de remplissage qui sera utilisée pour initialiser toutes les composantes. Les allocateurs avec initialisation, comme `cons`, `vector` ou `string`, prennent successivement toutes les valeurs d'initialisation, leur nombre devenant la taille de l'unique champ indexé pour `vector` et `string`. Comme nous autorisons d'initialiser plusieurs champs indexés à la fois, il est nécessaire de connaître la taille que prendra chacun d'entre eux puisque l'on ne peut l'inférer. MEROONET demande donc que soient préfixées les valeurs d'initialisation de champs indexés par leur nombre. Ainsi allouera-t-on un polygone à trois côtés en écrivant :

```
(make-ColoredPolygon 'x 'y 3 'Point0 'Point1 'Point2 'color)
```

Nous adopterons la technique suivante pour l'allocation avec initialisation : tous les arguments sont réunis en une liste `parms` ; un vecteur est alloué avec toutes ces valeurs et avec comme première composante le numéro de la classe appropriée. Il suffit alors de vérifier que l'objet a une structure correcte pour MEROONET. On parcourt alors les arguments et les champs de sa classe en vérifiant qu'il y au moins autant d'arguments que nécessaire. Au terme de cette vérification, l'objet est retourné. Cette technique est plus simple (et susceptible d'être plus efficace) que celle, apparemment plus naturelle qui consisterait à vérifier tous les arguments puis à allouer seulement après l'objet escompté.

```
(define (make-maker class)
  (or (make-fix-maker class)
      (let ((fields (Class-fields class)))
        (lambda parms
          ;; create the instance
          (let ((o (apply vector (Class-number class) parms)))
            ;; check its skeleton
            (let check ((fields fields)
                        (parms parms)
                        (offset *starting-offset*) )
              (if (pair? fields)
```

```
                (cond ((Mono-Field? (car fields))
                       (check (cdr fields)
                              (cdr parms)
                              (+ 1 offset) ) )
                      ((Poly-Field? (car fields))
                       (check (cdr fields)
                              (list-tail (cdr parms) (car parms))
                              (+ 1 (car parms) offset) ) ) )
              o ) ) ) ) ) ) )
```

On peut le constater : cet allocateur, d'une part, ignore les arguments superflus et, d'autre part, n'est pas très efficace car il alloue les arguments en une liste qu'il convertit ensuite en un vecteur. Comme cette liste, valeur de `parms`, est utilisée pour vérifier la structure de l'objet plutôt que, directement, les composantes de l'objet, on ne peut même pas espérer qu'il existe un compilateur suffisamment intelligent pour ne pas créer cette liste d'arguments. On a donc inséré une petite verrue en tête de `make-maker` pour améliorer le cas des allocateurs d'objets n'ayant pas de champ indexés.

```
(define (make-fix-maker class)
  (define (static-size? fields)
    (if (pair? fields)
        (and (Mono-Field? (car fields))
             (static-size? (cdr fields)) )
        #t ) )
  (let ((fields (Class-fields class)))
    (and (static-size? fields)
         (let ((size (length fields))
               (cn (Class-number class)) )
           (case size
             ((0)  (lambda () (vector cn)))
             ((1)  (lambda (a) (vector cn a)))
             ((2)  (lambda (a b) (vector cn a b)))
             ((3)  (lambda (a b c) (vector cn a b c)))
             ((4)  (lambda (a b c d) (vector cn a b c d)))
             ((5)  (lambda (a b c d e) (vector cn a b c d e)))
             ((6)  (lambda (a b c d e f) (vector cn a b c d e f)))
             ((7)  (lambda (a b c d e f g) (vector cn a b c d e f g)))
             (else #f) ) ) ) ) )
```

Les allocateurs de classes sans champ indexé et de moins de neuf champs normaux sont donc réalisés par des fermetures d'arité fixe. Si la classe n'a pas de champ indexé mais plus de neuf champs normaux, on revient au cas précédent qui vérifie si le nombre de valeurs d'initialisation est suffisant. Toujours pour minimiser la détection d'erreur, le fait qu'il n'y ait pas assez de valeurs d'initialisation est détecté par `cdr` ou `list-tail` qui doivent ne pas accepter de tronquer une liste déjà vide ou par la détection d'arité dans le cas des allocations de petits objets. Cette non-uniformité des messages d'erreur obtenus pour ce type d'anomalie est bien sûr préjudiciable. Le code de MEROONET augmenterait d'un bon quart si l'on prenait en compte ce souci de convivialité.

Une implantation native de MEROONET doit veiller à avoir une allocation efficace d'objets et, pour éviter cette monstruosité qui consiste à allouer pour allouer, se doterait probablement d'un mécanisme interne pour construire les allocateurs du type `(vector cn a b c ...)`.

11.5.4 Accès aux champs

Pour chaque champ d'une classe, MEROONET crée des fonctions d'accompagnement permettant de le lire, de l'écrire ou encore de connaître sa longueur s'il est indexé. Les fonctions d'accompagnement relatives aux champs, souvent nommées sélecteurs, sont construites par la sous-fonction de macro-expansion qui a pour nom `Field-generate-related-names`.

```
(define (Field-generate-related-names field class)
  (let* ((fname (careless-Field-name field))
         (cname (Class-name class))
         (cname-variable (symbol-concatenate cname '-class))
         (reader-name (symbol-concatenate cname '- fname))
         (writer-name (symbol-concatenate 'set- cname '- fname '!)) )
    `(begin
       (define ,reader-name
         (make-reader
          (retrieve-named-field ,cname-variable ',fname) ) )
       (define ,writer-name
         (make-writer
          (retrieve-named-field ,cname-variable ',fname) ) )
       ,@(if (Poly-Field? field)
             `((define ,(symbol-concatenate cname '- fname '-length)
                 (make-lengther
                  (retrieve-named-field ,cname-variable ',fname) ) ))
             '() ) ) ) )
```

Comme précédemment les fonctions d'accompagnement sont construites par fermeture et non par synthèse de leur code. Les constructeurs sont `make-reader`, `make-writer` et `make-lengther`. Ils prennent tous en arguments un champ et une classe. Comme les sélecteurs doivent être construits à l'évaluation, ils retrouveront ces objets par le nom de la variable globale qui contient la classe et par la fonction `retrieve-named-field`, qui permet de retrouver par son nom un champ dans une classe.

```
(define (retrieve-named-field class name)
  (let search ((fields (careless-Class-fields class)))
    (and (pair? fields)
         (if (eq? name (careless-Field-name (car fields)))
             (car fields)
             (search (cdr fields)) ) ) ) )
```

11.5.5 Lecteur de champ

Comme nous avons deux types de champs, indexé ou pas, nous avons aussi deux types de lecteurs d'arités différentes. Si l'on a la fibre quelque peu implantatoire, on remarquera que l'accès aux champs non précédés d'un champ indexé s'effectue à déplacement constant. La fonction `make-reader` de construction de lecteurs va tenir compte de ce fait. On parcourra la liste des champs afin de déterminer si le déplacement est constant ou pas. Si oui, une fonction appropriée est engendrée, sinon on recourrera à une fonction générale de lecture de champ : `field-value`. Ainsi tous les champs à déplacement constant (éventuellement indexé) sont-ils lus efficacement.

Les lecteurs sont des fonctions sûres en ce sens qu'elles vérifient que l'objet auquel ils s'appliquent est d'une classe convenable, c'est-à-dire d'une classe héritant de la

classe ayant introduit le champ en question : c'est le rôle de la fonction `check-class-membership` vue plus haut. Une autre raison de leur sûreté est que la lecture d'un champ indexé teste, grâce à `check-index-range`, que l'index est correct par rapport à la taille du champ indexé.

```
(define (make-reader field)
  (let ((class (Field-defining-class field)))
    (let skip ((fields (careless-Class-fields class))
               (offset *starting-offset*) )
      (if (eq? field (car fields))
          (cond ((Mono-Field? (car fields))
                 (lambda (o)
                   (check-class-membership o class)
                   (vector-ref o offset) ) )
                ((Poly-Field? (car fields))
                 (lambda (o i)
                   (check-class-membership o class)
                   (check-index-range i o offset)
                   (vector-ref o (+ offset 1 i)) ) ) )
          (cond ((Mono-Field? (car fields))
                 (skip (cdr fields) (+ 1 offset)) )
                ((Poly-Field? (car fields))
                 (cond ((Mono-Field? field)
                        (lambda (o)
                          (field-value o field) ) )
                       ((Poly-Field? field)
                        (lambda (o i)
                          (field-value o field i) ) ) ) ) ) ) ) ) )
(define (check-index-range i o offset)
  (let ((size (vector-ref o offset)))
    (if (not (and (>= i 0) (< i size)))
        (meroonet-error "Out of range index" i size) ) ) )
```

Pour les champs qui sont situés après un champ indexé, on s'est contenté d'un mécanisme qu'incarne la fonction `field-value` générale et non générique car MEROONET ne permet pas de créer de nouveaux types de champs. À la fonction `field-value` est bien évidemment associée la fonction `set-field-value!` ; les calculs de déplacements communs à ces deux fonctions sont concentrés en `compute-field-offset`.

```
(define (compute-field-offset o field)
  (let ((class (Field-defining-class field)))
    ;; (assume (check-class-membership o class))
    (let skip ((fields (careless-Class-fields class))
               (offset *starting-offset*) )
      (if (eq? field (car fields))
          offset
          (cond
           ((Mono-Field? (car fields))
            (skip (cdr fields) (+ 1 offset)) )
           ((Poly-Field? (car fields))
            (skip (cdr fields)
                  (+ 1 offset (vector-ref o offset)) ) ) ) ) ) ) )
(define (field-value o field . i)
  (let ((class (Field-defining-class field)))
    (check-class-membership o class)
    (let ((fields (careless-Class-fields class))
```

```
        (offset (compute-field-offset o field)) )
    (cond ((Mono-Field? field)
           (vector-ref o offset) )
          ((Poly-Field? field)
           (check-index-range (car i) o offset)
           (vector-ref o (+ offset 1 (car i))) ) ) ) ) )
```

11.5.6 Écrivain de champ

La définition des écrivains est exactement calquée sur celles des lecteurs et ne pose aucune difficulté. Par contre, la fonction `set-field-value!` a une signature curieuse. La signature des écrivains se déduit de la signature des lecteurs par ajout de la valeur à écrire en queue des arguments, ce qui est le mécanisme usuel en Scheme comme par exemple dans `car` et `set-car!`. Toutefois la présence de l'index optionnel vient perturber cette belle ordonnance, aussi avons-nous résolu de choisir l'ordre `(o v field . i)` [14].

```
(define (make-writer field)
  (let ((class (Field-defining-class field)))
    (let skip ((fields (careless-Class-fields class))
               (offset *starting-offset*) )
      (if (eq? field (car fields))
          (cond ((Mono-Field? (car fields))
                 (lambda (o v)
                   (check-class-membership o class)
                   (vector-set! o offset v) ) )
                ((Poly-Field? (car fields))
                 (lambda (o i v)
                   (check-class-membership o class)
                   (check-index-range i o offset)
                   (vector-set! o (+ offset 1 i) v) ) ) )
          (cond ((Mono-Field? (car fields))
                 (skip (cdr fields) (+ 1 offset)) )
                ((Poly-Field? (car fields))
                 (cond ((Mono-Field? field)
                        (lambda (o v)
                          (set-field-value! o v field) ) )
                       ((Poly-Field? field)
                        (lambda (o i v)
                          (set-field-value! o v field i) ) ) )
                 ) ) ) ) ) )
(define (set-field-value! o v field . i)
  (let ((class (Field-defining-class field)))
    (check-class-membership o class)
    (let ((fields (careless-Class-fields class))
          (offset (compute-field-offset o field)) )
      (cond ((Mono-Field? field)
             (vector-set! o offset v) )
            ((Poly-Field? field)
             (check-index-range (car i) o offset)
             (vector-set! o (+ offset 1 (car i)) v) ) ) ) ) )
```

D'un point de vue stylistique, les écrivains de champs sont construits avec le préfixe `set-` comme dans `set-car!`. On aurait pu aussi prendre le suffixe `-set!` comme

14. Cet ordre rappellera la signature de `putprop` aux nostalgiques des listes de propriétés.

dans `vector-set!`. Nous avons choisi le préfixe qui montre plus immédiatement qu'il s'agit d'une modification et qui ressemblera visuellement plus à une forme d'affectation.

11.5.7 Longueur de champ

On peut s'enquérir de la longueur des champs indexés soit par la fonction d'accompagnement spécialisée, soit par la fonction générale `field-length`. Là encore, on améliore l'accès aux champs non situés après un champ indexé.

```
(define (make-lengther field)
  ;; (assume (Poly-Field? field))
  (let ((class (Field-defining-class field)))
    (let skip ((fields (careless-Class-fields class))
               (offset *starting-offset*) )
      (if (eq? field (car fields))
          (lambda (o)
            (check-class-membership o class)
            (vector-ref o offset) )
          (cond ((Mono-Field? (car fields))
                 (skip (cdr fields) (+ 1 offset)) )
                ((Poly-Field? (car fields))
                 (lambda (o) (field-length o field)) ) ) ) ) ) ) )
(define (field-length o field)
  (let* ((class (Field-defining-class field))
         (fields (careless-Class-fields class))
         (offset (compute-field-offset o field)) )
    (check-class-membership o class)
    (vector-ref o offset) ) )
```

11.6 Création des classes

La forme `define-class` délègue le soin de créer un objet pour représenter la classe à définir à la fonction `register-class`. Celle-ci alloue l'objet puis appelle `Class-initialize!` pour analyser les paramètres de la définition de la classe et ainsi initialiser les différents champs de la classe. Lorsque ceux-ci sont tous remplis (avec l'aide de `parse-fields` pour les champs propres de la classe), la classe est insérée dans la hiérarchie des classes. Enfin l'appel à `update-generics` conférera à la classe nouvellement créée toutes les méthodes qu'elle hérite de sa super-classe.

```
(define (register-class name supername own-field-descriptions)
  (Class-initialize! (allocate-Class)
                     name
                     (->Class supername)
                     own-field-descriptions ) )
(define (Class-initialize! class name superclass
                           own-field-descriptions )
  (set-Class-number! class *class-number*)
  (set-Class-name! class name)
  (set-Class-superclass! class superclass)
  (set-Class-subclass-numbers! class '())
  (set-Class-fields!
   class (append (Class-fields superclass)
```

```
                (parse-fields class own-field-descriptions) ) )
;; install definitely the class
(set-Class-subclass-numbers!
 superclass
 (cons *class-number* (Class-subclass-numbers superclass)) )
(vector-set! *classes* *class-number* class)
(set! *class-number* (+ 1 *class-number*))
;; propagate the methods of the super to the fresh class
(update-generics class)
class )
```

Les spécifications des champs propres de la classe sont analysées par la fonction `parse-fields`. C'est elle qui s'occupe de l'analyse syntaxique (au demeurant fort simple) de ces spécifications. Lorsqu'une spécification est parenthésée, elle ne peut débuter que par un signe égal ou une étoile ; tout autre objet conduit à une erreur que l'on signalera par `meroonet-error`. Il n'est pas permis, en MEROONET, de redéfinir un champ hérité, c'est ce dont s'occupe la fonction `check-conflicting-name`. Il n'est par contre pas vérifié si un même nom apparaît plus d'une fois dans les champs propres.

```
(define (parse-fields class own-field-descriptions)
  (define (Field-initialize! field name)
    (check-conflicting-name class name)
    (set-Field-name! field name)
    (set-Field-defining-class-number! field (Class-number class))
    field )
  (define (parse-Mono-Field name)
    (Field-initialize! (allocate-Mono-Field) name) )
  (define (parse-Poly-Field name)
    (Field-initialize! (allocate-Poly-Field) name) )
  (if (pair? own-field-descriptions)
      (cons (cond
              ((symbol? (car own-field-descriptions))
               (parse-Mono-Field (car own-field-descriptions)) )
              ((pair? (car own-field-descriptions))
               (case (caar own-field-descriptions)
                 ((=) (parse-Mono-Field
                        (cadr (car own-field-descriptions)) ))
                 ((*) (parse-Poly-Field
                        (cadr (car own-field-descriptions)) ))
                 (else (meroonet-error
                         "Erroneous field specification"
                         (car own-field-descriptions) )) ) ) )
            (parse-fields class (cdr own-field-descriptions)) )
      '() ) )
(define (check-conflicting-name class fname)
  (let check ((fields (careless-Class-fields
                        (Class-superclass class) )))
    (if (pair? fields)
        (if (eq? (careless-Field-name (car fields)) fname)
            (meroonet-error "Duplicated field name" fname)
            (check (cdr fields)) )
        #t ) ) )
```

11.7 Fonctions d'accompagnement prédéfinies

L'ossature de base pour définir des classes a été entièrement présentée. Elle n'est cependant pas complètement opérationnelle : on ne peut encore définir une classe car les fonctions d'accompagnement des classes prédéfinies, comme `Class`, `Field`, etc. ne sont point présentes. Comme on ne peut utiliser `define-class` parce qu'elles manquent et que c'est justement le rôle de `define-class` que de les définir, nous nous retrouvons une fois de plus avec un problème d'auto-amorçage : il faut donc prendre son mal en patience et définir à la main ces fonctions. On peut se contenter d'écrire les seules qui soient nécessaires et on peut, en plus, simplifier leur code pour ne faire aucune vérification. Si tout marche bien, alors on expansera les définitions des classes prédéfinies et on coupera-collera ce code, comme nous l'avons fait. La vérité, toute entière et non fardée, est qu'il faut définir les prédicats avant les lecteurs, ceux-ci avant les écrivains, ces derniers avant les allocateurs et faire figurer les fonctions accompagnant `Class` avant toute autre. Donner ici tout le code serait par trop tirer à la ligne, aussi ne donnerons-nous que le synopsis de ces définitions :

```
(define Class? (make-predicate Class-class))
(define Generic? (make-predicate Generic-class))
(define Field? (make-predicate Field-class))
(define Class-name
  (make-reader (retrieve-named-field Class-class 'name)))
(define set-Class-name!
  (make-writer (retrieve-named-field Class-class 'name)))
(define Class-number
  (make-reader (retrieve-named-field Class-class 'number)))
(define set-Class-subclass-numbers!
  (make-writer (retrieve-named-field Class-class 'subclass-numbers)))
(define make-Class (make-maker Class-class))
(define allocate-Class (make-allocator Class-class))
(define Generic-name
  (make-reader (retrieve-named-field Generic-class 'name)))
(define allocate-Poly-Field (make-allocator Poly-Field-class))
```

À ce stade, il est maintenant possible d'user de `define-class`. Attention, n'en abusez point pour rédéfinir les classes prédéfinies `Object`, `Class` et *tutti quanti*, d'une part, parce que `define-class` n'est pas idempotente (vous y gagneriez douze classes initiales dont six inacessibles [15]), d'autre part, parce qu'en compilé, vous dupliqueriez les fonctions d'accompagnement.

11.8 Fonctions génériques

Les fonctions génériques sont le résultat de l'adaptation de la notion d'envoi de messages, chère au monde des objets, au monde fonctionnel Lispien. L'envoi de message à la Smalltalk [GR83] se présente comme :

```
receveur message: arguments...
```

15. Une astuce serait de ramener `*class-number*` à zéro et de redéfinir les classes pour qu'elles retrouvent les mêmes numéros.

Comme il a été souvent dit, tout est déjà en Lisp : il suffit d'ajouter des parenthèses ! Les premières importations d'envoi de message en Lisp utilisaient donc un mot-clé comme `send` (ou `=>` à la Planner [HS75]) et l'on obtenait :

```
(send receveur message arguments...)
```

Le receveur est l'objet qui reçoit le message ; il est unique en Smalltalk, c'est-à-dire que l'on n'envoie pas un message à plusieurs objets à la fois. Or dans les fols rêves que tout puisse être objet en Lisp, on s'aperçoit vite qu'une fonction comme l'addition binaire ne prend son sens que si elle connaît la classe de ses deux arguments : sur deux entiers, c'est l'addition binaire des entiers ; sur deux flottants, c'est celle des flottants et, sur les cas mixtes, il faut d'abord convertir l'argument entier en flottant avant d'additionner : c'est ce que l'on nomme la « contagion flottante ». L'addition dispose en fait de méthodes dont la sélection n'est pas faite sur un unique receveur. Ces méthodes sont nommées des *multi-méthodes*. La syntaxe de l'envoi de message, en plaçant le receveur en position privilégiée, nuit à la notion de multi-méthodes, aussi Common Loops proposa-t-il [BKK+86] de changer cette syntaxe en une autre plus suggestive :

```
(message receveur arguments...)
```

Le mot-clé `send` a disparu et le message apparaît en position fonctionnelle : c'est donc une fonction et puisqu'elle inspecte ses arguments afin d'en déduire la méthode à appliquer, c'est une fonction *générique*. La fonction générique pouvant prendre en compte tous les arguments qu'elle reçoit *(i)* peut considérer comme discriminant celui qu'elle souhaite ; *(ii)* ne pas accorder une quelconque prééminence au premier d'entre eux (l'ancien receveur) ; *(iii)* prendre en compte plus d'un argument discriminant comme dans le cas de l'addition binaire. L'envoi de messages a donc été généralisé et c'est ce concept de fonction générique que MEROONET procurera. Toutefois comme les multi-méthodes n'apparaissent pas dans ce livre et représentent en général moins de 5% des cas d'emploi [KR90], MEROONET n'implantera pas de multi-méthodes.

Certains seront peut-être intéressés par la notion de fonctions génériques et se demanderont si cette généralisation demeure fidèle à l'esprit objet. Nous ne tenterons point de répondre à cette question délicate. Toutefois l'on pourra noter que les fonctions génériques masquent complètement l'aspect objet puisque ce sont de vraies fonctions qui ne nécessitent pas un opérateur spécial comme `send`. Qu'une fonction soit générique et use d'envoi de message ou pas est donc laissé à l'implantation et n'a aucune conséquence sur son interface d'appel. Du côté de l'implantation, il y a un léger surcoût dû à cette encapsulation puisque l'application d'une fonction générique avec un premier argument discriminant (*g arguments*...) montre que *g* est, grossomodo, équivalente à `(lambda args (apply send (car args) g (cdr args)))`.

Comment représenter en Scheme des fonctions génériques ? Comme elles doivent être applicables, elles sont, en Scheme, nécessairement représentées par des fonctions. Mais dans un souci d'auto-description, on aimerait bien qu'elles soient aussi des objets de MEROONET appartenant à la classe `Generic`. On souhaiterait donc que ce soit à la fois des objets et des fonctions, bref des objets-fonctionnels (ou *funcallable objects*) ; on trouve de tels concepts en CLOS ou OakLisp [LP86, LP88]. De fait, les fonctions génériques sont vraiment des objets dotés d'un état interne correspondant à l'ensemble des méthodes qu'elles connaissent. Nous adopterons donc la technique largement suboptimale suivante : une fonction générique sera représentée à la fois par

un objet de MEROONET et par une fonction de Scheme, valeur de la variable portant le nom de la fonction générique ; cette fonction contiendra l'objet MEROONET dans sa fermeture. Les ajouts de méthodes s'effectueront sur l'objet qui sera donc vu de la fonction Scheme qui sera appliquée. Enfin, l'objet générique sera retrouvé par son nom (ce qui interdira les fonctions génériques anonymes) et non par la valeur de la variable globale portant son nom de façon à être insensible au contexte lexical où sont définies fonctions génériques et méthodes.

