UNIVERSITÉ DE MONTRÉAL

DEUX APPROCHES À L'IMPLANTATION DU LANGAGE SCHEME

PAR

MARC FEELEY

DÉPARTEMENT D'INFORMATIQUE ET DE RECHERCHE OPÉRATIONNELLE

FACULTÉ DES ARTS ET DES SCIENCES

MÉMOIRE PRÉSENTÉ À LA FACULTÉ DES ÉTUDES SUPÉRIEURES
EN VUE DE L'OBTENTION DU GRADE DE
MAÎTRE ÈS SCIENCES (M. Sc.)

MAI 1986

Table des matières

Table des illustrations v

Sommaire vi

Introduction 1

Chapitre I - Le langage Scheme 4

1.1 λ-calcul 5

1.1.1 Syntaxe 6

1.1.2 Règles de réécriture 6
1.1.2.1 α-conversion 6
1.1.2.2 β-conversion 7

1.1.3 Règles d'évaluation d'une expression 8

1.1.4 Programmation en λ-calcul 9

1.2 Scheme 10

1.2.1 Données 11
1.2.1.1 Booléen 12
1.2.1.2 Nombre 12
1.2.1.3 Caractère et chaîne de caractères 12
1.2.1.4 Symbole 13
1.2.1.5 Vecteur 13
1.2.1.6 Liste 13
1.2.1.7 Port 14
1.2.1.8 Fonction 14

1.2.2 Syntaxe 14

1.2.3 Règles d'évaluation des expressions 15
1.2.3.1 Constante 15
1.2.3.2 Variable 15
1.2.3.3 Application 15
1.2.3.4 Formes spéciales 17
1.2.3.4.1 **quote** 18
1.2.3.4.2 **set!** 18
1.2.3.4.3 **if** 18
1.2.3.4.4 **lambda** 19

1.2.4 Macros 21
1.2.4.1 **let** 22
1.2.4.2 **let*** 22
1.2.4.3 **letrec** 22
1.2.4.4 **rec** 23
1.2.4.5 **named-lambda** 23
1.2.4.6 **begin** 24
1.2.4.7 **and** 24
1.2.4.8 **or** 25
1.2.4.9 **cond** 25
1.2.4.10 **case** 26
1.2.4.11 **do** 26
1.2.4.12 **define** 27

1.2.5 Lien avec le λ-calcul 28
1.3 Rétrospective 29

Chapitre II - L'implantation des fermetures 31

2.1 Introduction générale aux femetures 32
2.1.1 Utilité des fermetures 32
2.1.2 Variables 34
2.1.3 Environnements 35
2.1.3.1 Environnements chaînés 36
2.1.3.2 Environnements à base de *display* 37
2.1.4 Implantation classique des fermetures 37
2.1.5 Efficacité 39
2.2 Notre méthode d'implantation des femetures 41
2.2.1 La traduction de λ-expressions 41
2.2.1.1 Méthode de base 41
2.2.1.2 Réduction du temps de création 43
2.2.1.3 Aspect interne de notre méthode 44
2.2.1.4 Traitement de l'affectation 46
2.2.1.5 Algorithme de traduction 47
2.2.2 Analyse de la performance 48
2.2.3 Simulation des fermetures 54

Chapitre III - La génération de code 56

3.1 Notre méthode de génération de code 57
3.2 Fonctions de génération de code 59
3.2.1 Forme spéciale **`quote`** 59
3.2.2 Référence de variable 59
3.2.3 Forme spéciale **`set!`** 60
3.2.4 Forme spéciale **`if`** 60
3.2.5 Application de fonction 61
3.2.6 Forme spéciale **`lambda`** 61
3.3 Exemple complet 62
3.4 Améliorations 64
3.5 Autres applications 68
3.6 Performance 69

Chapitre IV - Un compilateur Scheme 73

4.1 Structure et fonctionnement du compilateur 74
4.1.1 Expansion des macros 75
4.1.2 Traduction des λ-expressions 75

4.1.3 Génération de code 76
4.1.4 Écriture du code généré en assembleur 77
4.1.5 Lacunes du compilateur 78
4.2 Analyse de la performance 79