La forme de définition de fonctions génériques revêt la syntaxe suivante :

```
(define-generic (nom variables...) [defaut...] )
```

Le premier terme décrit la forme d'appel à la fonction générique et spécifie notamment la variable de discrimination, le receveur en Smalltalk. La variable discriminante apparaît entre parenthèses. Il est possible de définir le corps par défaut de la fonction générique dans le reste de la forme. Dans un langage où toutes les valeurs seraient des objets de MEROONET, ce serait équivalent à mettre une méthode sur la classe de toutes les valeurs : `Object`. Comme il existe dans cette implantation en Scheme des valeurs qui ne sont pas des objets, le corps par défaut sera appliqué sur toute valeur discriminante qui ne sera point un objet de MEROONET. Cette propriété rend plus simple l'intégration de MEROONET et Scheme et autorise des rattrapages d'erreur personnalisés. Une des originalités de MEROONET est que les fonctions génériques peuvent avoir n'importe quelle signature et notamment une variable pointée. Dans tous les cas, les méthodes devront posséder une signature compatible.

La classe `Generic` sera définie comme suit :

```
(define-class Generic Object (name default dispatch-table signature))
```

Le nom de la fonction générique permet de la retrouver par `->Generic`. Le champ `default` contient son corps par défaut donné par l'utilisateur ou synthétisé automatiquement (par défaut) pour signaler une erreur. La signature permettra de juger de la compatibilité des méthodes qui seront ajoutées à la fonction générique. Ce test est important d'une part en raison de l'insanité qu'il y aurait à ajouter des méthodes d'arité différentes : l'appel à une fonction générique est déjà suffisamment variable (en terme de méthode qui sera choisie) pour que l'on ne soit pas en plus confronté à des problèmes d'arité variable. D'autre part, une implantation native peut profiter de l'uniformité des méthodes pour tenter d'améliorer l'appel aux fonctions génériques [KR90].

L'état interne de la fonction générique est tout entier matérialisé par un vecteur, indexé par les numéros de classes, contenant toutes les méthodes connues de la fonction générique. Ce vecteur est connue comme la table d'aiguillage (ou *dispatch-table*) de la fonction générique. Le mécanisme d'appel d'une fonction générique est donc simple, le numéro de la classe de l'argument discriminant sert d'index dans la table d'aiguillage pour retrouver la méthode appropriée. Ce codage est extrêmement rapide mais consommateur de place, puisque l'ensemble de ces tables est équivalente à une matrice $n * m$ où n est le nombre total de classes et m le nombre total de fonctions génériques. Des techniques existent pour compacter cette table [VH94, Que93b], ce qui est largement possible car, la plupart du temps, les méthodes d'une fonction générique ne concernent qu'un sous-arbre de l'arbre des classes.

Les méthodes d'une fonction générique ainsi que son corps par défaut auront toutes une même arité finie et ce sera de même le cas des fonctions génériques ayant

une variable pointée. Par exemple, soit la fonction générique `f` ainsi définie :

```
(define-generic (f a (b) . c) (g b a c))
```

Le corps par défaut sera la fonction :

```
(lambda (a b c) (g b a c))
```

Et toutes les méthodes seront représentées par des fonctions d'arité semblable à `(a b c)`. L'image de la fonction générique en Scheme a, pour arité, l'arité spécifiée pour la fonction générique. Pour l'exemple précédent, ce sera :

```
(lambda (a b . c) ((determine-method G127 b) a b c))
```

La variable `G127` [16] a pour valeur l'objet générique contenant la table d'aiguillage au sein de laquelle la fonction `determine-method` choisit la méthode appropriée. Comme toutes les valeurs de Scheme ne sont pas des objets, il faut tout d'abord tester si la valeur de la variable discriminante est bien un objet de MEROONET.

```
(define (determine-method generic o)
  (if (Object? o)
      (vector-ref (Generic-dispatch-table generic)
                  (vector-ref o 0) )
      (Generic-default generic) ) )
```

Voici enfin la définition des fonctions génériques.

```
(define-meroonet-macro (define-generic call . body)
  (parse-variable-specifications
   (cdr call)
   (lambda (discriminant variables)
     (let ((generic (gensym)))          ; make generic hygienic
       `(define ,(car call)
          (let ((,generic
                 (register-generic
                  ',(car call)
                  (lambda ,(flat-variables variables)
                    ,(if (pair? body)
                         `(begin . ,body)
                         `(meroonet-error
                           "No method" ',(car call)
                           . ,(flat-variables variables) ) ) )
                  ',(cdr call) )))
            (lambda ,variables
              ((determine-method ,generic ,(car discriminant))
               . ,(flat-variables variables) ) ) ) ) ) ) ) )
```

La fonction `parse-variable-specifications` analyse la liste spécifiant les variables pour en extraire la variable discriminante et restructurer la liste des variables pour la rendre conforme à ce qu'attend Scheme ; elle sera commune à `define-generic` et `define-method`. Elle invoque son second argument sur ces deux résultats. Faisant toujours fi du dogmatisme, la fonction `parse-variable-specifications` ne vérifiera pas qu'il n'existe qu'une seule variable discriminante.

```
(define (parse-variable-specifications specifications k)
  (if (pair? specifications)
      (parse-variable-specifications
       (cdr specifications)
       (lambda (discriminant variables)
         (if (pair? (car specifications))
```

16. Pour rendre hygiénique l'expansion de `define-generic`, la variable enclosant l'objet générique porte un nom synthétisé par `gensym`.

```
            (k (car specifications)
               (cons (caar specifications) variables) )
            (k discriminant (cons (car specifications)
                                  variables )) ) ) )
      (k #f specifications) ) )
```

Le corps par défaut de la fonction générique est alors construit ; un objet générique est fabriqué par `register-generic` qui permet de ne pas exposer au niveau du résultat de l'expansion de `define-generic` les détails de codage des fonctions génériques et notamment la variable `*generics*`. Parmi ces détails, la table d'aiguillage est construite, on alloue donc un vecteur que l'on remplit initialement avec le corps par défaut qui est effectivement la méthode qui sera trouvée en l'absence de toute autre. La table d'aiguillage a pour taille le nombre total de classes possibles et non le nombre actuel de classes ; ce dernier cas nécessiterait lorsque de nouvelles classes sont définies d'augmenter la taille des tables d'aiguillage de toutes les fonctions génériques existantes.

```
(define (register-generic generic-name default signature)
  (let* ((dispatch-table (make-vector *maximal-number-of-classes*
                                      default ))
         (generic (make-Generic generic-name
                                default
                                dispatch-table
                                signature )) )
    (set! *generics* (cons generic *generics*))
    generic ) )
```

La fonction `flat-variables` aplatit une liste de variables et transforme l'éventuelle variable pointée finale en une variable normale. Cette transformation est effectuée sur le corps par défaut mais le sera aussi sur toutes les méthodes à venir.

```
(define (flat-variables variables)
  (if (pair? variables)
      (cons (car variables) (flat-variables (cdr variables)))
      (if (null? variables) variables (list variables)) ) )
```

Retour sur la définition de classes

L'on avait mentionné la fonction `update-generics` lorsqu'une classe était définie. Son rôle était de propager les méthodes de la super-classe sur cette nouvelle classe. Il paraît en effet normal que si l'on définit la classe `Point` et la méthode `show` permettant d'afficher des points, alors si l'on définit la classe `ColoredPoint` ceux-ci héritent *ipso facto* de cette méthode. C'est pour implanter cette caractéristique que nous avons besoin de disposer de toutes les fonctions génériques pour les mettre à jour.

```
(define (update-generics class)
  (let ((superclass (Class-superclass class)))
    (for-each (lambda (generic)
                (vector-set! (Generic-dispatch-table generic)
                             (Class-number class)
                             (vector-ref
                              (Generic-dispatch-table generic)
                              (Class-number superclass) ) ) )
              *generics* ) ) )
```

11.9 Méthodes

La définition de méthodes passe par `define-method` dont la syntaxe rappelle celle de `define`. La liste des variables est semblable à celle qui apparaît dans `define-generic` : la variable discriminante figure entre parenthèses ainsi que le nom de la classe sur laquelle la méthode est installée.

`(define-method (`nom variables...`)` formes... `)`

Les fonctions génériques que nous avons précédemment définies sont des objets auxquels on peut conférer de nouveaux comportements ou méthodes par le biais de `define-method`. Les fonctions génériques de MEROONET sont donc des objets modifiables et, de ce fait, se prêtent mal aux optimisations, sauf à geler la hiérarchie des classes (les sceller, `seal` en Dylan [App92b]) et à faire un peu d'analyse de types ou plutôt de classes. Que les fonctions génériques soient immuables serait une autre solution mais qui nécessiterait de disposer d'objet-fonctionnels, ce qui l'exclut ici.

Nous avons déjà évoqué l'arité des méthodes pour le corps par défaut lors de la description de l'image fonctionnelle en Scheme des fonctions génériques. La méthode, une fois construite, doit être insérée dans la table d'aiguillage non seulement sur la classe sur laquelle elle doit être définie mais aussi pour toutes ses sous-classes qui ne la redéfinissent point.

Le problème qui reste est l'implantation de ce que Smalltalk nomme `super` et CLOS `call-next-method`, c'est-à-dire la possibilité pour une méthode de pouvoir invoquer la méthode qui aurait été normalement invoquée si elle n'avait pas été là. La forme `(call-next-method)` ne peut apparaître que dans une méthode et correspond à l'invocation de la super-méthode avec, implicitement, les mêmes [17] arguments que la méthode. Bien sûr, la super-méthode de la classe `Object` est le corps par défaut de la fonction générique.

La recherche de la super-méthode qu'effectue la fonction `call-next-method` locale au corps de la méthode s'inspire de `determine-method`, sauf que l'on a pas besoin de tester que la valeur de la variable discriminante est bien un objet et que le numéro de la classe à considérer pour l'indexation est non plus celui de la classe directe de la valeur discriminante mais celui de sa super-classe. Il est donc nécessaire pour installer une méthode utilisant `(call-next-method)` que le numéro de la super-classe de la classe sur laquelle on l'installe soit connue. Attention, toujours pour des raisons de macroexpansion, ce numéro ne sera connu qu'à l'évaluation et pas tout de suite. Pour cela, nous recourrons donc à la technique suivante. La forme `define-method` fabriquera une pré-méthode qui est la méthode paramétrée par la fonction générique et la classe sur lesquelles elle sera installée. Afin de masquer les détails de cette installation et réduire la taille de la macroexpansion de `define-method`, la fonction `register-method` est créée.

```
(define-meroonet-macro (define-method call . body)
  (parse-variable-specifications
   (cdr call)
   (lambda (discriminant variables)
```

17. Nous n'avons pas retenu la possibilité de changer ces arguments comme en CLOS car il serait alors possible de changer le contenu de la variable de discrimination en n'importe quoi et cela violerait l'intention (et probablement les hypothèses de compilation) de la super-méthode.

```
(let ((g (gensym))(c (gensym)))    ; make g and c hygienic
  `(register-method
    ',(car call)
    (lambda (,g ,c)
      (lambda ,(flat-variables variables)
        (define (call-next-method)
          ((if (Class-superclass ,c)
               (vector-ref (Generic-dispatch-table ,g)
                           (Class-number
                            (Class-superclass ,c) ) )
               (Generic-default ,g) )
           . ,(flat-variables variables) ) )
        . ,body ) )
    ',(cadr discriminant)
    ',(cdr call) ) ) ) )
```

La fonction `register-method` détermine la classe et la fonction générique impliquées, convertit la pr-eméthode en une méthode, vérifie que les signatures sont compatibles et l'installe dans la table d'aiguillage. Tester la compatibilité de signatures à la macroexpansion aurait requis que `define-method` connaisse les signatures des fonctions génériques ; comme la vérification est simple et ne se fait qu'une seule fois à l'installation sans entraver l'efficacité de l'appel aux fonctions génériques, nous l'avons rejetée à l'évaluation. On notera pour la propagation des méthodes une comparaison de fonctions par `eq?`.

```
(define (register-method generic-name pre-method class-name signature)
  (let* ((generic (->Generic generic-name))
         (class (->Class class-name))
         (new-method (pre-method generic class))
         (dispatch-table (Generic-dispatch-table generic))
         (old-method (vector-ref dispatch-table
                                 (Class-number class) )) )
    (check-signature-compatibility generic signature)
    (let propagate ((cn (Class-number class)))
      (let ((content (vector-ref dispatch-table cn)))
        (if (eq? content old-method)
            (begin
              (vector-set! dispatch-table cn new-method)
              (for-each
               propagate
               (Class-subclass-numbers
                (number->class cn) ) ) ) ) ) ) ) )
(define (check-signature-compatibility generic signature)
  (define (coherent-signatures? la lb)
    (if (pair? la)
        (if (pair? lb)
            (and (or ;; similar discriminating variable
                     (and (pair? (car la)) (pair? (car lb)))
                     ;; similar regular variable
                     (and (symbol? (car la)) (symbol? (car lb))) )
                 (coherent-signatures? (cdr la) (cdr lb)) )
            #f )
        (or (and (null? la) (null? lb))
            ;; similar dotted variable
            (and (symbol? la) (symbol? lb)) ) ) )
  (if (not (coherent-signatures? (Generic-signature generic)
                                 signature ))
```

```
(meroonet-error "Incompatible signatures" generic signature) ) )
```

11.10 Conclusions

De nombreux points sont améliorables dans les programmes ci-dessus pour accroître la vitesse de MEROONET en Scheme ou ses qualités réflexives. D'autres améliorations sont aussi envisageables dans une implantation native dans laquelle toute valeur serait un objet de MEROONET, c'est pour cette raison que EULISP, Ilog Talk et CLtL2 intègrent, dans leurs fondements mêmes, les concepts d'objets.

11.11 Exercices

Exercice 11.1 : Concevez une version plus juste (moins approximative) de `Object?`.

Exercice 11.2 : Concevez une fonction générique, `clone`, capable de dupliquer (superficiellement) un objet MEROONET.

Exercice 11.3 : On peut créer de nouveaux types de classes, sous-classes de `Class`, ce que l'on nomme des méta-classes. Écrivez une méta-classe dont les instances sont des classes comptant le nombre d'objets qu'elles ont pu créer.

Exercice 11.4 : Afin d'augmenter les qualités réflexives de MEROONET, les classes et les champs pourraient disposer de champs supplémentaires afin de référencer les fonctions d'accompagnement (prédicat et allocateurs pour les classes, lecteur, écrivain et mesureur pour les champs) qui leur sont associées. Augmentez MEROONET pour ce faire.

Exercice 11.5 : CLOS ne requiert pas qu'une fonction générique existe pour qu'on lui adjoigne une nouvelle méthode. Modifiez `define-method` pour créer à la volée la fonction générique, si d'aventure elle n'existe pas déjà.

Exercice 11.6 : Dans certains systèmes d'objets comme CLOS ou EULISP, il est possible, conjointement à `call-next-method`, de savoir s'il y a une méthode qui suit grâce à la fonction `next-method?`. C'est une capacité réflexive qui permet de dévider toutes les super-méthodes sans erreur. La « méthode » qui suit celle sur `Object` correspond au corps par défaut ; `next-method?` répond Faux lorsqu'il n'y a plus que celle-là. Modifiez `define-method` pour procurer `next-method?`.

11.12 Lectures recommandées

Une autre approche des objets en Scheme s'inspire de ceux de T [AR88]. Les amateurs de méta-objets pourront lire l'article historique définissant ObjVlisp [Coi87] ou l'auto-description de CLOS [KdRB92].

Solutions des exercices

Seuls les problèmes appelés exercices sont corrigés ; les projets ne le sont pas.

Exercice 1.1 : Il suffit d'insérer des ordres de trace aux bons endroits :

```
(define (tracing.evaluate exp env)
  (if ... ...
      (case (car exp) ...
        (else (let ((fn        (evaluate (car e) env))
                    (arguments (evlis (cdr e) env)) )x
                (display `(calling ,(car e) with . ,arguments)
                         *trace-port* )
                (let ((result (invoke fn arguments)))
                  (display `(returning from ,(car e) with ,result)
                           *trace-port* )
                  result ) )) ) ) )
```

Deux points sont à noter, le nom de la fonction est imprimée plutôt que sa valeur, c'est souvent plus informatif. D'autre part les impressions sont émises sur un flux particulier qu'il sera ainsi plus simple de rediriger vers une fenêtre ou un fichier ou même de mélanger avec le flux normal.

Exercice 1.2 : Cette optimisation est attribuée à Friedman et Wise selon [Wan80b]. Elle permet d'oublier qu'il y a encore un `(evlis '() env)` à effectuer lorsque l'on évalue le dernier terme d'une liste. En effet, comme le résultat est indépendant de `env`, il est inutile de mémoriser cet appel ainsi que la valeur de `env`. Comme les listes d'arguments ont de l'ordre de trois à quatre termes et que cette optimisation peut toujours être effectuée en fin de liste, elle est intéressante. On remarquera la fonction locale qui permet d'économiser un test.

```
(define (evlis exps env)
  (define (evlis exps)
    ;; (assume (pair? exps))
    (if (pair? (cdr exps))
        (cons (evaluate (car exps) env)
              (evlis (cdr exps)) )
        (list (evaluate (car exps) env)) ) )
  (if (pair? exps)
      (evlis exps)
      '() ) )
```

Exercice 1.3 : Cette représentation est connue sous le nom de « cage thoracique »

(*rib cage*) car elle en rappelle la structure. Elle diminue le coût de l'appel fonctionnel en nombre de paires consommées mais augmente le coût de la recherche ou de la modification de la valeur des variables. De plus, la vérification d'arité n'est alors plus aussi sûre car on ne détecte plus les arguments superflus (il faudrait pour cela modifier `extend`).

```
(define (lookup id env)
  (if (pair? env)
      (let look ((names (caar env))
                 (values (cdar env)) )
        (cond ((symbol? names)
               (if (eq? names id) values
                   (lookup id (cdr env)) ) )
              ((null? names) (lookup id (cdr env)))
              ((eq? (car names) id)
               (if (pair? values)
                   (car values)
                   (wrong "Too less values") ) )
              (else (if (pair? values)
                        (look (cdr names) (cdr values))
                        (wrong "Too less values") )) ) )
      (wrong "No such binding" id) ) )
```

La fonction `update!` se déduit simplement de `lookup`.

Exercice 1.4 :

```
(define (s.make-function variables body env)
  (lambda (values current.env)
    (for-each (lambda (var val)
                (putprop var
                         'apval
                         (cons val (getprop var 'apval)) ) )
              variables values )
    (let ((result (eprogn body current.env)))
      (for-each (lambda (var)
                  (putprop var 'apval (cdr (getprop var 'apval))) )
                variables )
      result ) ) )
(define (s.lookup id env)
  (car (getprop id 'apval)) )
(define (s.update! id env value)
  (set-car! (getprop id 'apval) value) )
```

Exercice 1.5 : Comme il y a de nombreux autres prédicats dans le même cas, autant définir une macro pour les définir. Notez que la valeur de `the-false-value` est Vrai pour le Lisp de définition.

```
(define-syntax defpredicate
  (syntax-rules ()
    ((defpredicate name value arity)
     (defprimitive name
       (lambda values (or (apply value values) the-false-value))
       arity ) ) ) )
(defpredicate < < 2)
```

Exercice 1.6 : La difficulté de l'entreprise est l'arité très variable de `list`. On la définira donc directement par `definitial` :

```
(definitial list
  (lambda (values) values) )
```

Exercice 1.7 : La fonction `call/cc` se définit bien évidemment grâce à `call/cc` elle-même. L'astuce est de bien convertir les fonctions du Lisp sous-jacent en les fonctions du Lisp en cours de définition.

```
(defprimitive call/cc
  (lambda (f)
    (call/cc (lambda (g)
               (invoke
                f (list (lambda (values)
                          (if (= (length values) 1)
                              (g (car values))
                              (wrong "Incorrect arity" g) ) ))) )) )
  1 )
```

Exercice 1.8 : C'est le même problème que pour `call/cc`, il faut faire la part entre les abstractions du Lisp sous-jacent et du Lisp en cours de définition. La fonction `apply` est par ailleurs d'arité variable et elle doit transformer la liste de ses arguments et notamment le dernier d'entre eux pour en faire une vraie liste de valeurs.

```
(definitial apply
  (lambda (values)
    (if (>= (length values) 2)
        (let ((f (car values))
              (args (let flat ((args (cdr values)))
                      (if (null? (cdr args))
                          (car args)
                          (cons (car args) (flat (cdr args))) ) )) )
          (invoke f args) )
        (wrong "Incorrect arity" 'apply) ) ) )
```

Exercice 1.9 : Il suffit de mémoriser la continuation d'appel à l'interprète et de lier cet échappement, convenablement adapté au protocole d'appel fonctionnel, à la variable `end`.

```
(define (chapter1d-scheme)
  (define (toplevel)
    (display (evaluate (read) env.global))
    (toplevel) )
  (display "Welcome to Scheme")(newline)
  (call/cc (lambda (end)
             (defprimitive end end 1)
             (toplevel) )) )
```

Exercice 1.10 : Ces rapports dépendent à la fois de l'implantation que vous utilisez et des programmes sur lesquels vous effectuez vos mesures. Ces rapports sont classiquement de l'ordre de 5 à 15 [ITW86].
Cet exercice est cependant intéressant pour faire prendre conscience que la définition d'`evaluate` est écrite dans un Lisp fondamental et peut être évaluée à la fois par Scheme ou par le langage qu'`evaluate` définit.

Exercice 1.11 : Comme l'on peut toujours ramener les séquences à des séquences binaires, il suffit de montrer comment réécrire celles-là. L'idée est d'encapsuler dans des suspensions (ou *thunks*) les expressions qui risqueraient d'induire des accrochages.

```
(begin expression1 expression2)
≡  ((lambda (void other) (other))
          expression1
          (lambda () expression2) )
```

Exercice 2.1 : Elle se traduit simplement en `(cons 1 2)`. On peut aussi définir par compatibilité :

```
(define (funcall f . args) (apply f args))
(define (function f) f)
```

ou encore, sous forme de macros :

```
(define-syntax funcall
  (syntax-rules ()
     ((funcall f arg ...) (f arg ...)) ) )
(define-syntax function
  (syntax-rules ()
     ((function f) f) ) )
```

Exercice 2.2 : Le problème là est de savoir *(i)* s'il est légal de parler de la fonction `bar` alors qu'elle n'est pas encore définie ; *(ii)* si, dans le cas où `bar` est définie avant que le résultat de `(function bar)` ne soit appliqué, l'on va obtenir une erreur ou bien la nouvelle fonction qui vient d'être définie. La différence est dans la forme spéciale `function` : parle-t-on de *la* fonction `bar` ou de la valeur instantanément associée dans l'espace des fonctions ?

Exercice 2.3 : Il suffit de modifier `invoke` comme suit pour reconnaître nombres et listes en position fonctionnelle :

```
(define (invoke fn args)
  (cond ((procedure? fn) (fn args))
        ((number? fn)
         (if (= (length args) 1)
             (if (>= fn 0) (list-ref (car args) fn)
                 (list-tail (car args) (- fn)) )
             (wrong "Incorrect arity" fn) ) )
        ((pair? fn)
         (map (lambda (f) (invoke f args))
              fn ) )
        (else (wrong "Cannot apply" fn)) ) )
```

Exercice 2.4 : La difficulté est que la fonction passée appartient au Lisp défini et non au Lisp de définition.

```
(definitial new-assoc/de
  (lambda (values current.denv)
    (if (= 3 (length values))
        (let ((tag        (car values))
              (default    (cadr values))
              (comparator (caddr values)) )
          (let look ((denv current.denv))
            (if (pair? denv)
```

```
            (if (eq? the-false-value
                     (invoke comparator (list tag (caar denv))
                                        current.denv ) )
                (look (cdr denv))
                (cdar denv) )
            (invoke default (list tag) current.denv) ) ) )
        (wrong "Incorrect arity" 'assoc/de) ) ) )
```

Exercice 2.5 : Là encore on devra adapter la fonction `specific-error` au système d'exception sous-jacent.

```
(define-syntax dynamic-let
  (syntax-rules ()
    ((dynamic-let () . body)
     (begin . body) )
    ((dynamic-let ((variable value) others ...) . body)
     (bind/de 'variable (list value)
              (lambda () (dynamic-let (others ...) . body)) ) ) ) )
(define-syntax dynamic
  (syntax-rules ()
    ((dynamic variable)
     (car (assoc/de 'variable specific-error)) ) ) )
(define-syntax dynamic-set!
  (syntax-rules ()
    ((dynamic-set! variable value)
     (set-car! (assoc/de 'variable specific-error) value) ) ) )
```

Exercice 2.6 : Une variable privée `properties` commune aux deux fonctions contient les listes de propriétés de tous les symboles.

```
(let ((properties '()))
  (set! putprop
        (lambda (symbol key value)
          (let ((plist (assq symbol properties)))
            (if (pair? plist)
                (let ((couple (assq key (cdr plist))))
                  (if (pair? couple)
                      (set-cdr! couple value)
                      (set-cdr! plist (cons (cons key value)
                                            (cdr plist) )) ) )
                (let ((plist (list symbol (cons key value))))
                  (set! properties (cons plist properties)) ) ) )
          value ) )
  (set! getprop
        (lambda (symbol key)
          (let ((plist (assq symbol properties)))
            (if (pair? plist)
                (let ((couple (assq key (cdr plist))))
                  (if (pair? couple)
                      (cdr couple)
                      #f ) )
                #f ) ) ) ) )
```

Exercice 2.7 : On ajoutera la clause suivante à l'interprète `evaluate` :

```
...
((label)  ;Syntax: {(label { name} (lambda ({ variables}) { body}))}
```

```
(let* ((name    (cadr e))
       (new-env (extend env (list name) (list 'void)))
       (def     (caddr e))
       (fun     (make-function (cadr def) (cddr def) new-env)) )
  (update! name new-env fun)
  fun ) )
```

Exercice 2.8 : On ajoutera la clause suivante à `f.evaluate`. Notez la ressemblance avec `flet` sauf en ce qui concerne l'environnement fonctionnel de création des fonctions locales.

```
...
((labels)
 (let ((new-fenv (extend fenv
                         (map car (cadr e))
                         (map (lambda (def) 'void) (cadr e)) )))
   (for-each (lambda (def)
               (update! (car def)
                        new-fenv
                        (f.make-function (cadr def) (cddr def)
                                         env new-fenv ) ) )
             (cadr e) )
   (f.eprogn (cddr e) env new-fenv ) ) )
```

Exercice 2.9 : Comme une forme `let` préserve l'indétermination, nous devons faire en sorte de ne pas séquencer le calcul des formes d'initialisation et, donc, de les regrouper en un `let`, qui doit toutefois être dans le bon environnement. On pourra écrire en usant d'une expansion hygiénique ou à grands coups de `gensym` pour les variables temporaires $temp_i$:

```
(let ((variable_1 'void)
      ...
      (variable_n 'void) )
  (let ((temp_1 expression_1)
        ...
        (temp_n expression_n) )
    (set! variable_1 temp_1)
    ...
    (set! variable_n temp_n)
    corps ) )
```

Exercice 2.10 : Voici la version binaire. La η-conversion a été modifiée pour tenir compte d'une fonction binaire.

```
(define fix2
  (let ((d (lambda (w)
             (lambda (f)
               (f (lambda (x y) (((w w) f) x y))) ) )))
    (d d) ) )
```

Enfin, voici une version n-aire.

```
(define fixN
  (let ((d (lambda (w)
             (lambda (f)
               (f (lambda args (apply ((w w) f) args))) ) )))
    (d d) ) )
```

Exercice 2.11 : Un peu d'intuition (!) suffit pour voir que la définition précédente `fixN` peut s'étendre en :

```
(define NfixN
  (let ((d (lambda (w)
             (lambda (f*)
               (list ((car f*)
                      (lambda a (apply (car ((w w) f*)) a))
                      (lambda a (apply (cadr ((w w) f*)) a)) )
                     ((cadr f*)
                      (lambda a (apply (car ((w w) f*)) a))
                      (lambda a (apply (cadr ((w w) f*)) a)) ) ) ) )))
    (d d) ) )
```

Ce que l'on peut donc généraliser, tout en prenant garde de prévenir l'évaluation trop hâtive de `((w w) f)`, en :

```
(define NfixN2
  (let ((d (lambda (w)
             (lambda (f*)
               (map (lambda (f)
                      (apply f (map (lambda (i)
                                      (lambda a (apply
                                                  (list-ref ((w w) f*)
                                                            i )
                                                  a )) )
                                    (iota 0 (length f*)) )) )
                    f* ) ) )))
    (d d) ) )
```

Attention : notez que l'ordre des fonctions dont on prend le point fixe doit toujours être le même : la définition de `odd?` doit être la première de la liste et la fonctionnelle qui la définit doit comporter `odd?` en première variable.
La fonction `iota` est ainsi définie en souvenir d'APL.

```
(define (iota start end)
  (if (< start end)
      (cons start (iota (+ 1 start) end))
      '() ) )
```

Les fonctions précédentes ne sont pas très efficaces, on consultera [Gol02] pour de bien meilleures versions.

Exercice 2.12 : Cette fonction est attribuée à Klop [Bar84] et l'on pourra vérifier que `((klop meta-fact) 5)` calcule bien 120.
Comme toutes les variables internes `s`, `c`, `h`, `e` et `m` sont liées à `r` ; leur ordre importe peu dans l'application `(m e c h e s)` : il suffit que l'arité soit respectée. On peut en outre ajouter ou retrancher des variables. Si l'on réduit `s`, `c`, `h`, `e` et `m` à n'être plus que `w` alor on retrouve **Y**.

Exercice 2.13 : Le résultat est : 120. Admirez au passage l'auto-application qui permet de réaliser la récursion. Une autre possibilité est de modifier le codage de la factorielle pour se prendre elle-même en argument, ce qui conduit à :

```
(define (factfact n)
  (define (internal-fact f n)
    (if (= n 1) 1
        (* n (f f (- n 1))) ) )
  (internal-fact internal-fact n) )
```

Exercice 3.1 : Cette forme pourrait être nommée `(the-current-continuation)` car elle retourne sa propre continuation. On voit là l'intérêt de `call/cc` par rapport à `the-continuation` puisque cette dernière ne ramène que sa continuation, ce qui est sans intérêt.

Détaillons le calcul en indexant différemment les continuations et fonctions en jeu et en abrégeant l'abréviation `call/cc` en `cc`. Les continuations préfixent en indice les expressions. Soit donc $_{k_0}$`(call/cc`$_1$ `call/cc`$_2$`)` à calculer ; $_{k_0}$ est la continuation avec laquelle calculer `(cc`$_1$ `cc`$_2$`)`. Rappelons que la définition de `call/cc` est :

$_k$`(call/cc` ϕ`)` $\rightarrow$ $_k$`(`ϕ k`)`

On a donc que $_{k_0}$`(cc`$_1$ `cc`$_2$`)` se réécrit en $_{k_0}$`(cc`$_2$ `k`$_0$`)`, qui lui-même se réécrit en $_{k_0}$`(k`$_0$ `k`$_0$`)` qui retourne donc la valeur `k`$_0$.

Exercice 3.2 : Détaillons cette expression que l'on indexera comme suit et selon les conventions de l'exercice précédent : $_{k_0}$`((cc`$_1$ `cc`$_2$`) (cc`$_3$ `cc`$_4$`))`. Supposons pour simplifier que les termes d'une application fonctionnelle soient évalués de la gauche vers la droite. Alors l'expression originale devient : $_{k_0}$`(`$_{k_1}$`(cc`$_1$ `cc`$_2$`) (cc`$_3$ `cc`$_4$`))` où k_1 est $\lambda\phi.$$_{k_0}$`(`$\phi$ $_{k_2}$`(cc`$_3$ `cc`$_4$`))` et k_2 est $\lambda\epsilon.$$_{k_0}$`(`$k_1$ ϵ`)`. La forme initiale se réécrit en $_{k_0}(k_1\ k_2)$, c'est-à-dire $_{k_0}(k_2\ {}_{k_2'}$`(cc`$_3$ `cc`$_4$`))` où k_2' est $\lambda\epsilon.$$_{k_0}(k_2\ \epsilon)$, ce qui conduit à $_{k_0}(k_1\ k_2')$ puis à $_{k_0}(k_1'\ k_2'')$ ... On voit alors que le calcul boucle. Vous pouvez vérifier que ce fait ne dépend pas de l'ordre d'évaluation des termes de l'application fonctionnelle.

C'est probablement le plus court programme (en nombre de symboles) qui boucle indéfiniment !

Exercice 3.3 : Chaque calcul préfixé par un *label* sera transformé en une fonction locale à une gigantesque forme `labels`. Les `go` sont traduits en des appels à ces fonctions mais sont associés à un échappement dynamique assurant que les formes `go` ont la bonne continuation. L'expansion ressemblera à :

```
(block EXIT
  (let (LABEL (TAG (list 'tagbody)))
    (labels ((INIT () expressions0... (label1))
             (label1 () expressions1... (label2))
             ...
             (labeln () expressionsn... (return-from EXIT nil)) )
      (setq LABEL (function INIT))
      (while #t
        (setq LABEL (catch TAG (funcall LABEL))) ) ) ) )
```

Les formes `(go` *label*`)` sont traduites en `(throw TAG` *label*`)` tandis que `(return` *value*`)` deviendra `(return-from EXIT` *value*`)`. Dans tout ce qui précède les noms en majuscules sont censés ne créer aucun conflit.

La traduction un peu compliquée de `go` assure la bonne continuation du débranchement et évite que, dans `(bar (go L))`, `bar` ne soit appelée lorsque `(go L)` retourne une valeur, ce qui pourrait arriver si l'on écrivait par exemple :

```
(tagbody   A (return (+ 1 (catch 'foo (go B))))
           B (* 2 (throw 'foo 5)) )
```

On consultera aussi [Bak92b].

Exercice 3.4 : On introduira une nouvelle sous-classe de fonctions :

```
(define-class function-with-arity function (arity))
```

On redéfinira l'évaluation des formes `lambda` pour maintenant créer une telle instance :

```
(define (evaluate-lambda n* e* r k)
  (resume k (make-function-with-arity n* e* r (length n*))) )
```

Enfin on adaptera le protocole d'invocation de ces nouvelles fonctions :

```
(define-method (invoke (f function-with-arity) v* r k)
  (if (= (function-with-arity-arity f) (length v*))
      (let ((env (extend-env (function-env f)
                             (function-variables f)
                             v* )))
        (evaluate-begin (function-body f) env k) )
      (wrong "Incorrect arity" (function-variables f) v*) ) )
```

Exercice 3.5 :

```
(definitial apply
  (make-primitive
   'apply
   (lambda (v* r k)
     (if (>= (length v*) 2)
         (let ((f (car v*))
               (args (let flat ((args (cdr v*)))
                       (if (null? (cdr args))
                           (car args)
                           (cons (car args) (flat (cdr args))) ) )) )
           (invoke f args r k) )
         (wrong "Incorrect arity" 'apply) ) ) ) )
```

Exercice 3.6 : On s'inspirera de l'exercice précédent et l'on définira un nouveau type de fonctions.

```
(define-class function-nadic function (arity))
(define (evaluate-lambda n* e* r k)
  (resume k (make-function-with-arity n* e* r (length n*))) )
(define-method (invoke (f function-nadic) v* r k)
  (define (extend-env env names values)
    (if (pair? names)
        (make-variable-env
         (extend-env env (cdr names) (cdr values))
         (car names)
         (car values) )
        (make-variable-env env names values) ) )
  (if (>= (length v*) (function-nadic-arity f))
      (let ((env (extend-env (function-env f)
                             (function-variables f)
                             v* )))
        (evaluate-begin (function-body f) env k) )
      (wrong "Incorrect arity" (function-variables f) v*) ) )
```

Exercice 3.7 : La boucle d'interaction doit donc maintenant être dans la continuation initiale. On écrira par exemple :

```
(define (chap3j-interpreter)
  (letrec ((k.init (make-bottom-cont
```

```
                    'void (lambda (v) (display v)
                                      (toplevel) ) ))
          (toplevel (lambda () (evaluate (read) r.init k.init))) )
    (toplevel) ) )
```

Exercice 3.8 : On définira la classe des continuations réifiées qui ne fait qu'encapsuler une continuation interne (instance de `continuation`), la fonction `call/cc` pour fabriquer un tel objet et la méthode d'invocation adaptée.

```
(define-class reified-continuation value (k))
(definitial call/cc
  (make-primitive 'call/cc
                  (lambda (v* r k)
                    (if (= 1 (length v*))
                        (invoke (car v*)
                                (list (make-reified-continuation k))
                                r
                                k )
                        (wrong "Incorrect arity" 'call/cc v*) ) ) ) )
(define-method (invoke (f reified-continuation) v* r k)
  (if (= 1 (length v*))
      (resume (reified-continuation-k f) (car v*))
      (wrong "Continuations expect one argument" v* r k) ) )
```

Exercice 3.9 : Le propre de cette fonctionnelle est de ne jamais retourner.

```
(defun eternel-retour (thunk)                                  COMMON LISP
  (labels ((loop ()
             (unwind-protect (thunk)
                (loop) ) ))
    (loop) ) )
```

Exercice 3.10 : Les expressions valent 33 et 44. La fonction `make-box` simule une boîte sans aucune affectation ou fonction à effet de bord apparent. L'effet est obtenu par la conjonction de `call/cc` et `letrec` et s'explique bien si l'on pense à l'expansion de `letrec` en termes de `let` et `set!`. Il donne toutefois du fil à retordre aux partisans de la forme spéciale `letrec` en présence de continuations de première classe plusieurs fois invocables.

Exercice 3.11 : La fonction `evaluate` se réécrira simplement en une fonction générique :

```
(define-generic (evaluate (e) r k)
   (wrong "Not a program" e r k) )
```

On reprendra les fonctions déjà existantes pour orner `evaluate` :

```
(define-method (evaluate (e quotation) r k)
  (evaluate-quote (quotation-value e) r k) )
(define-method (evaluate (e assignment) r k)
  (evaluate-set! (assignment-name e)
                 (assignment-form e)
                 r k ) )
...
```

Il faut encore définir les classes correspondant aux différentes formes syntaxiques possibles :

```
(define-class program Object ())
(define-class quotation program (value))
(define-class assignment program (name form))
...
```

Il ne reste plus qu'à définir un lecteur approprié qui sache lire un programme et construire une instance de `program`. C'est l'objet de la fonction `objectify` au paragraphe 9.11.1.

Exercice 3.12 : On définira `throw` comme la fonction :

```
(definitial throw
   (make-primitive
    'throw
    (lambda (v* r k)
      (if (= 2 (length v*))
          (catch-lookup k (car v*)
                        (make-throw-cont k `(quote ,(cadr v*)) r) )
          (wrong "Incorrect arity" 'throw v*) ) ) ) )
```

Pour ne pas redéfinir une variante simple de `catch-lookup`, une fausse instance de `throw-cont` a été fabriquée qui leurre l'interprète. La seule valeur qui y est importante est la valeur à transmettre.

Exercice 3.13 : Le code traduit en CPS est plus lent car il créé de nombreuses fermetures pour simuler la continuation. Par ailleurs, la transformation CPS n'est pas idempotente, c'est-à-dire que transformer en CPS le résultat d'une transformation en CPS n'est pas l'identité. Cela est évident lorsque l'on considère la factorielle en CPS : la continuation `k` peut être n'importe quelle fonction et peut aussi avoir des effets de bord sur le contrôle. Par exemple, on pourra écrire :

```
(define (cps-fact n k)
  (if (= n 0) (k 1) (cps-fact (- n 1) (lambda (v) (k (* n v))))) )
```

La fonction `cps-fact` peut être invoquée avec une continuation un peu particulière comme dans :

```
(call/cc (lambda (k) (* 2 (cps-fact 4 k)))) → 24
```

Exercice 3.14 : La fonction `the-current-continuation` peut aussi être définie comme à l'exercice 3.1.

```
(define (cc f)
  (let ((reified? #f))
    (let ((k (the-current-continuation)))
      (if reified? k (begin (set! reified? #t) (f k))) ) ) )
```

Merci à Luc Moreau de m'avoir suggéré cet exercice [Mor94].

Exercice 4.1 : Plusieurs techniques existent comme de retourner des résultats partiels ou bien d'user de continuations. Voici ces deux solutions :

```
(define (min-max1 tree)
  (define (mm tree)
    (if (pair? tree)
        (let ((a (mm (car tree)))
              (d (mm (cdr tree))) )
          (list (min (car a) (car d))
                (max (cadr a) (cadr d)) ) )
```

```
        (list tree tree) ) )
  (mm tree) )
(define (min-max2 tree)
  (define (mm tree k)
    (if (pair? tree)
        (mm (car tree)
            (lambda (mina maxa)
              (mm (cdr tree)
                  (lambda (mind maxd)
                    (k (min mina mind)
                       (max maxa maxd) ) ) ) ) )
        (k tree tree) ) )
  (mm tree list) )
```

La première solution consomme énormément de listes qui sont aussi promptement oubliées. On peut d'ailleurs imaginer, sur ce genre d'algorithme, d'effectuer des transformations éliminant cette consommation abusive, c'est la déforestation [Wad88]. La seconde consomme énormément de fermetures. La version du texte est, malgré les effets de bord dont on peut concevoir qu'ils la rendent insupportable auprès de certains, bien plus efficace.

Exercice 4.2 : Les fonctions suivantes sont préfixées par `q` afin de ne pas interférer avec les fonctions primitives.

```
(define (qons a d) (lambda (msg) (msg a d)))
(define (qar pair) (pair (lambda (a d) a)))
(define (qdr pair) (pair (lambda (a d) d)))
```

Exercice 4.3 : On reprend l'idée que deux paires pointées sont la même si, lorsque l'on modifie l'une, la modification est visible de l'autre.

```
(define (pair-eq? a b)
  (let ((tag (list 'tag))
        (original-car (car a)) )
    (set-car! a tag)
    (let ((result (eq? (car b) tag)))
      (set-car! a original-car)
      result ) ) )
```

Exercice 4.4 : On supposera avoir ajouté la clause de reconnaissance de la forme spéciale `or` dans `evaluate` sous la forme :

```
...
((or) (evaluate-or (cadr e) (caddr e) r s k)) ...
```

```
(define (evaluate-or e1 e2 r s k)
  (evaluate e1 r s (lambda (v ss)
                     (((v 'boolify)
                       (lambda () (k v ss))
                       (lambda () (evaluate e2 r s k)) ) ) )) )
```

Notez que β est évalué avec la mémoire `s` et non `ss`.

Exercice 4.5 : L'énoncé est quelque peu ambigu. Voici deux solutions qui diffèrent suivant que l'on retourne la valeur qu'avait la variable avant le calcul de la valeur à affecter ou après celui-ci.

```
(define (new1-evaluate-set! n e r s k)
```

```
    (evaluate e r s
      (lambda (v ss)
        (k (ss (r n)) (update ss (r n) v)) ) ) )
(define (new2-evaluate-set! n e r s k)
  (evaluate e r s
    (lambda (v ss)
      (k (s (r n)) (update ss (r n) v)) ) ) )
```

Ces deux programmations donnent des résultats différents pour l'expression suivante :

```
(let ((x 1))
  (set! x (set! x 2)) )
```

Exercice 4.6 : Le seul problème avec `apply` est que le dernier argument est une liste du Scheme interprété que l'on doit décoder pour la transformer en une liste de valeurs. Rappelons que le `-11` qui suit est l'étiquette permettant d'identifier la primitive `apply`.

```
(definitial apply
  (create-function
   -11 (lambda (v* s k)
         (define (first-pairs v*)
           ;; (assume (pair? v*))
           (if (pair? (cdr v*))
               (cons (car v*) (first-pairs (cdr v*)))
               '() ) )
         (define (terms-of v s)
           (if (eq? (v 'type) 'pair)
               (cons (s (v 'car)) (terms-of (s (v 'cdr)) s))
               '() ) )
       (if (>= (length v*) 2)
           (if (eq? ((car v*) 'type) 'function)
               (((car v*) 'behavior)
                (append (first-pairs (cdr v*))
                        (terms-of (car (last-pair (cdr v*))) s) )
                s k )
               (wrong "First argument not a function") )
           (wrong "Incorrect arity for apply") ) ) ) )
```

Pour `call/cc`, on effectue une allocation pour conférer à chaque continuation une étiquette unique.

```
(definitial call/cc
  (create-function
   -13 (lambda (v* s k)
         (if (= 1 (length v*))
       (if (eq? ((car v*) 'type) 'function)
           (allocate 1 s
            (lambda (a* ss)
              (((car v*) 'behavior)
               (list (create-function
                      (car a*)
                      (lambda (vv* sss kk)
                        (if (= 1 (length vv*))
                            (k (car vv*) sss)
                            (wrong "Incorrect arity") ) ) ) )
               ss k ) ) )
           (wrong "Non functional argument for call/cc") )
       (wrong "Incorrect arity for call/cc") ) ) ) )
```

Exercice 4.7 : La difficulté est de tester la compatibilité d'arité et de convertir une liste (du Scheme interprétant) de valeurs en une liste de valeurs du Scheme interprété.

```
(define (evaluate-nlambda n* e* r s k)
  (define (arity n*)
    (cond ((pair? n*) (+ 1 (arity (cdr n*))))
          ((null? n*) 0)
          (else       1) ) )
  (define (update-environment r n* a*)
    (cond
     ((pair? n*)
      (update-environment (update r (car n*) (car a*))
                          (cdr n*)
                          (cdr a*) ) )
     ((null? n*) r)
     (else (update r n* (car a*))) ) )
(define (update-store s a* v* n*)
  (cond ((pair? n*)
         (update-store (update s (car a*) (car v*))
                       (cdr a*) (cdr v*) (cdr n*) ) )
        ((null? n*) s)
        (else (allocate-list v* s (lambda (v ss)
                                    (update ss (car a*) v) ))) ) )
(allocate 1 s
  (lambda (a* ss)
    (k (create-function
        (car a*)
        (lambda (v* s k)
          (if (compatible-arity? n* v*)
              (allocate (arity n*) s
               (lambda (a* ss)
                 (evaluate-begin e*
                                 (update-environment r n* a*)
                                 (update-store ss a* v* n*)
                                 k ) ) )
              (wrong "Incorrect arity") ) ) )
        ss ) ) ) )
(define (compatible-arity? n* v*)
  (cond ((pair? n*) (and (pair? v*)
                         (compatible-arity? (cdr n*) (cdr v*)) ))
        ((null? n*) (null? v*))
        ((symbol? n*) #t) ) )
```

Exercice 5.1 : La preuve se fait par induction sur le nombre des termes d'une application.