Conclusion 82

Remerciements 85

Appendice A - Programme de traduction de λ-expressions 86
Appendice B - Interpréteur Scheme écrit en Scheme 91
Appendice C - Compilateur Scheme non-optimisant écrit en Scheme 94
Appendice D - Compilateur Scheme optimisant écrit en Scheme 97
Appendice E - Tests du chapitre III 103
Appendice F - Exemple de génération de code en SIMULA 67 105
Appendice G - Compilateur Scheme optimisant pour MC68000 108
Appendice H - Tests du chapitre IV 128
Appendice I - Exemple de sortie du compilateur 131

Références 134

Table des illustrations

Figure 2.1 - Chaîne d'enregistrements 36

Figure 2.2 - Display et enregistrements 37

Figure 2.3 - Représentation classique d'une fermeture 38

Figure 2.4 - Structure de la pile à l'entrée de la fonction d'interface 44

Figure 2.5 - Structure de la pile à l'entrée de la fonction commune 45

Figure 2.6 - Exemple de fermetures créées par les deux méthodes 52

Table 2.1 - Résultats des tests pour les deux méthodes d'implantation de fermetures .. 52

Figure 3.1 - Exemple de code généré par le compilateur 63

Table 3.1 - Temps d'exécution des fonctions **`fib`**, **`tak`** et **`trier`** 71

Table 4.1 - Résultats des tests de performance du compilateur pour MC68000 80

Sommaire

Le langage Scheme est un dialecte de Lisp simple et homogène qui gagne de la popularité. Ce mémoire porte sur l'implantation efficace de deux aspects importants d'un système Scheme, c'est-à-dire les fermetures et la génération de code. Pour chacun de ces aspects, nous proposons une nouvelle approche d'implantation et la comparons à d'autres méthodes connues.

Notre approche d'implantation de fermetures est fondée sur le principe de β-conversion du λ-calcul. Nous raffinons une méthode simple basée sur cette dernière ce qui nous amène à concevoir les fermetures comme étant un bout de code. Les avantages de cette approche sont discutés et, à l'aide d'une batterie de tests, nous en analysons la performance. Les résultats obtenus indiquent que dans plusieurs situations notre approche est supérieure à l'approche classique.

Nous montrons qu'il est possible d'utiliser les fermetures pour représenter le code généré par un compilateur. Cette approche permet d'écrire un compilateur Scheme totalement en Scheme et de remplacer avantageusement les interpréteurs. De plus, cette approche peut être étendue à d'autres langages tel que les langages orienté-objet. L'intégration de cette approche dans un compilateur optimisant nous a permis d'en mesurer l'efficacité par rapport à d'autres méthodes d'évaluation.

L'implantation d'un système Scheme combinant nos deux approches a été réalisée. À l'aide de tests comparant celui-ci à d'autres systèmes couramment disponibles sur le marché, nous montrons la viabilité d'un système basé sur nos deux approches.

Introduction

Le langage Lisp est un des plus vieux langages de programmation. Il fut inventé au MIT en 1960 par McCarthy dans le cadre de ses recherches en intelligence artificielle [McCa60] [Stoy84]. Aujourd'hui, il est de loin le langage le plus utilisé dans ce domaine [Rich83].

Lisp, contrairement à la majorité des autres langages, n'est pas un langage stable. Depuis ses débuts, il a beaucoup évolué et il en existe maintenant de nombreux dialectes, souvent incompatibles malgré leur forte ressemblance. Dernièrement un regain d'intérêt s'est fait sentir dans le domaine de l'intelligence artificielle et conséquemment en Lisp. Une préoccupation marquée des chercheurs est de créer un dialecte standard de Lisp pour faciliter et unifier les efforts de développement dans le domaine. Il semble que Common Lisp [Stee84] soit le favori de ce côté. Common Lisp est un langage vaste car il combine la plupart des aspects intéressants des autres dialectes de Lisp. De ce fait, il n'est pas facile à maîtriser dans sa totalité et son implantation efficace sur des ordinateurs à architecture conventionnelle est difficile [Broo84].