Exercice 5.2 :

$\mathcal{L}[\![\texttt{(label}\ \nu\ \pi\texttt{)}]\!]\rho = (\mathbf{Y}\ \lambda\varepsilon.(\mathcal{L}[\![\pi]\!]\ \rho[\nu \rightarrow \varepsilon]))$

Exercice 5.3 :

$\mathcal{E}[\![\texttt{(dynamic}\ \nu\texttt{)}]\!]\rho\delta\kappa\sigma =$
let $\varepsilon = (\delta\ \nu)$
in **if** $\varepsilon =$ *no-such-dynamic-variable*

```
then  let α = (γ ν)
      in  if α = no-such-global-variable
          then wrong "No such variable"
          else (κ (σ α) σ)
          endif
else (κ ε σ)
endif
```

Exercice 5.4 : La macro transformera l'application en une série de suspensions que l'on évaluera selon un ordre non prescrit implanté par la fonction `determine!`.

```
(define-syntax unordered
  (syntax-rules ()
    ((unordered f) (f))
    ((unordered f arg ...)
     (determine! (lambda () f) (lambda () arg) ... ) ) ) )
(define (determine! . thunks)
  (let ((results (iota 0 (length thunks))))
    (let loop ((permut (random-permutation (length thunks))))
      (if (pair? permut)
          (begin (set-car! (list-tail results (car permut))
                           (force (list-ref thunks (car permut))) )
                 (loop (cdr permut)) )
          (apply (car results) (cdr results)) ) ) ) )
```

On notera que le choix de la permutation est entièrement fait au début, cette solution est donc inférieure à la dénotation du chapitre. Si la fonction `random-permutation` est définie comme suit, à l'aide d'une fonction de brassage `shuffle` :

```
(define (random-permutation n)
  (shuffle (iota 0 n)) )
```

alors, on pourra rendre le choix de la permutation entièrement dynamique grâce à `d.determine !` :

```
(define (d.determine! . thunks)
  (let ((results (iota 0 (length thunks))))
    (let loop ((permut (shuffle (iota 0 (length thunks)))))
      (if (pair? permut)
          (begin (set-car! (list-tail results (car permut))
                           (force (list-ref thunks (car permut))) )
                 (loop (shuffle (cdr permut))) )
          (apply (car results) (cdr results)) ) ) ) )
```

Exercice 6.1 : Une façon simpliste serait d'ajouter le nom de la variable (le symbole ou simplement son nom) comme argument supplémentaire du combinateur). Quelque chose comme :

```
(define (CHECKED-GLOBAL-REF- i n)
  (lambda ()
    (let ((v (global-fetch i)))
      (if (eq? v undefined-value)
          (wrong "Uninitialized variable" n)
          v ) ) ) )
```

Mais cela augmenterait la taille globale du code engendré. Une meilleure solution est que soit engendrée une table des symboles et qu'y soit puisé le nom de la variable fautive.

```
(define sg.current.names (list 'foo))
(define (stand-alone-producer e)
  (set! g.current (original.g.current))
  (let* ((m (meaning e r.init #t))
         (size (length g.current))
         (global-names (map car (reverse g.current))) )
    (lambda ()
      (set! sg.current (make-vector size undefined-value))
      (set! sg.current.names global-names)
      (set! *env* sr.init)
      (m) ) ) )
(define (CHECKED-GLOBAL-REF+ i)
  (lambda ()
    (let ((v (global-fetch i)))
      (if (eq? v undefined-value)
          (wrong "Uninitialized variable"
                 (list-ref sg.current.names i) )
          v ) ) ) )
```

Exercice 6.2 : La fonction `list` n'est en fait que `(lambda l l)` ; il suffit d'utiliser cette définition, en jouant directement avec les combinateurs :

```
(definitial list ((NARY-CLOSURE (SHALLOW-ARGUMENT-REF 0) 0)))
```

Exercice 6.3 : Il suffit pour cela d'imprimer le résultat du prétraitement en ayant eu soin préalablement de redéfinir chaque combinateur c en `(lambda args '(`c `. ,args))`.

Exercice 6.4 : La façon la plus directe d'y arriver est de modifier l'ordre d'évaluation des arguments des applications et donc d'adopter l'ordre droite-gauche.

```
(define (FROM-RIGHT-STORE-ARGUMENT m m* rank)
  (lambda ()
    (let* ((v* (m*))
           (v (m)) )
      (set-activation-frame-argument! v* rank v)
      v* ) ) )
(define (FROM-RIGHT-CONS-ARGUMENT m m* arity)
  (lambda ()
    (let* ((v* (m*))
           (v (m)) )
      (set-activation-frame-argument!
       v* arity (cons v (activation-frame-argument v* arity)) )
      v* ) ) )
```

On pourrait aussi modifier `meaning*` pour conserver l'ordre d'évaluation mais toujours allouer le bloc avant le calcul des arguments. On voit au passage que calculer le terme fonctionnel avant les arguments (quel que soit l'ordre dans lequel ils sont évalués) permet d'inspecter la fermeture obtenue pour, par exemple, allouer un bloc d'activation adapté au nombre de variables temporaires que celle-ci requerra : il n'y aura ainsi qu'une seule allocation et non deux.

Exercice 6.5 : La nouvelle forme spéciale est insérée dans la fonction `meaning` sous la forme de la ligne suivante :

```
... ((redefine) (meaning-redefine (cadr e))) ...
```

Puis l'on définit son effet :

```
(define (meaning-redefine n)
  (let ((kind1 (global-variable? g.init n)))
    (if kind1
        (let ((value (vector-ref sg.init (cdr kind1))))
          (let ((kind2 (global-variable? g.current n)))
            (if kind2
                (static-wrong "Already redefined variable" n)
                (let ((index (g.current-extend! n)))
                  (vector-set! sg.current index value) ) ) ) )
        (static-wrong "No such variable to redefine" n) )
    (lambda () 2001) ) )
```

Notez que la redéfinition est faite à la préparation et non à l'exécution. La valeur retournée par la forme `redefine` est quelconque.

Exercice 6.6 : Une fonction sans variable n'a pas besoin d'étendre l'environnement courant, ce qui ralentit d'un cran l'accès aux variables profondes. On changera `meaning-fix-abstraction` pour détecter le cas des suspensions et l'on définira un nouveau combinateur pour ce faire.

```
(define (meaning-fix-abstraction n* e+ r tail?)
  (let ((arity (length n*)))
    (if (= arity 0)
        (let ((m+ (meaning-sequence e+ r #t)))
          (THUNK-CLOSURE m+) )
        (let* ((r2 (r-extend* r n*))
               (m+ (meaning-sequence e+ r2 #t)) )
          (FIX-CLOSURE m+ arity) ) ) ) )
(define (THUNK-CLOSURE m+)
  (let ((arity+1 (+ 0 1)))
    (lambda ()
      (define (the-function v* sr)
        (if (= (activation-frame-argument-length v*) arity+1)
            (begin (set! *env* sr)
                   (m+) )
            (wrong "Incorrect arity") ) )
      (make-closure the-function *env*) ) ) )
```

Exercice 7.1 : On créera tout d'abord le registre :

```
(define *dynenv* -1)
```

On modifiera les fonctions préservant l'environnement :

```
(define (preserve-environment)
  (stack-push *dynenv*)
  (stack-push *env*) )
(define (restore-environment)
  (set! *env* (stack-pop))
  (set! *dynenv* (stack-pop)) )
```

Il n'y plus guère d'effort à faire pour trouver l'environnement dynamique mais, par contre, les manipulations de piles sont un peu altérées.

```
(define (search-dynenv-index)
  *dynenv* )
(define (pop-dynamic-binding)
```

```
    (stack-pop)
    (stack-pop)
    (set! *dynenv* (stack-pop)) )
  (define (push-dynamic-binding index value)
    (stack-push *dynenv*)
    (stack-push value)
    (stack-push index)
    (set! *dynenv* (- *stack-index* 1)) )
```

Exercice 7.2 : La fonction est simple :

```
(definitial load
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (make-primitive
     (lambda ()
       (if (= arity+1 (activation-frame-argument-length *val*))
           (let ((filename (activation-frame-argument *val* 0)))
             (set! *pc* (install-object-file! filename)) )
           (signal-exception
            #t (list "Incorrect arity" 'load) ) ) ) ) ) )
```

Mais cette définition pose cependant quelques problèmes. Analysez notamment ce qu'est la reprise d'une continuation capturée pendant le chargement d'un fichier. Par exemple, le fichier :

```
(display 'attention)
(call/cc (lambda (k) (set! *k* k)))
(display 'caution)
```

Invoquer, après chargement de ce fichier, la continuation `*k*` réimprimera-t-il le symbole `caution` ? Oui, avec cette implantation !

On notera aussi que, si l'on charge un fichier compilé définissant une variable globale, disons `bar`, qui était inconnue de l'application, alors elle restera inconnue de l'application.

Exercice 7.3 : Voici cette fonction :

```
(definitial global-value
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (define (get-index name)
      (let ((where (memq name sg.current.names)))
        (if where (- (length where) 1)
            (signal-exception
             #f (list "Undefined global variable" name) ) ) ) )
    (make-primitive
     (lambda ()
       (if (= arity+1 (activation-frame-argument-length *val*))
           (let* ((name (activation-frame-argument *val* 0))
                  (i (get-index name)) )
             (set! *val* (global-fetch i))
             (when (eq? *val* undefined-value)
               (signal-exception
                #f (list "Uninitialized variable" i) ) )
             (set! *pc* (stack-pop)) )
           (signal-exception
            #t (list "Incorrect arity" 'global-value) ) ) ) ) ) )
```

On notera, puisque l'on a réintroduit la possibilité de variable inexistante, que l'on doit bien sûr tester si la variable est bien initialisée.

Exercice 7.4 : Après avoir ajouté à la fonction `run-machine` l'allocation d'un vecteur où stocker les valeurs courantes des variables dynamiques :

```
... (set! *dynamics* (make-vector (+ 1 (length dynamics))
                                  undefined-value )) ; NEW
```

on redéfinira les fonctions d'accès appropriées.

```
(define (find-dynamic-value index)
  (let ((v (vector-ref *dynamics* index)))
    (if (eq? v undefined-value)
        (signal-exception #f (list "No such dynamic binding" index))
        v ) ) )
(define (push-dynamic-binding index value)
  (stack-push (vector-ref *dynamics* index))
  (stack-push index)
  (vector-set! *dynamics* index value) )
(define (pop-dynamic-binding)
  (let* ((index (stack-pop))
         (old-value (stack-pop)) )
    (vector-set! *dynamics* index old-value) ) )
```

Cette implantation est malheureusement fausse car l'accès immédiat se paye par le fait que les valeurs sauvegardées se trouvent maintenant dans la pile. Une capture de continuation ne capture que les valeurs à restaurer et non les valeurs courantes ; un échappement, en supprimant une tranche de pile, omet de restaurer les variables dynamiques à ce qu'elles valaient lorsque la forme `bind-exit` commença d'être évaluée. Il faut, pour corriger cela, une forme du style `unwind-protect` ou, plus simplement, abandonner l'implantation superficielle pour l'implantation profonde qui ne pose pas ce genre de problèmes et s'étend naturellement au parallélisme.

Exercice 7.5 : Avec le renommeur suivant, il est même possible d'intervertir deux variables en écrivant une liste de substitutions telles que `((fact fib) (fib fact))`. Attention, ce jeu peut se révéler dangereux !

```
(define (build-application-renaming-variables
         new-application-name application-name substitutions )
  (if (probe-file application-name)
      (call-with-input-file application-name
        (lambda (in)
          (let* ((dynamics     (read in))
                 (global-names (read in))
                 (constants    (read in))
                 (code         (read in))
                 (entries      (read in)) )
            (close-input-port in)
            (write-result-file
             new-application-name
             (list ";;; renamed variables from " application-name)
             dynamics
             (let sublis ((global-names global-names))
               (if (pair? global-names)
                   (cons (let ((s (assq (car global-names)
                                        substitutions )))
```

```
                        (if (pair? s) (cadr s)
                            (car global-names) ) )
                      (sublis (cdr global-names)) )
                global-names ) )
          constants
          code
          entries ) ) ) )
    (signal #f (list "No such file" application-name)) ) )
```

Exercice 7.6 : Il faut juste faire attention à modifier la bonne instruction !

```
(define-instruction (CHECKED-GLOBAL-REF i) 8
  (set! *val* (global-fetch i))
  (if (eq? *val* undefined-value)
    (signal-exception #t (list "Uninitialized variable" i))
    (vector-set! *code* (- *pc* 2) 7) ) )
```

Exercice 8.1 : Ce n'est pas nécessaire car il ne saurait y avoir deux variables de même nom. Ceci interdit que la liste des variables soit cyclique.

Exercice 8.2 : Voici l'astuce :

```
(define (prepare e)
  (eval/ce `(lambda () ,e)) )
```

Exercice 8.3 :

```
(define (eval/at e)
  (let ((g (gensym)))
    (eval/ce `(lambda (,g) (eval/ce ,g))) ) )
```

Exercice 8.4 : En programmant un rattrapeur d'erreur approprié.

```
(set! variable-defined?
      (lambda (env name)
        (bind-exit (return)
          (monitor (lambda (c ex) (return #f))
            (eval/b name env)
            #t ) ) ) )
```

Exercice 8.5 : On passera sous silence le traitement de la forme spéciale `monitor` qui apparaît dans la définition de l'interprète réflexif. Après tout, si l'on ne commet pas d'erreur, `monitor` est similaire à `begin`. Ce qui suit n'est pas du Scheme conforme car la définition des formes spéciales requiert que leur nom soit utilisé en tant que variable, ce qui n'est pas légal. Cela marche toutefois dans de nombreux implantations de Scheme. Pour la forme `the-environment`, on la définira pour capturer les liaisons nécessaires :

```
(define-syntax the-environment
  (syntax-rules ()
    ((the-environment)
     (capture-the-environment make-toplevel make-flambda flambda?
      flambda-behavior prompt-in prompt-out exit it extend error
      global-env toplevel eval evlis eprogn reference quote if set!
      lambda flambda monitor ) ) ) )
(define-syntax capture-the-environment
```

```
(syntax-rules ()
  ((capture-the-environment word ...)
   (lambda (name . value)
     (case name
       ((word) ((handle-location word) value)) ...
       ((display) (if (pair? value)
                      (wrong "Immutable" 'display)
                      show ))
       (else (if (pair? value)
                 (set-top-level-value! name (car value))
                 (top-level-value name) )) ) ) ) ) )
(define-syntax handle-location
  (syntax-rules ()
    ((handle-location name)
     (lambda (value)
       (if (pair? value)
           (set! name (car value))
           name ) ) ) ) )
```

On définira enfin les fonctions de manipulation d'environnement de première classe : `variable-defined?`, `variable-value` et `set-variable-value!`.

```
(define undefined (cons 'un 'defined))
(define-class Envir Object
  ( name value next ) )
(define (enrich env . names)
  (let enrich ((env env)(names names))
    (if (pair? names)
        (enrich (make-Envir (car names) undefined env) (cdr names))
        env ) ) )
(define (variable-defined? name env)
  (if (Envir? env)
      (or (eq? name (Envir-name env))
          (variable-defined? name (Envir-next env)) )
      #t ) )
(define (variable-value name env)
  (if (Envir? env)
      (if (eq? name (Envir-name env))
          (let ((value (Envir-value env)))
            (if (eq? value undefined)
                (error "Uninitialized variable" name)
                value ) )
          (variable-value name (Envir-next env)) )
      (env name) ) )
```

Comme on le voit, l'environnement est une liste chainée de nœuds se terminant par une fermeture. L'interprète réflexif peut maintenant tourner !

Exercice 9.1 : On peut utiliser les macros hygiéniques de Scheme et écrire :

```
(define-syntax repeat1
  (syntax-rules (:while :unless :do)
    ((_ :while p :unless q :do body ...)
     (let loop ()
       (if p (begin (if (not q) (begin body ...))
                    (loop) )) ) ) ) )
```

On peut aussi utiliser directement `define-abbreviation` et écrire :

```
(with-aliases ((+let let) (+begin begin) (+when when) (+not not))
```

```
(define-abbreviation (repeat2 . parms)
  (let ((p    (list-ref parms 1))
        (q    (list-ref parms 3))
        (body (list-tail parms 5))
        (loop (gensym)) )
    `(,+let ,loop ()
          (,+when ,p (,+begin (,+when (,+not ,q) . ,body)
                              (,loop) )) ) ) ) )
```

Exercice 9.2 : La difficulté est ici de faire de l'arithmétique avec le langage des macros de Scheme. On peut approcher une solution en engendrant des appels à l'exécution à la fonction `length` sur des listes de longueur sélectionnées.

```
(define-syntax enumerate
  (syntax-rules ()
    ((enumerate) (display 0))
    ((enumerate e1 e2 ...)
     (begin (display 0) (enumerate-aux e1 (e1) e2 ...) ) ) ) )
(define-syntax enumerate-aux
  (syntax-rules ()
    ((enumerate-aux e1 len) (begin e1 (display (length 'len))))
    ((enumerate-aux e1 len e2 e3 ...)
     (begin e1 (display (length 'len))
            (enumerate-aux e2 (e2 . len) e3 ...) ) ) ) )
```

Exercice 9.3 : Il suffit en fait de modifier la fonction `make-macro-environment` afin que tous les étages soient confondus.

```
(define (make-macro-environment current-level)
  (let ((metalevel (delay current-level)))
    (list (make-Magic-Keyword 'eval-in-abbreviation-world
           (special-eval-in-abbreviation-world metalevel) )
          (make-Magic-Keyword 'define-abbreviation
           (special-define-abbreviation metalevel))
          (make-Magic-Keyword 'let-abbreviation
           (special-let-abbreviation metalevel))
          (make-Magic-Keyword 'with-aliases
           (special-with-aliases metalevel) ) ) ) )
```

Exercice 9.4 : La conversion s'écrit comme l'âne qui trotte. Le seul véritable point d'intérêt est la façon dont on renomme les variables. On maintiendra une A-liste pour mémoriser ces correspondances.

```
(define-generic (->Scheme (e) r))
(define-method (->Scheme (e Alternative) r)
  `(if ,(->Scheme (Alternative-condition e) r)
       ,(->Scheme (Alternative-consequent e) r)
       ,(->Scheme (Alternative-alternant e) r) ) )
(define-method (->Scheme (e Local-Assignment) r)
  `(set! ,(->Scheme (Local-Assignment-reference e) r)
         ,(->Scheme (Local-Assignment-form e) r) ) )
(define-method (->Scheme (e Reference) r)
  (variable->Scheme (Reference-variable e) r) )
(define-method (->Scheme (e Function) r)
  (define (renamings-extend r variables names)
    (if (pair? names)
```

```
        (renamings-extend (cons (cons (car variables) (car names)) r)
                          (cdr variables) (cdr names) )
        r ) )
(define (pack variables names)
  (if (pair? variables)
      (if (Local-Variable-dotted? (car variables))
          (car names)
          (cons (car names) (pack (cdr variables) (cdr names))) )
      '() ) )
(let* ((variables (Function-variables e))
       (new-names (map (lambda (v) (gensym))
                       variables ))
       (newr (renamings-extend r variables new-names)) )
  `(lambda ,(pack variables new-names)
     ,(->Scheme (Function-body e) newr)) ) )
(define-generic (variable->Scheme (e) r))
```

Exercice 9.5 : Tel qu'a été écrit MEROONET, les deux mondes (d'expansion et d'exécution) sont assez mêlés et quelques-unes des ressources appartiennent aux deux mondes à la fois (par exemple, `register-class` qui est invoquée à l'expansion et au chargement du fichier).

Exercice 10.1 : Il suffit de modifier la fonction `SCM_invoke` et de changer le protocole d'appel des fermetures de petite arité fixe. Il suffit de reprendre celui des primitives d'arité fixe en n'omettant pas de fournir l'objet représentant la fermeture en premier argument. Il faut aussi changer l'interface des fonctions C engendrées et leur faire adopter la signature adoptée.

Exercice 10.2 : Les variable globales vont être raffinées pour comporter un champ supplémentaire indiquant si elles sont sûrement initialisées ou pas. La détection des variables globales libres sera ajustée en conséquence.

```
(define-class Global-Variable Variable (initialized?))
(define (objectify-free-global-reference name r)
  (let ((v (make-Global-Variable name #f)))
    (set! g.current (cons v g.current))
    (make-Global-Reference v) ) )
```

L'analyse sera insérée dans le compilateur et sera le fait de l'interaction entre l'arpenteur générique et la fonction générique `inian!`.

```
(define (compile->C e out)
  (set! g.current '())
  (let ((prg (extract-things!
              (lift! (initialization-analyze! (Sexp->object e))) )))
    (gather-temporaries! (closurize-main! prg))
    (generate-C-program out e prg) ) )
(define (initialization-analyze! e)
  (call/cc (lambda (exit) (inian! e (lambda () (exit 'finished)))))
  e )
(define-generic (inian! (e) exit)
  (update-walk! inian! e exit) )
```

Le problème est maintenant de suivre le flot de calcul et de déterminer les variables globales qui seront sûrement écrites avant d'être lues. Plutôt que de développer une

analyse précise et complexe, on s'attachera à déterminer un sous-ensemble des variables sûrement initialisées. C'est le fait des cinq méthodes suivantes :

```
(define-method (inian! (e Global-Assignment) exit)
  (call-next-method)
  (let ((gv (Global-Assignment-variable e)))
    (set-Global-Variable-initialized?! gv #t)
    (inian-warning "Surely initialized variable" gv)
    e ) )
(define-method (inian! (e Global-Reference) exit)
  (let ((gv (Global-Reference-variable e)))
    (cond ((Predefined-Variable? gv) e)
          ((Global-Variable-initialized? gv) e)
          (else (inian-error "Surely uninitialized variable" gv)
                (exit) ) ) ) )
(define-method (inian! (e Alternative) exit)
  (inian! (Alternative-condition e) exit)
  (exit) )
(define-method (inian! (e Application) exit)
  (call-next-method)
  (exit) )
(define-method (inian! (e Function) exit)
  e )
```

Cette analyse suit le calcul, détermine les affectations et s'arrête dès que le calcul devient trop compliqué à suivre, c'est-à-dire dès qu'une fonction est appliquée ou qu'une alternative apparaît. En revanche, il n'est pas nécessaire de regarder le corps des abstractions puisqu'une fermeture se calcule en temps fini et sans erreur.

Exercice 11.1 : Le prédicat `Object?` peut être renforcé et être moins erroné si l'on réserve une nouvelle composante dans les vecteurs, représentant des objets, pour contenir une étiquette. Après avoir créé cette étiquette, on modifiera le prédicat `Object?`, sans oublier tous les points d'allocation où il faudra ajouter cette étiquette et notamment l'amorçage, c'est-à-dire la définition des classes prédéfinies.

```
(define *starting-offset* 2)
(define meroonet-tag (cons 'meroonet 'tag))
(define (Object? o)
  (and (vector? o)
       (>= (vector-length o) *starting-offset*)
       (eq? (vector-ref o 1) meroonet-tag) ) )
```

Cette modification améliore un peu la résistance du prédicat `Object?` (mais pas sa vitesse). Le prédicat peut cependant encore être berné si l'étiquette d'un objet est extraite (par `vector-ref`) et est insérée dans un vecteur quelconque.

Exercice 11.2 : La fonction est générique ainsi l'on pourra la particulariser pour certaines classes. La version qui suit présente une surconsommation de paires :

```
(define-generic (clone (o))
  (list->vector (vector->list o)) )
```

Exercice 11.3 : On définira donc un nouveau type de classe, la méta-classe `Counting-Class`, avec un champ supplémentaire pour compter les allocations qui seront effectuées.

```
(define-class CountingClass Class (counter))
```

Le code de MEROONET a été écrit pour que les modifications à apporter ne remettent pas tout en question. Il doit être possible de définir une classe avec cette méta-classe.

```
(define-meroonet-macro (define-CountingClass name super-name
                                             own-field-descriptions )
  (let ((class (register-CountingClass
                name super-name own-field-descriptions )))
    (generate-related-names class) ) )
(define (register-CountingClass name super-name
                                own-field-descriptions )
  (initialize! (allocate-CountingClass)
               name
               (->Class super-name)
               own-field-descriptions ) )
```

Une meilleure façon de faire serait d'étendre la syntaxe de `define-class` pour accepter une option indiquant la méta-classe à utiliser (et, par défaut, `Class`). Il faudrait rendre aussi génériques de nombreuses fonctions comme :

```
(define-generic (generate-related-names (class)))
(define-method (generate-related-names (class Class))
   (Class-generate-related-names class) )
(define-generic (initialize! (o) . args))
(define-method (initialize! (o Class) . args)
  (apply Class-initialize! o args) )
(define-method (initialize! (class CountingClass) . args)
   (set-CountingClass-counter! class 0)
   (call-next-method) )
```

Les allocateurs dans les fonctions d'accompagnement doivent être modifiés pour entretenir le compteur `counter`.

```
(define-method (generate-related-names (class CountingClass))
  (let ((cname (symbol-append (Class-name class) '-class))
        (alloc-name (symbol-append 'allocate- (Class-name class)))
        (make-name (symbol-append 'make- (Class-name class))) )
    `(begin ,(call-next-method)
            (set! ,alloc-name           ; patch the allocator
                  (let ((old ,alloc-name))
                    (lambda sizes
                      (set-CountingClass-counter!
                       ,cname (+ 1 (CountingClass-counter ,cname)) )
                      (apply old sizes) ) ) )
            (set! ,make-name            ; patch the maker
                  (let ((old ,make-name))
                    (lambda args
                      (set-CountingClass-counter!
                       ,cname (+ 1 (CountingClass-counter ,cname)) )
                      (apply old args) ) ) ) ) ) )
```

Il ne nous reste plus qu'à donner un exemple d'emploi :

```
(define-CountingClass CountedPoint Object (x y))
(unless (and (= 0 (CountingClass-counter CountedPoint-class))
             (allocate-CountedPoint)
             (= 1 (CountingClass-counter CountedPoint-class))
             (make-CountedPoint 11 22)
             (= 2 (CountingClass-counter CountedPoint-class)) )
```

```
;; should not be evaluated if everything is OK
(meroonet-error "Failed test on CountedPoint") )
```

Exercice 11.4 : On peut encore créer une nouvelle méta-classe `ReflectiveClass` avec les champs supplémentaires `predicate`, `allocator` et `maker` puis modifier la génération des fonctions d'accompagnement pour que ces champs soient remplis lors de la définition de la classe. On agirait de même pour les champs.

```
(define-class ReflectiveClass Class (predicate allocator maker))
(define-method (generate-related-names (class ReflectiveClass))
  (let ((cname (symbol-append (Class-name class) '-class))
        (predicate-name (symbol-append (Class-name class) '?))
        (allocator-name (symbol-append 'allocate- (Class-name class)))
        (maker-name (symbol-append 'make- (Class-name class))) )
    `(begin ,(call-next-method)
            (set-ReflectiveClass-predicate! ,cname ,predicate-name)
            (set-ReflectiveClass-allocator! ,cname ,allocator-name)
            (set-ReflectiveClass-maker! ,cname ,maker-name) ) ) )
```

Exercice 11.5 : Le problème essentiel est de pouvoir détecter si une fonction générique existe. Comme il n'est pas possible de savoir en Scheme si une variable globale existe ou pas, nous nous rabattrons sur la liste `*generics*` qui, heureusement pour nous, contient l'ensemble des fonctions génériques connues.

```
(define-meroonet-macro (define-method call . body)
  (parse-variable-specifications
   (cdr call)
   (lambda (discriminant variables)
     (let ((g (gensym))(c (gensym)))    ; make g and c hygienic
       `(begin
          (unless (->Generic ',(car call))
            (define-generic ,call) )        ; new
          (register-method
           ',(car call)
           (lambda (,g ,c)
             (lambda ,(flat-variables variables)
               (define (call-next-method)
                 ((if (Class-superclass ,c)
                      (vector-ref (Generic-dispatch-table ,g)
                                  (Class-number
                                   (Class-superclass ,c) ) )
                      (Generic-default ,g) )
                  . ,(flat-variables variables) ) )
               . ,body ) )
           ',(cadr discriminant)
           ',(cdr call) ) ) ) ) ) )
```

Exercice 11.6 : Toute méthode dispose maintenant de deux fonctions locales nommées `call-next-method` et `next-method?`, il serait plus astucieux de ne les engendrer que si le corps de la méthode contient des appels à ces fonctions.

```
(define-meroonet-macro (define-method call . body)
  (parse-variable-specifications
   (cdr call)
   (lambda (discriminant variables)
     (let ((g (gensym))(c (gensym)))    ; make g and c hygienic
```

```
    `(register-method
      ',(car call)
      (lambda (,g ,c)
        (lambda ,(flat-variables variables)
          ,@(generate-next-method-functions g c variables)
          . ,body ) )
      ',(cadr discriminant)
      ',(cdr call) ) ) ) ) )
```

La fonction `next-method?` s'inspire de `call-next-method` mais ne fait que vérifier que la méthode existe au lieu de l'invoquer.

```
(define (generate-next-method-functions g c variables)
  (let ((get-next-method (gensym)))
    `((define (,get-next-method)
        (if (Class-superclass ,c)
            (vector-ref (Generic-dispatch-table ,g)
                        (Class-number (Class-superclass ,c)) )
            (Generic-default ,g) ) )
      (define (call-next-method)
        ((,get-next-method) . ,(flat-variables variables)) )
      (define (next-method?)
        (not (eq? (,get-next-method) (Generic-default ,g))) ) ) ) )
```

Bibliographie

[85M85] *Macscheme Reference Manual*. Semantic Microsystems, Sausalito, California, 1985.

[AG85] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.

[All78] John Allen. *Anatomy of Lisp*. Computer Science Series. McGraw-Hill, 1978.

[ALQ95] Sophie Anglade, Jean-Jacques Lacrampe, and Christian Queinnec. Semantics of combinations in scheme. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 7(4) :15–20, October–December 1995.

[App87] Andrew Appel. Garbage Collection can be faster than Stack Allocation. *Information Processing Letters*, 25(4) :275–279, June 1987.

[App92a] Andrew Appel. *Compiling with continuations*. Cambridge Press, 1992.

[App92b] Apple Computer, Eastern Research and Technology. *Dylan, An object-oriented dynamic language*. Apple Computer, Inc., April 1992.

[AR88] Norman Adams and Jonathan Rees. Object-oriented programming in Scheme. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 277–288, August 1988.

[AS94] Andrew W Appel and Zhong Shao. An empirical and analytic study of stack vs heap cost for languages with closures. Technical Report CS-TR-450-94, Princeton University, Department of Computer Science, Princeton (New-Jersey USA), March 1994.

[Att95] Giuseppe Attardi. The embeddable common Lisp. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 8(1) :30–41, January–April 1995. LUV-94, Fourth International LISP Users and Vendors Conference.

[Bak78] Henry G Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4) :280–294, April 1978.

[Bak92a] Henry G Baker. Inlining semantics for subroutines which are recursive. *SIGPLAN Notices*, 27(12) :39–46, December 1992.

[Bak92b] Henry G Baker. Metacircular semantics for common Lisp special forms. *Lisp Pointers*, 5(4) :11–20, 1992.

[Bak93] Henry G Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4) :2–27, October 1993.

[Bar84] H P Barendregt. *The Lambda Calculus*. Number 103 in Studies in Logic. North Holland, Amsterdam, 1984.

[Bar89] Joel F. Bartlett. Scheme->c a portable scheme-to-c compiler. Research Report 89 1, DEC Western Research Laboratory, Palo Alto, California, January 1989.

[Baw88] Alan Bawden. Reification without evaluation. In *Conference Record of the 1988 ACM Symposium on LISP and Functional Programming*, pages 342–351, Snowbird, Utah, July 1988. ACM Press.

[BC87] Jean-Pierre Briot and Pierre Cointe. A uniform model for object-oriented languages using the class abstraction. In *IJCAI '87*, pages 40–43, 1987.

[BCSJ86] Jean-Pierre Briot, Pierre Cointe, and Emmanuel Saint-James. Réécriture et récursion dans une fermeture, étude dans un Lisp à liaison superficielle et application aux objets. In Pierre Cointe and Jean Bézivin, editors, 3èmes *journées LOO/AFCET*, number 48 in Revue Bigre+Globule, pages 90–100, IRCAM, Paris (France), January 1986.

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification. *SIGPLAN Notices*, 23, September 1988. special issue.

[BDP+04] Anne Brygoo, Titou Durand, Maryse Pelletier, Christian Queinnec, and Michèle Soria. *Programmation récursive (en Scheme)*. Dunod, December 2004.

[BDP+06] Anne Brygoo, Titou Durand, Maryse Pelletier, Christian Queinnec, and Michèle Soria. *Programmation récursive – annales corrigées novembre 2001-septembre 2005*. Paracamplus, Paris, France, 2006.

[Bel73] James R Bell. Threaded code. *Communications of the ACM*, 16(6) :370–372, June 1973.

[Bet91] David Michael Betz. *XSCHEME : An Object-oriented Scheme*. P.O. Box 144, Peterborough, NH 03458 (USA), July 1991. version 0.28.

[BG94] Henri E Bal and Dick Grune. *Programming Language Essentials*. Addison Wesley, 1994.

[BHY87] Adrienne Bloss, Paul Hudak, and Jonathan Young. Code optimizations for lazy evaluation. *International journal on Lisp and Symbolic Computation*, 1(2) :147–164, 1987.

[BJ86] David H Bartley and John C Jensen. The implementation of pc Scheme. In *Conference Record of the 1986 ACM Symposium on LISP and Functional Programming*, pages 86–93, Cambridge, Massachusetts, August 1986. ACM Press.

[BKK+86] D G Bobrow, K Kahn, G Kiczales, L Masinter, M Stefik, and F Zdybel. Commonloops : Merging Lisp and object oriented programming. In

OOPSLA '86 — Object-Oriented Programming Systems and LAnguages, pages 17–29, Portland (Oregon, USA), 1986.

[Blo00] Laurent Bloch. *Initiation à la programmation avec Scheme*. Éditions Technip, 2000. ISBN 2-7108-0795-5.

[BM84] Robert S Boyer and J Struther Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the Association for Computing Machinery*, 31(3) :441–485, July 1984.

[BR88] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Symposium on LISP and Functional Programming*, Salt Lake City, Utah., July 1988.

[BW88] Hans J Boehm and M Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9), September 1988.

[Car93] Luca Cardelli. An implementation of $F_{<}$. Research Report 97, DEC-SRC, February 1993.

[Car94] George J. Carrette. *Siod, Scheme In One Defun*. PARADIGM Associates Incorporated, 29 Putnam Ave, Suite 6, Cambridge, MA 02138, USA, 1994.

[Cay83] Michel Cayrol. *Le Langage Lisp*. Cepadues Editions, Toulouse (France), 1983.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77 — 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977. ACM Press.

[CEK+89] L W Cannon, R A Elliott, L W Kirchhoff, J H Miller, J M Milner, R W Mitze, E P Schan, N O Whittington, H Spencer, and D Keppel. *Recommended C Style and Coding Standards*, November 1989.

[CG77] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in LISP. *Communications of the ACM*, 20(2) :78–87, February 1977.

[CH94] William D Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programmming* [lfp94], pages 128–139.

[Cha80] Jérôme Chailloux. *Le modèle VLISP : description, implémentation et évaluation*. Thèse de troisième cycle, Université de Vincennes, April 1980. Rapport LITP 80-20, Paris (France).

[Cha94] Gregory J Chaitin. The limits of mathematics. IBM PO Box 704, Yorktown Heights, NY 10598 (USA), July 1994.

[Cha96] Jacques Chazarain. *Programmer avec Scheme, de la pratique à la théorie*. International Thomson Publishing, 1996.

[CHO88] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for continuations. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 124–131, August 1988.

[Chr95] Christophe. *La Famille Fenouillard*. Armand Colin, 1895.

[Cla79] Douglas W. Clark. Measurements of dynamic list structure use in Lisp. *IEEE Transactions on Software Engineering*, 5(1) :51–59, January 1979.

[Cli84] William Clinger. The Scheme 311 compiler : an exercise in denotational semantics. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, 1984.

[Coi87] Pierre Cointe. The ObjVlisp kernel : a reflexive architecture to define a uniform object oriented system. In P. Maes and D. Nardi, editors, *Workshop on MetaLevel Architectures and Reflection*, Alghiero, Sardinia (Italy), October 1987. North Holland.

[Com84] Douglas Comer. *Operating System Design : The XINU Approach*. Prentice-Hall, 1984.

[CR91a] William Clinger and Jonathan Rees. Macros that work. In *POPL '91 – Eighteenth Annual ACM symposium on Principles of Programming Languages*, pages 155–162, Orlando, (Florida USA), January 1991.

[CR91b] William Clinger and Jonathan A Rees. The revised^4 report on the algorithmic language Scheme. *Lisp Pointer*, 4(3), 1991.

[Cur89] Pavel Curtis. (algorithms). *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 3(1) :48–61, July 1989.

[Dan87] Olivier Danvy. Memory allocation and higher-order functions. In *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, SIGPLAN Notices, Vol. 22, No 7, pages 241–252, Saint-Paul, Minnesota, June 1987. ACM, ACM Press.

[DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principle*, 1991.

[Del89] Vincent Delacour. Picolo expresso. *Revue Bigre+Globule*, (65) :30–42, July 1989.

[Deu80] L Peter Deutsch. Bytelisp and its alto implementation. In *Conference Record of the 1980 LISP Conference* [lfp80], pages 231–242.

[Deu89] Alain Deutsch. Génération automatique d'interprèteurs et compilation à partir de spécifications dénotationnelles. Rapport de DEA-LAP 1988 LITP-RXF 89-17, LITP, January 1989.

[Dev85] Matthieu Devin. Le portage du système Le-Lisp. Rapport technique 50, INRIA-Rocquencourt, May 1985.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *LFP '90 – ACM Symposium on Lisp and Functional Programming*, pages 151–160, Nice (France), June 1990.

[DFH86] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style : Beyond conventional macros. In *Conference Record of the 1986 ACM Conference on Lisp and Functional Programming*, pages 143–150, 1986.

[DFH88] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style : a general macro mechanism. *International journal on Lisp and Symbolic Computation*, 1(1) :53–76, June 1988.

[DH92] Olivier Danvy and John Hatcliff. Thunks (continued). In *Proceedings of the Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 3–11, Bordeaux, France, September 1992. IRISA, Rennes, France. Extended version available as Technical Report CIS-92-28, Kansas State University.

[DHB93] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *International journal on Lisp and Symbolic Computation*, 5(4) :295–326, 1993.

[DHHM92] R Ducournau, M Habib, M Huchard, and M L Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *OOPSLA '92 — Object-Oriented Programming Systems and LAnguages*, pages 16–4, 1992.

[Dil88] Antoni Diller. *Compiling Functional Languages*. John Wiley and sons, 1988.

[DM88] Olivier Danvy and Karoline Malmkjær. A Blond primer. DIKU report 88/21, University of Copenhagen, Copenhagen, Denmark, 1988.

[dM95] Antoine Dumesnil de Maricourt. *Macro-expansion en Lisp, sémantique et réalisation*. Thèse d'université, Université Paris 7, Paris (France), June 1995.

[DPS94a] Harley Davis, Pierre Parquier, and Nitsan Séniak. Sweet harmony : the talk/c++ connection. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programmming* [lfp94], pages 121–127.

[DPS94b] Harley Davis, Pierre Parquier, and Nitsan Séniak. Talking about modules and delivery. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programmming* [lfp94], pages 113–120.

[dR87] Jim des Rivières. Control-related meta-level facilities in Lisp. In P. Maes and D. Nardi, editors, *Workshop on Meta-Level Architecture and Reflection*, Alghiero, Sardinia (Italy), October 1987. North Holland.

[dRS84] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 331–347, Austin, Texas, August 1984. ACM Press.

[Dyb87] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[ES70] Jay Earley and Howard Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13(10) :607–617, October 1970.

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *POPL '88 – Fifteenth Annual ACM symposium on Principles of Programming Languages*, pages 180–190, San Diego (California USA), January 1988.

[Fel90] Matthias Felleisen. On the expressive power of programming languages. In Neil Jones, editor, *ESOP '90 – European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 134–151, Copehaguen (Danmark), 1990. Springer-Verlag.

[FF87] Matthias Felleisen and Daniel P. Friedman. A reduction semantics for imperative higher-order languages. *Parallel Architectures and Languages Europe*, 259 :206–223, 1987.

[FF89] Matthias Felleisen and Daniel P Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3) :243–287, 1989. Preliminary version in : *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987, 314-325.

[FFDM87] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Computer Science Dept. Technical Report 216, Indiana University, Bloomington, Indiana, February 1987.

[FFFK03] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs – An Introduction to Computing and Programming*. MIT Press, Cambridge MA, 2003.

[FH89] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. Computer Science Technical Report No. 100, Rice University, June 1989.

[FL87] Marc Feeley and Guy LaPalme. Using closures for code generation. *Journal of Computer Languages*, 12(1) :47–66, 1987.

[FM90] Marc Feeley and James S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[FW76] D.P. Friedman and D.S. Wise. Cons should not evaluate its arguments. In Michaelson and Milner, editors, *Proceedings of the 3rd International Colloquium on "Automata, Languages and Programming"*, pages 257–284. Edinburgh University Press, July 1976.

[FW84] Daniel P. Friedman and Mitchell Wand. Reification : Reflection without metaphysics. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 348–355, Austin, TX., August 1984.

[FWFD88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce Duba. Abstract continuations : a mathematical semantics for handling functional jumps. In *Proceedings of the 1988 ACM Symposium on LISP and Functional Programming*, Salt Lake City, Utah., July 1988.

[FWH92] Daniel P Friedman, Mitchell Wand, and Christopher Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge MA and McGraw-Hill, 1992.

[Gab88] Richard P Gabriel. The why of y. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 2(2) :15–25, 1988.

[GBM82] Martin L. Griss, Eric Benson, and Gerald Q Maguire. Psl : A portable LISP system. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pages 88–97, Pittsburgh, Pennsylvania, August 1982. ACM Press.

[Gol02] Mayer Goldberg. A variadic extension of curry's fixed-point combinator. In Olin Shivers, editor, *Scheme 2002 — Proceedings of the Third Workshop on Scheme and Functional Programming*, volume Georgia Tech, Technical Report GIT-CC-02-48, Pittsburgh (Pennsylvania, USA), October 2002.

[Gor75] Michael J C Gordon. Operational reasoning and denotational semantics. In G. Huet and G. Kahn, editors, *Actes du Colloque IRIA "Constructions et Justifications de Programmes"*, pages 83–98, Arc et Senans, July 1975.

[Gor88] Michael J C Gordon. *Programming Language Theory and its Implementation*. International Series in Computer Science. Prentice-Hall, 1988.

[GP88] Richard P Gabriel and Kent M Pitman. Technical issues of separation in function cells and value cells. *International journal on Lisp and Symbolic Computation*, 1(1) :81–101, June 1988.

[GR83] Adèle Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation*. Addison Wesley, 1983.

[Gra93] Paul Graham. *On Lisp, Advanced Techniques for Common Lisp*. Prentice-Hall, 1993.

[Gre77] Patrick Greussay. *Contribution à la définition interprétative et à l'implémentation des Lambda-langages*. Thèse d'état, Université Paris VI, November 1977. Rapport LITP 78-2.

[Gud93] David Gudeman. Representing type information in dynamically typed languages. Technical Report 93–27, Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA, October 1993.

[HD90] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *PPOPP '90 – ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 128–136, Seattle (Washington US), March 1990.

[HDB90] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, White Plains, New York, June 1990.

[Hen80] Peter Henderson. *Functional Programming, Application and Implementation*. International Series in Computer Science. Prentice-Hall, 1980.

[Hen92a] Fritz Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 205–215, San Francisco, USA, June 1992.

[Hen92b] Wade Hennessey. Wcl : Delivering efficient COMMON LISP applications under UN⋆X. In *Conference Record of the 1992 ACM Symposium on LISP and Functional Programming*, pages 260–269, San Francisco, California, June 1992. ACM Press.