Le langage Scheme, dialecte de Lisp surnommé à juste titre *Uncommon Lisp*, partage avec Common Lisp cette idée de standardisation. Cependant, l'emphase est mise sur l'élégance, la simplicité et l'homogénéité du langage ce qui en facilite la compréhension et l'implantation. Néanmoins, il n'est pas pour autant simpliste et possède une puissance expressive considérable. En Scheme, il existe peu de constructions spéciales mais celles-ci sont de caractère général. En particulier, les λ-expressions, apparentées aux abstractions de la théorie du λ-calcul [Chur41], y sont un outil fondamental. Elles permettent de créer des fonctions (qu'on nomme aussi fermetures), des abstractions de données et même d'implanter les constructions spéciales que possèdent les autres dialectes de Lisp.

Ce mémoire porte sur l'implantation efficace du langage Scheme. Deux aspects particulièrement importants sont examinés. Le premier consiste en l'implantation des fermetures et le second en la génération de code dans un compilateur pour Scheme. Pour chacun de ces sujets, nous avons développé une méthode efficace. Ces deux sujets sont à première vue indépendants mais, comme nous le verrons, ils se complètent mutuellement car notre méthode de génération de code est basée sur les fermetures et notre méthode de création de fermetures est basée sur l'utilisation d'un compilateur.

Le premier chapitre décrit le langage Scheme tel quel et agit comme introduction aux fermetures. On y montre comment les fermetures peuvent être utilisées pour implanter la plupart

des constructions spéciales de Scheme. On y retrouve aussi une introduction au λ-calcul duquel Scheme est largement inspiré.

Les deux chapitres suivants présentent chacune des deux méthodes que nous avons développées. Le chapitre II, qui porte sur l'implantation efficace des fermetures, débute par une discussion de l'utilité des fermetures et une description d'une méthode classique d'implantation. Nous décrivons ensuite l'idée de base de notre méthode et nous la raffinons, en plusieurs étapes successives, pour en améliorer l'efficacité. Nous procédons alors à l'analyse de la performance de notre méthode en la comparant à la méthode classique précédemment décrite. Celle-ci est appuyée par une batterie de tests effectuée sur les deux méthodes.

Le chapitre III porte sur la génération de code pour un compilateur Scheme. Nous y décrivons notre méthode de génération de code et la comparons à l'interprétation, méthode souvent utilisée pour l'implantation des langages du style de Lisp. Les optimisations possibles à notre méthode sont discutées et implantées dans un compilateur écrit totalement en Scheme. Finalement, une analyse de la performance de notre méthode basée sur des tests est fournie.

Pour terminer, un compilateur Scheme formé de la synthèse de nos deux méthodes est présenté et analysé au chapitre IV. Ce dernier est comparé à divers systèmes Lisp auxquels nous avions accès ce qui permet de juger de la viabilité d'un système Scheme basé sur nos méthodes.

Chapitre I

Le langage Scheme

Le langage de programmation Scheme est un des nombreux dialectes descendant du langage Lisp. Scheme partage de nombreuses caractéristiques avec Lisp. Tous deux sont des langages interactifs, basés sur l'expression et l'application fonctionnelle.

Par ailleurs, les concepteurs de Scheme [Suss75] ont mis l'emphase sur la simplicité et la généralité du langage plutôt que sur la compatibilité envers les autres dialectes de Lisp et sur la diversité des fonctions disponibles. Scheme a des fondements théoriques beaucoup plus solides que Lisp et peut, à juste titre, être considéré comme un Lisp *épuré*.

Scheme tire la plupart de ses concepts de la théorie du λ-calcul. Bien que Lisp soit aussi inspiré de celle-ci, la plupart des dialectes implantent cette théorie incorrectement.

La première section de ce chapitre fait un survol du λ-calcul et de ses aspects pertinents à Scheme. La seconde section décrit le langage Scheme tel quel.

1.1 λ-calcul

Le λ-calcul, inventé par le logicien Alonzo Church en 1941, est un formalisme mathématique qui permet d'étudier les notions de fonction et d'application de fonction [Chur41] [Bare81]. Le λ-calcul s'attaque non seulement au calcul du résultat de fonctions mais aussi à la manipulation de fonctions.