[HFW84] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, TX., 1984.

[HKK99] Max Hailperin, Barbara Kaiser, and Karl Knight. *Concrete Abstractions, An Introduction to Computer Science Using Scheme*. Brooks/Cole Publishing Company, a division of International Thomson Publishing Inc, 1999. ISBN 0-534-95211-9.

[Hof93] Ulrich Hoffmann. Using c as target code for translating highlevel programming languages. Technical Report APPLY/CAU/IV.4/2, Christian-Albrechts-Universitey of Kiel, 1993.

[Hon93] P. Joseph Hong. Threaded code designs for forth interpreters. *SIG FORTH, Newsletter of the ACM's Special Interest Group on FORTH*, 4(2) :11–18, fall 1993.

[HS75] Carl Hewitt and Brian Smith. Towards a programming apprentice. *IEEE Transactions on Software Engineering*, 1(1) :26–45, March 1975.

[HS91] Samuel P Harbison and Guy L Steele, Jr. *C : A Reference Manual*. Prentice-Hall, 1991.

[Huf96] Jean-Michel Hufflen. *Programmation fonctionnelle en Scheme. De la conception à la mise en oeuvre*. Masson, 1996.

[IEE91] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[ILO94] ILOG, 2 Avenue Galliéni, BP 85, 94253 Gentilly, France. *Ilog-Talk Reference Manual*, 1994.

[IM89] Takayasu Ito and Manabu Matsui. A parallel lisp language PaiLisp and its kernel specification. In Takayasu Ito and Robert H Halstead, Jr., editors, *Proceedings of the US/Japan Workshop on Parallel Lisp*, volume Lecture Notes in Computer Science 441, pages 58–100, Sendai (Japan), June 1989. Springer-Verlag.

[IMY92] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCL : A reflective object-oriented concurrent language without a run-time kernel. In Yonezawa and Smith [YS92], pages 24–35.

[ISO90] ISO/IEC 9899 :1990. Programmming language — c. Technical report, Information Technology, 1990.

[ISO94] ISO-IEC/JTC1/SC22/WG16. Programming language ISLISP, CD 13816. Technical report, ISO-IEC/JTC1/SC22/WG16, 1994.

[ITW86] Takayasu Ito, Takashi Tamura, and Shin-ichi Wada. Theoretical comparisons of interpreted/compiled executions of Lisp on sequential and parallel machine models. In H J Kugler, editor, *Proceedings of the IFIP 10th World Computer Congress*, Dublin (Ireland), September 1986. North Holland.

[Jaf94] Aubrey Jaffer. *Reference Manual for scm*, 1994.

[JF92] Stanley Jefferson and Daniel P Friedman. A simple reflective interpreter. In Yonezawa and Smith [YS92], pages 48–58.

[JGS93] Neil D Jones, C Karsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993. With chapters by L.O. Andersen and T. Mogensen.

[Kah87] Gilles Kahn. Natural semantics. In *Proc. of STACS 1987, Lecture Notes in Computer Science*, chapter 247. Springer–Verlag, March 1987.

[Kam90] Samuel Kamin. *Programming Languages : an Interpreter-Based Approach*. Addison-Wesley, Reading, Mass., 1990.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1) :7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[KdRB92] Gregor Kiczales, Jim des Rivières, and Daniel G Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge MA, 1992.

[Kes88] Robert R. Kessler. *Lisp, Objects, and Symbolic Programming*. Scott, Foreman/Little, Brown College Division, Glenview, Illinois, 1988.

[KFFD86] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. *Symposium on LISP and Functional Programming*, pages 151–161, August 1986.

[KH89] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *POPL'89 — 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 281–292, Austin (Texas, USA), January 1989. ACM Press.

[Knu84] Donald Ervin Knuth. *The TEX Book*. Addison Wesley, 1984.

[KR78] Brian W Kernighan and Dennis M Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[KR90] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in pcl. In *LFP '90 – ACM Symposium on Lisp and Functional Programming*, pages 99–1052, Nice (France), June 1990.

[KW90] Morry Katz and Daniel Weise. Continuing into the future : on the interaction of futures and first-class continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[Lak80] Fred H. Lakin. Computing with text-graphic forms. In *Conference Record of the 1980 Lisp Conference*, pages 100–105, 1980.

[Lan65] P J Landin. A correspondence between algol 60 and church's lambda-notation. *Communications of the ACM*, 8 :89–101 and 158–165, February 1965.

[Leb05] Maurice Leblanc. *813*. Éditions Pierre Lafitte, 1905.

[LF88] Henry Lieberman and Christopher Fry. Common eval. *Lisp Pointers*, 2(1) :23–33, July 1988.

[LF93] Shinn-Der Lee and Daniel P Friedman. Quasi-static scoping : Sharing variable bindings across multiple lexical scopes. In *POPL '93 – Twentieth Annual ACM symposium on Principles of Programming Languages*, pages 479–492, Charleston (South Carolina, USA), January 1993. ACM Press.

[lfp80] *Conference Record of the 1980 LISP Conference*, Stanford (California USA), August 1980. The LISP Conference.

[lfp94] *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, Orlando (Florida USA), June 1994. ACM Press.

[Lie87] Henry Lieberman. Reversible object-oriented interpreters. In *ECOOP '87 – European Conference on Object-Oriented Programming*, volume Special issue of Bigre 54, pages 13–22, Paris (France), June 1987. AFCET.

[LLSt93] Bil Lewis, Dan LaLiberte, Richard Stallman, and the GNU Manual Group. Gnu emacs Lisp reference manual. Technical report, Free Software Foundation, 675 Massachusetts Avenue, Cambridge MA 02139 USA, Edition 2.0 1993.

[LP86] Kevin J. Lang and Barak A. Pearlmutter. Oaklisp : an object-oriented Scheme with first class types. In *ACM Conference on Object-Oriented Systems, Programming, Languages and Applications*, pages 30–37, September 1986.

[LP88] Kevin J. Lang and Barak A. Pearlmutter. Oaklisp : an object-oriented dialect of Scheme. *International journal on Lisp and Symbolic Computation*, 1(1) :39–51, May 1988.

[LW93] Xavier Leroy and Pierre Weis. *Manuel de référence du langage Caml*. InterÉditions, 1993.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[MAE+62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. Lisp 1.5 programmer's manual. Technical report, MIT Press, Cambridge, MA (USA), 1962.

[Man74] Zohar Manna. *Mathematical Theory of Computation*. Computer Science Series. McGraw-Hill, 1974.

[Mas86] Ian A Mason. *The Semantics of Destructive Lisp*. CSLI Lecture Notes, Leland Stanford Junior University, California USA, 1986.

[Mat92] Luis Mateu. Efficient implementation of coroutines. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 230–247, Saint-Malo (France), September 1992. Springer-Verlag.

[MB93] Luis Mateu-Brule. *Stratégies avancées de gestion de blocs de contrôle*. Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris 6), Paris (France), February 1993.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine – part i. *Communications of the ACM*, 3(1) :184–195, 1960.

[McC78a] John McCarthy. Lisp history. In *Proc. SIGPLAN History of Programming Languages Conference*, pages 217–223, 1978. Also Sigplan Notices 13(8).

[McC78b] John McCarthy. A micro-manual for Lisp - not the whole truth. In *Proc. SIGPLAN History of Programming Languages Conference*, pages 215–216, 1978. Also Sigplan Notices 13(8).

[McD93] Raymond C McDowell. The relatedness and comparative utility of various approaches to operational semantics. Technical Report MS-CIS-93-16, LINC LAB 246, University of Pennsylvania, February 1993.

[MFG+04] Jacob Matthews, Robert Bruce Findler, Paul T. Graunke, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring software for the web. *Journal of Automated Software Engineering*, 2004.

[ML95] Vincent Manis and James J Little. *The Schematics of computation*. Prentice-Hall, 1995.

[MNC+89] Gérald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard, and Karl Tombre. *Les langages à objets*. InterÉditions, 1989.

[Mor92] Luc Moreau. An operational semantics for a parallel functional language with continuations. In D. Etiemble and J-C. Syre, editors, *PARLE '92 – Parallel Architectures and Languages Europe*, pages 415–430, Paris (France), June 1992. Lecture Notes in Computer Science 605, Springer-Verlag.

[Mor94] Luc Moreau. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations*. Thèse de docteur en sciences appliquées, Université de Liège (Belgique), avril 1994.

[Mos70] Joel Moses. The function of FUNCTION in LISP. *SIGSAM Bulletin*, 15 :13–27, July 1970.

[Moz87] Wolfgang A Mozart. Don Giovanni, 1787.

[MQ94] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations, a duumvirate of control operators. In Manuel Hermenegildo and Jaan Penjam, editors, *Lecture Notes in Computer Science 844*, pages 182–197, Madrid (Spain), September 1994. International Workshop PLILP '94 – Programming Language : Implementation and Logic Programming, Springer-Verlag.

[MQRS99] Luc Moreau, Christian Queinnec, Daniel Ribbens, and Manuel Serrano. *Recueil de petits problèmes en Scheme*. Scopos. Springer-Verlag, 1999.

[MR91] James S Miller and Guillermo J Rozas. Free variables and first-class environments. *International journal on Lisp and Symbolic Computation*, 4(2) :107–141, 1991.

[MS80] F Lockwood Morris and Jerald S Schwarz. Computing cyclic list structures. In *Conference Record of the 1980 Lisp Conference*, pages 144–153. The Lisp Conference, 1980.

[Mul92] Robert Muller. M-lisp : A representation-independent dialect of lisp with reduction semantics. *ACM Transactions on Programming Languages and Systems*, 14(4) :589–615, October 1992.

[Nei84] Eugen Neidl. *Étude des relations avec l'interprète dans la compilation de Lisp*. Thèse de troisième cycle, Université Paris 6, Paris (France), 1984.

[Nor72] Eric Norman. 1100 Lisp reference manual. Technical report, University of Wisconsin, 1972.

[NQ89] Greg Nuyens and Christian Queinnec. Identifier Semantics : a Matter of References. Technical Report LIX RR 89 02, 67–80, Laboratoire d'Informatique de l'École Polytechnique, May 1989.

[PE92] Julian Padget and Greg Nuyens (Editors). The eulisp definition. Technical report, University of Bath, 1992.

[Per79] Jean-François Perrot. Lisp et λ-calcul. In Bernard Robinet (ed), editor, *λ-calcul et sémantique formelle des langages de programmation*, La Châtre (France), 1979. AFCET-GROPLAN, LITP-ENSTA, Paris.

[Pit80] Kent Pitman. Special forms in Lisp. In *Conference Record of the 1980 LISP Conference* [lfp80], pages 179–187.

[PJ87] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, 1987.

[PNB93] Padget, J.A., Nuyens, G., and Bretthauer, H. An overview of EuLisp. *Lisp and Symbolic Computation*, 6(1/2) :9–98, 1993.

[QC88] Christian Queinnec and Pierre Cointe. An open-ended Data Representation Model for Eu-Lisp. In *LFP '88 – ACM Symposium on Lisp and Functional Programming*, pages 298–308, Snowbird (Utah, USA), 1988.

[QD93] Christian Queinnec and David De Roure. Design of a concurrent and distributed language. In Robert H Halstead Jr and Takayasu Ito, editors, *Parallel Symbolic Computing : Languages, Systems, and Applications, (US/Japan Workshop Proceedings)*, volume Lecture Notes in Computer Science 748, pages 234–259, Boston (Massachusetts, USA), October 1993.

[QD96] Christian Queinnec and David De Roure. Sharing code through first-class environments. In *Proceedings of ICFP'96 — ACM SIGPLAN International Conference on Functional Programming*, pages 251–261, Philadelphia (Pennsylvania, USA), May 1996.

[QG92] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to symbolic pattern matching with intelligent backtrack. In M Billaud, P Castéran, MM Corsini, K Musumbu, and A Rauzy, editors, *WSA '92— Workshop on Static Analysis*, number 81-82 in Revue Bigre+Globule, pages 109–117, Bordeaux (France), September 1992.

[QP90] Christian Queinnec and Julian Padget. A deterministic model for modules and macros. Bath Computing Group Technical Report 90-36, University of Bath, Bath (UK), 1990.

[QP91a] Christian Queinnec and Julian Padget. A proposal for a modular Lisp with macros and dynamic evaluation. In *Journées de Travail sur l'Analyse Statique en Programmation Équationnelle, Fonctionnelle et Logique*, pages 1–8, Bordeaux (France), October 1991. Revue Bigre+Globule 74.

[QP91b] Christian Queinnec and Julian Padget. Modules, macros and Lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123, Santiago (Chile), October 1991. Plenum Publishing Corporation, New York NY (USA).

[QS91] Christian Queinnec and Bernard Serpette. A Dynamic Extent Control Operator for Partial Continuations. In *POPL '91 – Eighteenth Annual ACM symposium on Principles of Programming Languages*, pages 174–184, Orlando (Florida, USA), January 1991.

[Que82] Christian Queinnec. *Langage d'un autre type : Lisp*. Eyrolles, Paris (France), 1982.

[Que89] Christian Queinnec. Lisp – Almost a whole Truth. Technical Report LIX/RR/89/03, 79–106, Laboratoire d'Informatique de l'École Polytechnique, December 1989.

[Que90a] Christian Queinnec. A Framework for Data Aggregates. In Pierre Cointe, Philippe Gautron, and Christian Queinnec, editors, *Actes des JFLA 90 – Journées Francophones des Langages Applicatifs*, pages 21–32, La Rochelle (France), January 1990. Revue Bigre+Globule 69.

[Que90b] Christian Queinnec. *Le filtrage : une application de (et pour) Lisp*. Inter-Éditions, Paris (France), 1990.

[Que90c] Christian Queinnec. PolyScheme : A Semantics for a Concurrent Scheme. In *Workshop on High Performance and Parallel Computing in Lisp*, Twickenham (UK), November 1990. European Conference on Lisp and its Practical Applications.

[Que92a] Christian Queinnec. Compiling syntactically recursive programs. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 5(4) :2–10, October-December 1992.

[Que92b] Christian Queinnec. A concurrent and distributed extension to scheme. In D. Etiemble and J-C. Syre, editors, *PARLE '92 – Parallel Architectures and Languages Europe*, pages 431–446, Paris (France), June 1992. Lecture Notes in Computer Science 605, Springer-Verlag.

[Que93a] Christian Queinnec. Continuation conscious compilation. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 6(1) :2–14, January 1993.

[Que93b] Christian Queinnec. Designing MEROON v3. In Christian Rathke, Jürgen Kopp, Hubertus Hohl, and Harry Bretthauer, editors, *Object-Oriented Programming in Lisp : Languages and Applications. A Report on the ECOOP'93 Workshop*, number 788, Sankt Augustin (Germany), September 1993.

[Que93c] Christian Queinnec. A library of high-level control operators. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 6(4) :11–26, October 1993.

[Que93d] Christian Queinnec. Literate programming from scheme to TEX. Research Report LIX RR 93.05, Laboratoire d'Informatique de l'École Polytechnique, 91128 Palaiseau Cedex, France, November 1993.

[Que94] Christian Queinnec. Locality, causality and continuations. In *LFP '94 – ACM Symposium on Lisp and Functional Programming*, pages 91–102, Orlando (Florida, USA), June 1994. ACM Press.

[Que96] Christian Queinnec. Macroexpansion reflective tower. In Gregor Kiczales, editor, *Proceedings of the Reflection'96 Conference*, pages 93–104, San Francisco (California, USA), April 1996.

[Que04] Christian Queinnec. Continuations and web servers. *Higher-Order and Symbolic Computation*, 17(4) :277–295, December 2004.

[R3R86] Revised^3 report on the algorithmic language scheme. *ACM Sigplan Notices*, 21(12), December 1986.

[RA82] Jonathan A. Rees and Norman I. Adams. T : a dialect of lisp or, lambda : the ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122, 1982.

[Rab55] François Rabelais. *La vie très horrificque du grand Gargantua, père de Pantagruel* Jadis composée par M. Alcofribas, abstracteur de quinte essence. Pléïade, Éditions Gallimard, 1955.

[RAM84] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual, Fourth Edition*. Yale University Computer Science Department, January 1984.

[Ray91] Eric Raymond. *The New Hacker's Dictionary*. MIT Press, Cambridge MA, 1991. With assistance and illustrations by Guy L. Steele Jr.

[Rey72] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[Rib69] Daniel Ribbens. *Programmation non numérique : Lisp 1.5*. Monographies d'Informatique, AFCET, Dunod, Paris, 1969.

[Ric06] Jeffrey Richter. *CLR via C#*. Microsoft Press, 2nd edition edition, 2006.

[RM92] John R Rose and Hans Muller. Integrating the Scheme and c languages. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 247–259, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

[Row90] William Rowan. A Lisp compiler producing compact code. In *Conference Record of the 1980 Lisp Conference* [lfp80], pages 216–222.

[Roz92] Guillermo Juan Rozas. Taming the y operator. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 226–234, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

[Sam79] Hanan Sammet. Deep and shallow binding : The assignment operation. *Computer Languages*, 4 :187–198, 1979.

[Sch86] David A Schmidt. *Denotational Semantics, a Methodology for Language Development*. Allyn and Bacon, 1986.

[Sco76] Dana Scott. Data types as lattices. *Siam J. Computing*, 5(3) :522–587, 1976.

[Sén89] Nitsan Séniak. Compilation de Scheme par spécialisation explicite. *Revue Bigre+Globule*, (65) :160–170, July 1989.

[Sén91] Nitsan Séniak. *Théorie et pratique de Sqil, un langage intermédiaire pour la compilation des langages fonctionnels*. Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris 6), Paris (France), October 1991.

[Ser93] Manuel Serrano. De l'utilisation des analyses de flot de contrôle dans la compilation des langages fonctionnels. In Pierre Lescanne, editor, *Actes des journées du GDR de Programmation*, 1993.

[Ser94] Manuel Serrano. *Bigloo User's Manual*, 1994. available on `ftp://ftp.inria.fr/INRIA/Projects/icsla/Implementations`.

[SF89] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming*. MIT Press and McGraw-Hill, 1989.

[SF92] Amr Sabry and Matthias Felleisen. Reasoning about continuation-passing style programs. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 288–298, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

[SG93] Guy L. Steele, Jr. and Richard P Gabriel. The evolution of Lisp. In *The Second ACM SIGPLAN History of Proramming Languages Conference (HOPL-II)*, pages 231–270, Cambridge (Massachusetts, USA), April 1993. ACM SIGPLAN Notices 8, 3.

[Shi91] Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation*. The MIT Press, Cambridge, MASS, 1991.

[SJ87] Emmanuel Saint-James. *De la Méta-Récursivité comme Outil d'Implémentation*. Thèse d'état, Université Paris VI, December 1987.

[SJ93] Emmanuel Saint-James. *La programmation appplicative (de LISP à la machine en passant par le lambda-calcul)*. Hermès, 1993.

[Sla61] J R Slagle. *A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculees, Symbolic Automatic Integration (SAINT)*. PhD thesis, MIT, Lincoln Lab, 1961.

[Spi90] Éric Spir. *Gestion dynamique de la mémoire dans les langages de programmation, application à Lisp*. Science informatique. InterÉditions, Paris (France), 1990.

[SS75] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme : an interpreter for extended lambda calculus. MIT AI Memo 349, Massachusetts Institute of Technology, Cambridge, Mass., December 1975.

[SS78a] Guy Lewis Steele Jr. and Gerald Jay Sussman. The art of the interpreter, or the modularity complex (parts zero, one, and two). MIT AI Memo 453, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[SS78b] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of lisp. MIT AI Memo 452, Massachusetts Institute of Technology, Cambridge, Mass., January 1978.

[SS80] Guy Lewis Steele Jr. and Gerald Jay Sussman. The dream of a lifetime : a lazy variable extent mechanism. In *Conference Record of the 1980 Lisp Conference*, pages 163–172. The Lisp Conference, 1980.

[Ste78] Guy Lewis Steele Jr. Rabbit : a compiler for Scheme. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[Ste84] Guy L. Steele, Jr. *Common Lisp, the Language*. Digital Press, Burlington MA (USA), 1984.

[Ste90] Guy L. Steele, Jr. *Common Lisp, the Language*. Digital Press, Burlington MA (USA), 2nd edition, 1990.

[Sto77] Joseph E Stoy. *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge Massachusetts USA, 1977.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986. ou "Le langage C++", InterÉditions 1989.

[SW94] Manuel Serrano and Pierre Weis. 1+1=1 : An optimizing caml compiler. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 101–111, Orlando (Florida, USA), June 1994. INRIA RR 2265.

[Tak88] M. Takeichi. Lambda-hoisting : A transformation technique for fully lazy evaluation of functional programs. *New Generation Computing*, (5) :377–391, 1988.

[Tei74] Warren Teitelman. *InterLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto (California, USA), 1974.

[Tei76] Warren Teitelman. Clisp : Conversational LISP. *IEEE Trans. on Computers*, C-25(4) :354–357, April 1976.

[VH94] Jan Vitek and R Nigel Horspool. Taming message passing : Efficient method look-up for dynamically typed languages. In *ECOOP '94 — 8th European Conference on Object-Oriented Programming*, Bologna (Italy), 1994.

[Wad88] Philip Wadler. Deforestation : Transforming programs to eliminate trees. In H Ganzinger, editor, *ESOP '88 – European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358, 1988.

[Wan80a] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28. The Lisp Conference, 1980.

[Wan80b] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1) :164–180, 1980.

[Wan84] Mitchell Wand. A semantic prototyping system. In *Proceedings ACM SIGPLAN '84 Compiler Construction Conference*, pages 213–221, 1984.

[Wan86] Mitchell Wand. The mystery of the tower revealed : a non-reflective description of the reflective tower. In *Proceedings of the 1986 ACM Symposium on LISP and Functional Programming*, pages 298–307, August 1986.

[Wat93] Richard C Waters. Macroexpand-all : An example of a simple Lisp code walker. *Lisp Pointers, ACM SIGPLAN Special Interest Publication on Lisp*, 6(1) :25–32, January 1993.

[WC94] Andrew K Wright and Robert Cartwright. A practical soft typing system for Scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programmming* [lfp94], pages 250–262.

[WH88] Patrick H Winston and Berthold K Horn. *Lisp*. Addison Wesley, third edition, 1988.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[WS90] Larry Wall and Randal L Schwartz. *Programming perl*. O'Reilly & Associates, Inc., 1990.

[WS94] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *POPL'94 — 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 435–445, Portland (Oregon, USA), January 1994. ACM Press.

[YH85] Taiichi Yuasa and Masami Hagiya. Kyoto common Lisp report. Technical report, Kyoto University, Japan, 1985.

[YS92] Akinori Yonezawa and Brian C. Smith, editors. *Reflection and Meta-Level Architecture : Proceedings of the International Workshop on New Models for Software Architecture '92*, Tokyo, Japan, November 1992. Research Institute of Software Engineering (RISE) and Information-Technology Promotion Agency, Japan (IPA), in cooperation with ACM SIGPLAN, JSSST, IPSJ.

Index

#

#. , 136
#1#, 259
#1=, 259
#*n*#, 137
#*n*=, 137
$\mathcal{B}$, 155
$\mathcal{L}$, 153
&aux, 13
&key, 13
&rest, 13
&whole, 304
(), 375
*, 129
*
    en COMMON LISP, 34
active-catchers, 72
classes, 397
class-number, 397
dynamic-variables, 242
dynenv, 443
env, 198
generics, 398
last-defined-class, 403
macros, 306
maximal-fixnum, 358
maximal-number-of-classes, 397
minimal-fixnum, 358
shared-memo-quotations, 133
special-form-keywords, 333
stack, 216
stack-index, 216
starting-offset, 400
+, 25, 366
+
    en COMMON LISP, 34
+let, 447
".scm", 249
".so", 249
<, 25, 428
<=, 129
=, 366
==, 290
->C, 361–365, 367
->CPS, 385, 386
->Class, 398
->Generic, 398
->Scheme, 448
??, 290

A

a.out, 305
abréviation, 295
abstraction, 11
accès aux champs, 413
acons, 325
activation-frame, 177
activation-frame, 177
active-catchers, 73
add-method!, 282
adjoin-definition !, 351
adjoin-free-variable !, 348
adjoin-global-variable !, 194
adjoin-temporary-variables !, 353
adjonction d'un registre, 241
adresse, 109, 178
affectation, 10, 105
    sur variable close, 106
    sur variable locale, 106
    sur variable partagée, 106
    sur variable prédéfinie, 114
A-liste, 12, 270
allocate, 150
allocate, 126
allocate, 126
allocate-Class, 418
allocate-Class, 418
ALLOCATE-DOTTED-FRAME, 207

ALLOCATE-FRAME, 206, 231
ALLOCATE-FRAME1, 231
allocate-immutable-pair, 136
`allocate-list`, 128
allocate-list, 128
`allocate-pair`, 128
allocate-pair, 128
`allocate-Poly-Field`, 418
allocate-Poly-Field, 418
`allocate-Polygon`, 407
allocateur, 400
    avec initialisation, 411
    sans initialisation, 408
α, 144
α-conversion, 21, 352
alphatization, 352
already-seen-value ?, 357
ALTERNATIVE, 204, 219
Alternative, 328
alternative, 8
`amb`, 157
analyse
    d'initialisation, 393
    syntaxique, 258
and, 451
annotations, 343
apostrophe, 296
appel
    par nécessité, 143
    par nom, 143
    par valeur, 10, 142
    terminal, 197
Application, 328
application, 6, 190
application fonctionnelle, 6, 11
applications autonomes, 269
`apply`, 314, 376, 379, 381, 389
apply, 209, 429, 435, 439
apply-cont, 88
arbre
    d'héritage, 402
    syntaxique, 354
`*arg1*`, 217
`*arg2*`, 217
argument discriminant, 419
argument-cont, 88
Arguments, 328
arguments->C, 365
arguments->CPS, 386
`arguments_`, 377
arité des fonctions, 227
ARITY=?, 232
ARITY=?2, 232
arpenteur de code, 303, 324, 343
ASCII, 144
assoc/de, 48
`atom?`, 4
auto-amorçage, 405, 418
    de MEROON, 319
auto-description
    en MEROONET, 396
auto-interprétation, 26, 292
autoquote, 5
`autre-fact`, 60

B

backquote, 317, 323
bar, 25
base-error-handler, 246
`begin`, xviii, 9
begin, 291
begin-cont, 86
`behavior`, 125
β-réduction, 142
better-map, 75
between-parentheses, 361
Bezout, 98
bezout, 98
bibliothèque, 23
    d'exécution, 309, 377
    d'expansion, 309
Bigloo, 226, 342, 393
bignums, 230
bind/de, 48
`bind-exit`, 95, 237
bindings->C, 365
blanc souligné, 357
bloc d'activation, 177
    allocation, 181, 212
`block`, 72, 237
block, 76
block-cont, 92
block-env, 92
block-lookup, 93, 94
boîte, 108, 344
    introduction, 353
`boolean?`, xviii
boolean->C, 363

booléen
de C, 363, 374
en λ-calcul, 147
`boolify`, 124
bottom-cont, 90
boucle d'interaction, 112
compilante, 187
fondamentale, 297
Box-Creation, 344
boxify-mutable-variables, 345
Box-Read, 344
box-ref, 108
box-set !, 108
Box-Write, 344
build-application, 250
build-application-renaming-variables, 445
bury-r, 273
`Bus error, core dumped`, 409
bytecode, 214

C

C, 341
C K&R, 393
C++, 393
`cadr`, xviii, 114
calcul bouclant
migration de, 176
call/cc, 90, 187, 209, 237, 429, 436, 439
CALL0, 208
CALL1, 221
CALL1-car, 231
CALL1-cdr, 231
CALL2, 221
CALL3, 208, 217, 221
`call-arguments-limit`, 381
`call/cc`, 182, 235, 382, 384, 429
`call/ep`, 238, 382
`call-next-method`, 423
`call-with-exit-procedure`, 238
Caml Light, 215, 296
`can-access?`, 110
capture, 326
capture-the-environment, 446
`car`, xviii, 130, 378
car, 25, 35, 89, 130, 187, 366
`careless-Class-fields`, 406
careless-Class-fields, 406
`careless-Class-name`, 406
careless-Class-name, 406
`careless-Class-number`, 406
careless-Class-number, 406
`careless-Class-superclass`, 406
careless-Class-superclass, 406
`careless-Field-defining-class-number`, 406
careless-Field-defining-class-number, 406
`careless-Field-name`, 406
careless-Field-name, 406
`case`, xviii, 226
`catch`, 70
avec `unwind-protect`, 80
coût de, 73
simulé par `block`, 72
simulé variable dynamique, 74
catch, 72, 74, 80
catch-cont, 91
catch-lookup, 91
`cc`, 299, 368
cc, 327, 437
`cdr`, xviii
`cdr` avant `car`, 127
`cerror`, 243
chap3j-interpreter, 435
chapter1d-scheme, 429
chapter1-scheme, 26
chapter3-interpreter, 90
`chapter4-interpreter`, 131
chapter4-interpreter, 131
chapter61-interpreter, 187
chapter62-interpreter, 198
chapter63-interpreter, 209
chapter7d-interpreter, 234
`char?`, xviii
check-byte, 228
`check-class-membership`, 408
check-class-membership, 408
`check-conflicting-name`, 417
check-conflicting-name, 417
CHECKED-DEEP-REF, 277
`checked-global`, 275
CHECKED-GLOBAL-REF-, 441
CHECKED-GLOBAL-REF, 203, 229
CHECKED-GLOBAL-REF+, 441
CHECKED-GLOBAL-REF-code, 251
checked-r-extend*, 276
`check-index-range`, 414

check-index-range, 414
`check-signature-compatibility`, 424
check-signature-compatibility, 424
cheveu sur la langue, 26
Church-Rosser, 142
circularité, 397
citation, 7, 133
    compilation, 357
    compositionnelle, 133
    de cycle, 137
    description, 133
    implicite ou explicite, 7
    mémorisante, 133
    partage, 135
citoyen de première classe, 30
cl.lookup, 45
`->Class`, 398
`Class?`, 418
class `Polygon`, 399
Class ?, 418
Class-class, 405
`Class-class`, 405
classe, 82
    `Class`, 404
    `ColoredPolygon`, 399, 401
    `CountedPoint`, 451
    `Field`, 405
    `Generic`, 420
    `Mono-Field`, 405
    `Object`, 404
    `Point`, 398
    `Poly-Field`, 405
`*classes*`, 397
Class-generate-related-names, 407
`Class-generate-related-names`, 407
Class-initialize !, 416
`Class-initialize!`, 416
Class-name, 418
`Class-name`, 418
Class-number, 418
`Class-number`, 418
`*class-number*`, 397
cleanup forms, 80
CLICC, 342
clone, 387
CLOS, 82, 409
`closure`, 107
closure, 202
Closure-Creation, 350
closurize-main !, 351
Cname-clash ?, 356
codage, 4
`*code*`, 250
codéfinition, 114
code-prologue, 234, 246
collecte
    citations, 348, 354
    fonctions, 354
    variables temporaires, 354
collect-temporaries !, 352, 353
combinaison, 11
combinateur, 175, 202
    de point fixe, 59
    K, 15
    paradoxal **Y**, 59
Common Loops, 82, 419
communications cachées, 298
comparaison
    $Lisp_1$/$Lisp_2$, 32
    physique, 130
compatibilité des méthodes, 420
compatible-arity ?, 440
compilation, 213
    des macros, 316
    indépendante, 297
    séparée, 248, 297, 355, 403
    vers C, 342
compile->C, 354, 384, 449
compile-and-evaluate, 274
compile-and-run, 264
`compile-file`, 297, 311
compile-file, 248
compile-on-the-fly, 264
`compiler-let`, 314
COMPILE-RUN, 264
complexité d'`is-a?`, 408
compteur ordinal, 218
compute-another-Cname, 356
compute-Cname, 356
`compute-field-offset`, 414
compute-field-offset, 414
`compute-kind`, 282
compute-kind, 183, 194, 273
concordance des temps, 402
`cond`, xviii
conditionneur de programme, 195
conflit de noms, 21
    en $Lisp_2$, 39
confusion
    valeur/programme, 263

`cons`, xviii, 130, 175, 377
cons, 25, 35, 89, 130, 187, 366
CONS-ARGUMENT, 207, 221
conséquences inconnues, 409
consommation en mémoire, 176
CONSTANT, 203, 215, 230
Constant, 328
CONSTANT0, 230
CONSTANT-1, 230
CONSTANT-code, 251
`*constants*`, 229, 251
contiguïté de valeurs, 396, 399
Continuation, 384
continuation, 85, 237
continuation, 67, 69
    constante, 98, 198
    flottante, 81
    partielle, 100
    retour multiple, 181
    Web, 102
Continuation Passing Style, 169
convert2arguments, 330
convert2Regular-Application, 385
count, 338
CountedPoint, 451
CountingClass, 450
`CountingClass`, 450
`cpp`, 299, 305
CPS, 169
    naïve, 386
cps, 170
cps-abstraction, 171
cps-application, 171
cps-begin, 170
cps-fact, 437
cps-if, 170
cpsify, 384
cps-quote, 170
cps-set !, 170
cps-terms, 171
CREATE-1ST-CLASS-ENV, 273
`create-boolean`, 127
create-boolean, 127
CREATE-CLOSURE, 220, 232
create-evaluator, 335
create-first-class-environment, 273
`create-function`, 127
create-function, 127
create-immutable-pair, 136
`create-number`, 127
create-number, 127
create-pair, 126
CREATE-PSEUDO-ENV, 282
create-pseudo-environment, 282
`create-symbol`, 127
create-symbol, 127
création de variables globales, 265
crochets sémantiques, 145
`crypt`, 111
crypt, 111
curryfication, 303
Cval, 270
C-value ?, 358
`cycle`, 137

D

d.determine !, 441
`d.evaluate`, 16
d.evaluate, 16
`d.invoke`, 16
d.invoke, 16, 20
d.make-closure, 20
`d.make-function`, 16
d.make-function, 20
data coalescence, 135
dd.eprogn, 47
dd.evaluate, 47
dd.evlis, 47
dd.make-function, 47
DEEP-ARGUMENT-REF, 203, 229
DEEP-ARGUMENT-SET !, 204, 221
`deep-fetch`, 178
deep-fetch, 178
`deep-update!`, 178
deep-update !, 178
`defforeignprimitive`, 392
`define`, xix, 51, 315
define, 293
`define-abbreviation`, 301, 302, 306
define-abbreviation, 308, 314
define-alias, 317
`define-class`, 83, 303, 401
    compilée, 403
    état interne, 401
    interprétée, 402
define-class, 402, 403
`define-CountingClass`, 451
define-CountingClass, 451

`define-generic`, 83, 421
    syntaxe, 420
define-generic, 421
`define-handy-method`, 324
define-inline, 323
`define-instruction`, 225
define-instruction-set, 225
`define-macro`, 300
`define-meroonet-macro`, 401
define-meroonet-macro, 320
`define-method`, 83, 423
    variante, 425, 452
define-method, 423, 452
`define-syntax`, 300, 306
définir
    c'est créer, 51
    c'est modifier, 51
`definitial`, 24, 128
definitial, 24, 89, 128
definitial-function, 35
définition
    de classe, 400
    de Lisp en Lisp, 1
    multiple, 114
`defmacro`, 306
déforestation, 438
defpredicate, 428
`defprimitive`, 24, 129
defprimitive, 24, 35, 89, 129, 191, 366
defprimitive2, 208
defprimitive3, 191
defvariable, 194
delay, 168
δ, 160
déplacement, 399
désassemblage, 212
desc.init, 191
descripteur de champ, 398
description-extend !, 191
determine !, 441
`determine-method`, 421
determine-method, 421
df.eprogn, 42
df.evaluate, 42, 45
df.evaluate-application, 42
df.make-function, 42
diamant, 142
différence, 116
`disassemble`, 212
discriminateur programme/donnée, 7
dispatch-table, 420
displacing macro, 305
`display`, xviii
display-cyclic-spine, 44
domaine, 144
donnée
    cyclique, 117
    immédiate, 4
durée de vie, 73
Dylan, 95
`dynamic`, 160
dynamic, 431
`dynamically-changing-evaluation-order?`, 157
`dynamic-let`, 160, 240, 285
dynamic-let, 431
DYNAMIC-POP, 241
DYNAMIC-PUSH, 241
DYNAMIC-PUSH-code, 252
DYNAMIC-REF, 241
DYNAMIC-REF-code, 252
`dynamic-set!`, 285
dynamic-set !, 431
`*dynamic-variables*`, 242
`*dynenv*`, 241
dynenv-tag, 241

E

$\mathcal{E}^+$, 146
échappement, 67
    dynamique, 76
    lexical, 76
    valide, 239
EcoLisp, 342
écrivain de champ, 415
édition de liens, 248
ef, 147
effet de bord, 105, 106, 115
effet mémo-viscéral, 135
`egal`, 117
égalité, 116
    de fonctions, 118
élimination des fonctions imbriquées, 354
empilements/dépilements, 216
empty-begin, 9
`\endgroup`, 113
`enrich`, 275
enrich, 276, 447

ensemble des variables libres, 347
entête standard `scheme.h`, 355
entier
    31 bits, 371
    petit, 371
enumerate, 106, 448
enumerate-aux, 448
`*env*`, 220
env.global, 24
`env.init`, 13
env.init, 13
Envir, 447
Environment, 332
environment, 85, 177
environnement, 3
    à *display*, 193
    automatiquement extensible, 165
    chaîné, 178
    de première classe, 272
    des échappements lexicaux, 72
    en côtelettes, 178
    enrichi, 275
    fonctionnel, 31
    global, 15, 23, 51, 162
        automatiquement extensible, 112
        figé, 111
        hyperstatique, 113, 165
        universel, 110
    initial, 13, 128
    plat, 194, 342
    réifié, 272
    représentation, 12
environnements
    mutuellement exclusifs, 40
envoi de message, 126, 418
`eprogn`, 9
eprogn, 9
EPS, 302
`eq?`, 116
eq ?, 25
`eql`, 117
`equal?`, 117
    coût, 131
`equalp`, 116
`eqv?`, 117, 119, 130
eqv ?, 130
erreur
    dynamique, 259
    statique, 186, 259
`*error*`, 160
`escape`, 237
escape, 238
ESCAPER, 238
escape-tag, 238
escape-valid ?, 239
espace
    de nom, 40
    des fonctions, 41
    des macros, 311
    des variables dynamiques, 41
    des variables lexicales, 41
étalon linguistique, 175
étendre un langage, 295
EuLisp, 95
`eval`, 2, 257, 300, 316, 393
    fonction, 265
    forme spéciale, 263
    interprété, 268
    propriété, 3
eval, 265, 266
eval at top-level, 266
eval/at, 446
eval/b, 274
EVAL/CE, 263
`eval/at`, 266
`eval/b`, 272
`eval/ce`, 266, 320
`eval-in-abbreviation-world`, 312
`evaluate`, 4, 7, 121, 258, 333
evaluate, 7, 84, 121, 258, 261
`evaluate-amnesic-if`, 122
evaluate-amnesic-if, 122
`evaluate-application`, 124
evaluate-application, 32, 88, 124
evaluate-application2, 33
evaluate-arguments, 88, 99
`evaluate-begin`, 122
evaluate-begin, 86, 99, 122
evaluate-block, 92
evaluate-catch, 91
evaluate-eval, 261
`evaluate-ftn-lambda`, 125
evaluate-ftn-lambda, 125
`evaluate-if`, 122
evaluate-if, 85, 122
evaluate-immutable-quote, 136
`evaluate-lambda`, 125
evaluate-lambda, 87, 125, 435
evaluate-memo-quote, 133
evaluate-nlambda, 440

`evaluate-or`, 438
evaluate-or, 438
evaluate-quote, 85, 133
evaluate-return-from, 93
`evaluate-set!`, 123, 438
evaluate-set !, 87, 123
evaluate-throw, 91
evaluate-unwind-protect, 94
évaluateur, 2
`evaluate-variable`, 123
evaluate-variable, 86, 123
évaluation, 257
    simultanée, 315
Evaluator, 334
`eval-when`, 312
`even?`, 52, 114, 275
even ?, 63
evfun-cont, 88
`evlis`, 11
evlis, 11, 427
exception, 213, 243
exécutable, 298
existence d'un langage, 26
expand-expression, 313
`expand-store`, 126
expand-store, 126
expanseur, 300–302
    EPS, 302
    classique, 302
Expansion Passing Style, 302
EXPLICIT-CONSTANT, 230
`export`, 272
expression de C, 352
`extend`, 13
extend, 13, 27
EXTEND-ENV, 232
extend-env, 87
extensionnalité, 145
extract !, 350, 351
extract-addresses, 274
extract-things !, 350

F

f, 24, 129, 187
f.eprogn, 31
f.evaluate, 31, 38
f.evaluate-application, 38
f.evlis, 31
f.lookup, 38
f.make-function, 31
`fact`, 51
    avec continuation, 97
    avec `call/cc`, 78
    avec `prog`, 67
fact, 25, 51, 67, 78, 249, 308, 310
fact1, 307, 309
fact2, 68, 309
`Fact-Closure`, 347
factfact, 433
factorial, 307–310
factorielle, 51, 347
features, 314
`*features*`, 315
fenv.global, 35
fermeture, 20, 202
    du pauvre, 128
    par enregistrement, 347
    zéroadique, 351
fetch-byte, 226
FEXPR, 287
fib, 25
`Field?`, 418
Field ?, 418
Field-class, 405
`Field-class`, 405
`Field-define-class`, 405
Field-defining-class, 405
Field-generate-related-names, 413
`field-length`, 416
field-length, 416
`field-value`, 414
field-value, 414
filtrage, 322
find-dynamic-value, 445
`find-expander`, 302
find-global-environment, 332
`find-symbol`
    avec `block`, 72
    avec `call/cc`, 77
    avec `throw`, 70
    naïf, 68
find-symbol, 68, 70, 72, 77
find-variable ?, 332
FINISH, 234
fix, 59
fix2, 432
FIX-CLOSURE, 205, 217, 221
FIX-LET, 207, 216, 221

Fix-Let, 328
fixN, 432
`flambda?`, 290
Flat-Function, 347
Flattened-Program, 350
`flat-variables`, 422
flat-variables, 422
`flet`, 36
`fluid-let`, 82
fonction, 14
    close, 345
    d'accompagnement, 403, 406
    d'accompagnement prédéfinie, 418
    d'arité variable, 187
    générique, 82, 396, 418, 419
        corps par défaut, 422
    imbriquée, 345
    intégrable, 25
    représentation, 14
foo, 25
force, 168
foreign interface, 392
`format`, 355
forme
    de contrôle, 70
    `lambda` en position fonctionnelle, 32
    liante, 64
    normale, 142
    réductible, 188
    spéciale, 6, 295
    spéciale occulte, 325
Forth, 211
frame coalescing, 364
Free Software Foundation, 361
Free-Environment, 347
Free-Reference, 347
FROM-RIGHT-CONS-ARGUMENT, 442
FROM-RIGHT-STORE-ARGUMENT, 442
FTP, xix
full-env, 86
Full-Environment, 332
`*fun*`, 217
`funcall`, 34
funcallable objects, 419
Function, 328
`function`, 34
    fermeture, 20
function, 87
`function_`, 367
Functional-Description, 366
Function-Definition, 350
FUNCTION-GOTO, 232
FUNCTION-INVOKE, 218, 232
function-nadic, 435
function-with-arity, 435
fusion
    de constantes, 135
    des blocs temporaires, 364
Fval, 270

G

`g.current`, 183
g.current-extend !, 183
g.current-initialize !, 184
`g.init`, 183
g.init-extend !, 183
g.init-initialize !, 184
`g.predef`, 333
γ, 162
`gatekeeper`, 111
gatekeeper, 111
gather-cont, 88
gather-temporaries !, 352
GC, 371
    conservatif, 377, 409
`gcc -ansi -pedantic`, 370
`gdb`, 361
generate-arity, 367
generate-closure-structure, 367
generate-C-program, 354
generate-C-value, 358
generate-functions, 367
generate-global-environment, 355
generate-global-variable, 355
generate-header, 355
generate-local-temporaries, 368
generate-main, 369
generate-next-method-functions, 453
generate-pair, 359
generate-possibly-dotted-defini
        tion, 368
generate-quotation-alias, 358
generate-quotations, 357
`generate-related-names`, 451
generate-related-names, 451, 452
generate-symbol, 358
generate-trailer, 355
generate-vector-of-fix-makers, 318

génération de code, 354
`->Generic`, 398
Generic ?, 418
Generic-class, 405
`Generic-class`, 405
Generic-name, 418
`Generic-name`, 418
`*generics*`, 398
`gensym`, 325
get-description, 191
get-dynamic-variable-index, 242
`get-x`, 402
glaçon, 168
Global-Assignment, 328
`global-env`, 291
global-env, 291–293
global-fetch, 184
globalisation, 197
GLOBAL-REF, 203, 229
GLOBAL-REF-code, 251
Global-Reference, 328
GLOBAL-SET !, 204, 215, 221
global-update !, 184
`global-value`, 269
global-value, 444
Global-Variable, 328, 449
global-variable ?, 183
Gnu Emacs Lisp, 18, 256
`go`, 102
GOTO, 219, 230
`goto`, 67
grammaire de Scheme, 258
grec, 144

H

handle-location, 446
héritage
    des champs, 400
    des méthodes, 400
    multiple, 396
hygiène des macros, 325
hyper-fact, 65
hyperstatique, 51, 113

I

identificateurs légaux, 356
identité remarquable, 33
IdScheme->IdC, 356
IF, 147
If, 147
`if`, 8
    arité, 7
if-cont, 85
Ilog Talk, 342
image de fonction générique, 421
immuabilité, 117
immutable-transcode, 136
implantation
    de MEROONET, 397
    de référence, 140
`import`, 282
incredible, 305
index, 399
inian !, 449, 450
initialization-analyze !, 449
`initialize!`, 451
initialize !, 451
inlining, 110, 190, 323, 365
insert-box !, 344, 345
insert-global !, 332
install-code !, 250
`install-macro!`, 307
install-macro !, 306
install-object-file !, 250
`instruction-decode`, 226
`instruction-size`, 226
`int->char`, xviii
intégration, 190, 365
    MEROONET/Scheme, 420
intérêt de la séquence, 10
interprète
    rapide, 175
    réflexif, 287
introspection, 287
    des environnements, 287
invocation
    d'échappement, 239
`invoke`, 14, 345
invoke, 14, 84, 87, 90, 91, 202, 233, 237, 239, 268, 430, 435, 436
INVOKE1, 231
iota, 433
`is-a?`, 408
is-a ?, 408

J

`jmp_buf`, 383
JUMP-FALSE, 219, 230

K

κ, 144
`kar`, 115
kar, 115
Kleene, 399
klop, 65
`kons`, 115
kons, 115
Kronecker, 116
Kyoto COMMON LISP, 342

L

`label`, 53
labeled-cont, 91
`labels`, 53
`lambda`, xviii, 11
    allocateur, 120
lambda, 288
λ-calcul, 59, 139
λ-*drifting*, 176
λ-*hoisting*, 176
`lambda-parameters-limit`, 228
langage, 139
    arborescent, 213
    cible, 341
    d'assemblage, 219
    décrit, 1
    de définition, 1
    du macroexpanseur, 300
    intermédiaire, 202, 214
    paresseux, 55
    purement fonctionnel, 58
    universel, 2
`*last-defined-class*`, 403
`ld`, 250
lecteur de champ, 413
Leibniz (Gottfried Wilhelm), 116
lenteur excécrable, 175
`let*`, xix
`let`, xviii
    créant des liaisons non initialisées, 57
    intérêt, 54
let, 44
let/cc, 95
`let-abbreviation`, 302, 306
`let/cc`, 237
letify, 387
`letrec`, xix, 54, 275
    expansion, 54
`letrec-syntax`, 306
`let-syntax`, 306
liaison, 107
    capture, 107
    dynamique, 159
    immuable, 25
    muable, 25
    non initialisée, 56
    profonde, 22
    superficielle, 22
library, 23
lien d'appartenance, 397
lift !, 347
lift-procedures !, 347, 348
linéarisation, 216
Lisp
    dynamique, 18
    lexical, 18
Lisp$_1$, 30
Lisp$_2$, 30
LiSP2TEX, 167
`list`, xviii, 428
    avec combinateurs, 442
list, 428, 442
liste
    d'association, 12
    de propriétés, 118
    de variables, 13
    en position fonctionnelle, 37
    infinie, 55
listify !, 188
littérature sur Lisp, 1
LLM3, 140
`load`, 255, 297, 299, 316, 393
load, 444
Local-Assignment, 328
Local-Reference, 328
Local-Variable, 328
`local-variable?`, 178
local-variable ?, 178
location, 109
longévité de Lisp, 295

LONG-GOTO, 231
`longjmp`, 382, 384
longueur de champ, 416
`lookup`, 12
lookup, 12, 58, 84, 86, 271, 428
`loop`, 296, 327

M

`m4`, 300
machine
    de Turing, 2
    virtuelle, 140, 175
macro, 295
    à état interne, 401
    composable, 299
    cyclique, 304
    définissant des macros, 317
    en C, 371
    enjolivure, 322
    exportée, 318
    expressivité, 300
    filtrage, 323
    hygiénique, 323
    hyperstatique, 316
    locale, 318
    masque, 322
    ponctuelle, 317
    raccourci, 322
macro-caractère, 136, 296
`macro-eval`, 306, 312, 316
`macroexpand`, 298, 303
macroexpanseur, 298
    monolithique, 299
macroexpansion, 298
    endogène, 298
    exogène, 298
`macrolet`, 303, 306
macrologie, 295, 300
macrosymbole, 301
MacScheme, 232
Magic-Keyword, 329
magie noire, 304
`make-allocator`, 410
make-allocator, 410
make-box, 103, 108
`make-Class`, 418
make-Class, 418
make-code-segment, 234
`make-Fact-Closure`, 347
`make-fix-maker`, 412
make-fix-maker, 412
`make-flambda`, 290
`make-function`, 17
make-function, 268
`make-lengther`, 416
make-lengther, 416
make-macro-environment, 336, 448
`make-maker`, 411
make-maker, 411
`make-named-box`, 119
make-named-box, 119
make-predefined-application-generator, 366
`make-predicate`, 408
make-predicate, 408
`make-reader`, 414
make-reader, 414
`make-string`, xviii
`make-toplevel`, 290
`make-vector`, xviii
`make-writer`, 415
make-writer, 415
`malloc`, 371
mangling, 356
`map`, 21
mark-global-preparation-environment, 332
`*maximal-number-of-classes*`, 397
`meaning`, 179, 258, 263, 282
meaning, 167, 180, 198, 262
meaning*, 182, 200, 206
meaning*-multiple-sequence, 180, 200, 204
meaning*-single-sequence, 180, 200, 204
meaning-abstraction, 167, 176, 188, 205
meaning-alternative, 180, 199, 204
meaning-application, 192, 206
meaning-assignment, 185, 204, 277
meaning-bind-exit, 238
meaning-closed-application, 189
meaning-define, 195
meaning-dotted*, 189, 207
meaning-dotted-abstraction, 188, 205
meaning-dotted-closed-application, 189, 207
meaning-dynamic-let, 240
meaning-dynamic-reference, 240
meaning-eval, 262, 263
meaning-export, 273

meaning-fix-abstraction, 182, 200, 205, 279, 443
meaning-fix-closed-application, 189
meaning-import, 282
meaning-monitor, 244
meaning-no-argument, 182, 200, 206
meaning-no-dotted-argument, 189, 207
meaning-primitive-application, 192, 208
meaning-quotation, 180, 203
meaning-redefine, 443
meaning-reference, 184, 199, 203, 277, 284
meaning-regular-application, 181, 200, 206
meaning-sequence, 180, 200, 204
meaning-some-arguments, 182, 200, 206
meaning-some-dotted-arguments, 189, 207
memo-delay, 169
mémo-fonction, 133
mémoire, 125
memoization, 305
`memq`, 232
MEROON, 395
`meroon-apply`, 322
MEROONET, 301, 395
    introduction, 82
meroon-if, 322
message passing style, 108
meta-fact, 60
métalangage, 293
méta-méthode, 343
méthode, 82
M-expression, 7
migration, 176
min-max, 105
`min-max1`, 437
min-max1, 437
`min-max2`, 437
min-max2, 437
mode *autoquote*, 5, 12
modèle de calcul, 106
modificateur physique, 116
monceau, 96
monde central, 297
mondes multiples, 297
`monitor`, 244
Mono-Field-class, 405
`Mono-Field-class`, 405
mot-clé, 300
    superflu, 296
Mozart (Wolfgang), 67
muabilité des liaisons, 25
multi-méthode, 419

N

naive-endogeneous-macroexpander, 306
name, 106, 107
NARY-CLOSURE, 205, 220
new1-evaluate-set !, 438
new2-evaluate-set !, 438
new-assoc/de, 430
`newline`, xviii
`new-location`, 126
new-location, 126
new-renamed-variable, 353
new-Variable, 384
`next-method?`, 425
NfixN, 433
NfixN2, 433
`NIL`, 375
nil, 24, 129, 187
niveaux d'évaluation, 333
No-Argument, 328
nœud gordien, 405
No-Free, 347
nombre fonctionnel, 37
no-more-argument-cont, 99
no-more-arguments, 88
NON-CONT-ERR, 244, 246
non-initialisation de champ, 409
notation *x*`->`*y*, 5
ν, 144
`NULL`, 363
null-env, 86
`number?`, xviii
number->class, 397
`number->class`, 397
number-of, 330
numéro de classe, 397

O

OakLisp, 419
`Object?`, 400
    plus sûr, 450
`Object`, 83
object->class, 400
Object ?, 400
Object-class, 405

`Object-class`, 405
`object->class`, 400
objectification, 328, 343
objectify, 329
objectify-alternative, 329
objectify-application, 330
objectify-assignment, 332
objectify-free-global-reference, 331, 449
objectify-function, 331
objectify-quotation, 329
objectify-sequence, 329
objectify-symbol, 331
objectify-variables-list, 331
objet, 82
    composite, 116
    du pauvre, 197
    en MEROONET, 397
    fonctionnel, 419
    fonctionnellement applicable, 37
ObjVlisp, 396
octet, 225
`odd?`, 52, 114, 275
odd ?, 63
odd-and-even, 63
ω, 142
OO-lifting, 347
opérateur
    chirurgical, 1
    spécial, 6
ordre
    d'évaluation, 11, 442
    d'initialisation des variables, 55
    droite-gauche, 442
    gauche-droite, 156, 307
    normal, 142
ordre de macroexpansion, 403
other-box-ref, 108
other-box-set !, 108
other-make-box, 108
`other-make-named-box`, 120
other-make-named-box, 120

P

package, 320
PACK-FRAME !, 220
`pair?`, 130
pair ?, 130
`pair?`,, xviii
paire
    par boîte, 115
`pair-eq?`, 438
pair-eq ?, 438
`parse-fields`, 417
parse-fields, 417
`parse-variable-specifications`, 421
parse-variable-specifications, 421
passwd, 111
patte d'oie, 219
`*pc*`, 219
PC-independent code, 221
PC-Scheme, 215
Perl, 227, 300
π, 144
PIC, 221
pile, 216
    d'évaluation, 67
    de C, 383
Planner, 419
P-liste, 270
point fixe, 59
    plus petit, 60
pointeur
    ballant, 341
    fou, 341
Poly-Field-class, 405
`Poly-Field-class`, 405
polymorphe, 150
POP-ARG1, 232
POP-ARG2, 232
pop-dynamic-binding, 241, 443, 445
POP-ESCAPER, 238
pop-exception-handler, 245
POP-FRAME !0, 231
POP-FRAME !, 231
POP-FUNCTION, 218, 232
POP-HANDLER, 244
portée, 19, 20, 64
    des macros, 317
position terminale, 98
pouvoir linguistique, 295
`pp`, 355
PREDEFINED, 203, 229
PREDEFINED0, 229
Predefined-Application, 328
predefined-fetch, 184
Predefined-Reference, 328
Predefined-Variable, 328
prédicat, 400, 407

préfixe `set-`, 415
pré-méthode, 423
préparation, 296
prepare, 298, 300, 446
PRESERVE-ENV, 218, 232
preserve-environment, 233, 443
pretty-printer, 354
primitive, 89
primitives, 171
`procedure?`, xviii
procedure->definition, 280
procedure->environment, 280
`procedure->definition`, 278
`procedure->environment`, 278
process-closed-application, 330
process-nary-closed-application, 330
`prog`, 67, 68
`progn`, 9
Program, 328
program ?, 260
programmation par exception, 69, 243
programme, 111, 139
    prétraité, 195
    représentation, 4
promesse, 169
    exemple, 336
propagation de méthodes, 422
properties, 431
protect-return-cont, 94
protocole d'appel, 176, 197, 217, 218
pseudo-activation-frame, 282
pseudo-bloc d'activation, 283
Pseudo-Variable, 384
PSL, 140
push-dynamic-binding, 241, 443, 445
PUSH-ESCAPER, 238
push-exception-handler, 245
PUSH-HANDLER, 244
PUSH-VALUE, 218, 232
PVM, 140

Q

`qar`, 438
qar, 438
`qdr`, 438
qdr, 438
`qons`, 438
qons, 438
quotation ?, 260
quotation-fetch, 229
Quotation-Variable, 350
`quote`, xviii, 7, 133, 263, 296
quote78, 135
quote79, 135
quote80, 135
quote81, 135
quote82, 135

R

`r.global`, 128
r.global, 128
r.init, 89, 123, 187, 273
random-permutation, 441
rattrapage d'erreur, 243
Ravel (Maurice), 105
`read`, xviii, 132, 134, 258, 296
read-file, 248
réalité virtuelle, 51
receveur, 419
recompilation automatique, 110
récursion, 50
    enveloppée, 99
    locale, 53
    mutuelle, 52
    simple, 51
    terminale, 98, 99
redéfinition de macros, 315
redex administratif, 386
reducible expression, 142
Reference, 328
reference->C, 362
ReflectiveClass, 452
`ReflectiveClass`, 452
REFLECTIVE-FIX-CLOSURE, 279
`reflective-lambda`, 280
réflexivité, 257, 401
`register-class`, 416
register-class, 416
`register-CountingClass`, 451
register-CountingClass, 451
`register-generic`, 422
register-generic, 422
`register-method`, 424
register-method, 424
règle
    d'abréviation, 295

d'évaluation, 142
d'hygiène, 326
régression infinie, 321, 408
Regular-Application, 328
REGULAR-CALL, 206, 217, 218
réification, 328
reified-continuation, 436
reified-environment, 273
relation entre symbole et variable, 5
relocate-constants !, 251
relocate-dynamics !, 252
relocate-globals !, 251
Renamed-Local-Variable, 353
renaming-variables-counter, 353
renommage, 352
`repeat`, 296
repeat1, 447
représentation
des booléens, 8
des programmes, 4
fonction générique, 419
légale, 261
ressemblance, 116
RESTORE-ENV, 218, 232
restore-environment, 233, 443
restore-stack, 235
resume, 84–88, 90–94, 99
`retrieve-named-field`, 413
retrieve-named-field, 413
RETURN, 221, 232
`return`, 68, 368
`return-from`, 72, 237
return-from, 76
return-from-cont, 93
`r-extend*`, 178
r-extend, 332
r-extend*, 178, 273, 332
rez-de-chaussée, 334
ρ, 144
rib cage, 178
rope, 136
`run`, 226, 258
run, 219
run-application, 253
run-clause, 225
run-machine, 234, 246
`RunTime-Primitive`, 333

S

`s.global`, 128
s.global, 128
s.init, 126
s.lookup, 22, 428
s.make-function, 22, 428
s.update !, 22, 428
saut en avant, 219
save-stack, 235
scan-pair, 359
scan-quotations, 357
scan-symbol, 358
`scheme.h`, 371
Scheme->C-names-mapping, 356
`schemelib.c`, 375
Scheme->C, 342
`SCM`, 372
SCM, 393
`SCM_`, 388
`SCM_DefineGlobalVariable`, 355
`SCM_DefinePredefinedFunction-Variable`, 376
`SCM_invoke_continuation`, 388
`SCM_2bool`, 374
`SCM_allocate_box`, 363
`SCM_Car`, 375
`SCM_Cdr`, 375
`SCM_CheckedGlobal`, 362, 375
`SCM_close`, 366, 378
`SCM_CLOSURE_TAG`, 378
`SCM_cons`, 366
`SCM_Content`, 362, 376
`SCM_DeclareFunction`, 368
`SCM_DeclareLocalDottedVariable`, 368
`SCM_DeclareLocalVariable`, 368
`SCM_Define`, 374
`SCM_Define...`, 360
`SCM_DefineClosure`, 367, 376
`SCM_DefinePair`, 360
`SCM_DefineString`, 360
`SCM_DefineSymbol`, 360
`SCM_error`, 379
`SCM_false`, 360, 363
`SCM_Fixnum2int`, 372
`SCM_FixnumP`, 372
`SCM_Free`, 362
`SCM_header`, 372

SCM_Int2fixnum, 360, 372
SCM_invoke, 364, 379
SCM_invoke,, 376
SCM_list, 377
SCM_nil, 360
SCM_nil_object, 375
SCM_object, 372
SCM_Plus, 366
SCM_print, 368
SCM_STACK_HIGHER, 383
SCM_tag, 372
SCM_true, 360
SCM_undefined, 375
SCM_unwrapped_object, 373
SCM_unwrapped_pair, 374
SCM_Wrap, 360
SCMq_, 388
SCMref, 373
search-dynenv-index, 241, 443
search-exception-handlers, 245
sélecteur, 400
    careless-, 406
    générique, 402
self_, 377
sémantique
    algébrique, 141
    axiomatique, 141
    dénotationnelle, 139
    des citations, 132
    du λ-calcul, 151
    naturelle, 141
    opérationnelle, 140
send, 419
send super, 423
sens, 139
SEQUENCE, 204, 216, 221
Sequence, 328
séquence, 9
set!, xviii, 10
set!-cont, 87
set!, 343
set-car!, xviii, 117
set-cdr!, xviii, 117, 130, 378
set-cdr!, 25, 130
set-Class-name!, 418
set-Class-name!, 418
set-Class-subclass-numbers!, 418
set-Class-subclass-numbers!, 418
SET-DEEP-ARGUMENT!, 229
set-difference, 195
(setf symbol-function), 270
(setf symbol-value), 270
set-field-value!, 415
set-field-value!, 415
SET-GLOBAL!-code, 251
SET-GLOBAL!, 229
set-global-value!, 269
setjmp, 382
set-kdr!, 115
set-kdr!, 115
setq, 10
SET-SHALLOW-ARGUMENT!2, 228
SET-SHALLOW-ARGUMENT!, 218, 228
set-variable-value!, 286
set-winner!, 106
sexe de NIL, 8
sg.current, 184
sg.current.names, 441
sg.init, 184
sg.predef, 333
sh, 297, 392
shadowable-fetch, 284
shadow-extend*, 283
shadow-extend*, 283
SHADOW-REF, 284
SHALLOW-ARGUMENT-REF, 203, 226, 228
SHALLOW-ARGUMENT-REF0, 228
SHALLOW-ARGUMENT-REF1, 228
SHALLOW-ARGUMENT-REF2, 228
SHALLOW-ARGUMENT-REF3, 228
SHALLOW-ARGUMENT-SET!, 204, 218
shell, 297
SHORT-GOTO, 231
SHORT-JUMP-FALSE, 231
SHORT-NUMBER, 230
σ, 144
signal-exception, 245
signature pointée, 420
simulation, 116
simultaneous-eval-macroexpander, 315
single-threaded, 160
SIOD, 393
site d'affectation, 109
size_, 377
size-clause, 225
Smalltalk, 258, 398, 418
some-facts, 307
special-begin, 333
special-define-abbreviation, 336

special-eval-in-abbreviation-world, 336
special-extend, 45
special-if, 333
special-lambda, 333
special-let-abbreviation, 336
special-quote, 333
special-set !, 333
special-with-aliases, 337
Sqil, 226
sr.init, 187
`sr-extend*`, 177
`sr-extend`, 333
sr-extend*, 177
stack-pop, 216
stack-push, 216
stammer, 291
stand-alone-producer, 195, 441
stand-alone-producer7d, 234
`*starting-offset*`, 400
static, 113
`static-wrong`, 186
static-wrong, 186
statique et dynamique, 179
statut des constantes, 24
STORE-ARGUMENT, 206, 216, 221
string, 136
`string?`, xviii
`string`, xviii
`string-ref`, xviii
`string-set!`, xviii
`string->symbol`, xviii, 270
structure grammaticalement molle, 302
suboptimal, 419
substituabilité, 116
substitution, 142
suffixe `-set!`, 415
super-méthode, 423
suspension, 168
`symbol?`, xviii
`symbol-concatenate`, 406
symbol-concatenate, 406
symbole, 118, 270
    propriété, 118
`symbol-function`, 270
`symbol-value`, 270
`symbol->variable`, 4
syntaxe
    des booléens, 24
    des données, 258
    essentielle, 339
    externe, 305
syntaxiquement récursif, 259
`syntax-rules`, 327
`system`, 392
system, 392
système à objets, 395

T

t, 24, 129, 187
table d'aiguillage, 420
`tagbody`, 102
taille, 401
TEΛOΣ, 82
test de type, 190
TEX, 18, 113
`the-empty-list`, 126
the-empty-list, 126
`the-environment`, 288
the-environment, 291, 446
`the-false-value`, 8
the-false-value, 8
the-non-initialized-marker, 58
thèse de Church, 2
`thing`, 360
threaded code, 211
`throw`, 70
throw, 72, 74, 437
throw-cont, 91
throwing-cont, 91
THUNK-CLOSURE, 443
thunks, 168
toplevel, 297
`toplevel`, 291
tour réflexive, 287
`trace-procedure1`, 278
`transcode`, 132
transcode, 132
`transcode-back`, 132, 262
transcode-back, 132
transformation
    des citations, 134
    en boîtes, 108
transparence référentielle, 21, 106
`tree-equal`, 116
tree-shaker, 310
TR-FIX-LET, 207, 221
trou noir, 79
TR-REGULAR-CALL, 206, 221

type, 372, 399
de `fix`, 63
`typecase`, 325

U

`#<UFO>`, 13, 56
unbound variable, 110
undefined, 447
underscore, 357
`#<uninitialized>`, 56, 409
UNLINK-ENV, 232
unordered, 441
`#<unspecified>`, 10
`until`, 296
unwind, 93, 94
unwind-cont, 94
`unwind-protect`, 79, 240
unwind-protect-cont, 94
`upate`, 123
`update!`, 12
`update*`, 123
update !, 12, 84, 87, 271
update, 123
update*, 123
`update-generics`, 422
update-generics, 422
update-walk !, 343

V

`va_list`, 381
`*val*`, 215
valeur
de l'affectation, 13
intruse, 305
multiple, 97
valuation, 141
value, 85
Variable, 328
variable
accrochage de, 21
discriminante, 420
flottante, 283
globale, 355
globale modifiable, 355
globale prédéfinie, 355
libre, 3
liée, 3
locale modifiable, 344
muable, 109
muette, 21
non initialisée, 441
partagée, 107
spéciale, 21
variable->C, 362
variable->Scheme, 448
`variable-defined?`, 286, 293
variable-defined ?, 286, 447
variable-env, 86
variables-list ?, 260
`variable-value`, 286
`variables-list?`, 293
variable-value, 286, 447
variable-value-lookup, 284
VDM, 140
vecteur des méthodes, 420
`vector?`, xviii
`vector`, xviii, 175
vector-copy !, 235
`vector-ref`, xviii
`vector-set!`, xviii
vérification
anti-pervers, 397
d'arité, 232
Vlisp, 237
vowel1<=, 135
`vowel1<=`, 135
vowel2<=, 135
`vowel2<=`, 135
vowel3<=, 135
`vowel3<=`, 135
vowel<=, 134
`vowel<=`, 134

W

W, 59
WCL, 342
`when`, 292, 318
when, 292
`while`, 295
while, 304
`winner`, 106
`with-aliases`, 327
`with-quotations-extracted`, 303
With-Temp-Function-Definition, 352
write-result-file, 248

`wrong`, 6, 186
wrong, 146

X

xscheme, 256

Y

Y, 59

Z

zézayer, 26
zozoter, 26