Concrètement, un *programme* λ-calcul est une chaîne de symboles. Il existe certaines règles qui permettent de transformer une chaîne de symboles en une autre qui lui est équivalente. Un calcul est constitué d'une suite de réécritures à l'aide de ces règles. La chaîne obtenue lorsqu'il n'y a plus de transformations applicables est le *résultat* du programme. Au fond, le calcul n'est qu'une série de transformations syntaxiques et, tout au long des transformations, la chaîne de symboles représente toujours un programme sémantiquement équivalent au programme original.

Les quatres prochaines sections décrivent respectivement la syntaxe, les règles de réécritures, les méthodes d'évaluation de programmes λ-calcul et les implications de la programmation en λ-calcul.

1.1.1 Syntaxe

La syntaxe du λ-calcul est très simple et est basée sur l'expression. Tout programme est une expression dont la syntaxe est définie par la grammaire BNF suivante:

<expression> ::= <variable> |
λ <variable> <expression> |
(<expression> <expression> **)**

La catégorie <variable> représente les noms de variables. Pour nos besoins, une variable est dénotée par une lettre de l'alphabet. Une expression constituée uniquement d'une variable est une *référence* de variable.

Une expression débutant par un λ s'appelle une *abstraction*. La variable et l'expression qui suivent le λ dénotent respectivement le *paramètre* et le *corps* de l'abstraction. Intuitivement, une abstraction représente une fonction à un seul paramètre. Une référence de variable est dite *liée* dans une expression si elle se trouve dans le corps d'une abstraction, contenue dans cette expression, qui déclare un paramètre de même nom. Une référence de variable qui n'est pas liée dans une expression est dite *libre* dans cette expression. Par exemple, dans l'expression **λx(xy)**, la référence à **x** est liée, par contre, la référence à **y** est libre.

Une *combinaison* est une expression constituée de deux expressions entre parenthèses. Intuitivement, une combinaison représente une application de fonction. La première expression dénote la fonction (ou opérateur) à appliquer et la seconde dénote l'argument (ou opérande) sur lequel la fonction s'applique.

1.1.2 Règles de réécriture

Jusqu'à présent, nous nous sommes fiés à l'intuition pour décrire la sémantique d'une expression. Le λ-calcul possède deux règles de réécriture pour décrire le calcul: l'α-conversion et la β-conversion. Celles-ci constituent un mécanisme précis de transformation de chaînes qui, bien que fort simple, n'attache pas directement de sémantique à chacune des diverses formes d'expressions et n'indique pas l'ordre dans lequel ces règles doivent être appliquées pour obtenir le résultat.

1.1.2.1 α-conversion

La règle de l'α-conversion stipule que la variable correspondant au paramètre d'une abstraction peut être uniformément remplacée, dans l'abstraction, par une autre variable, à condition que cette autre variable n'apparaisse pas déjà dans le corps de cette abstraction.

Cette règle traduit l'idée des langages de programmation que le choix du nom d'une variable n'est pas important à condition que celui-ci n'entre pas en conflit avec le nom d'autres variables. Ceci garantit que chaque référence de variable désigne toujours la même variable après une α-conversion. Par exemple, la conversion suivante, qui consiste à substituer la variable **x** par la variable **z**, est une α-conversion valide:

$$\lambda\mathbf{x(xy)} \rightarrow_{\alpha} \lambda\mathbf{z(zy)}$$

Par contre, l'α-conversion suivante, qui consiste à substituer la variable **x** par la variable **y**, n'est pas valide car la référence à la variable **y** dans le corps de l'abstraction change de libre à liée:

$$\lambda\mathbf{x(xy)} \rightarrow_{\alpha} \lambda\mathbf{y(yy)}$$

1.1.2.2 β-conversion

La règle de la β-conversion stipule qu'une expression ou sous-expression de la forme **(**λ*vCA***)** peut être remplacée par l'expression *R* (où *R* est l'expression *C* avec toutes les occurences de la variable *v* remplacées par l'expression *A*), à condition que:

a) la variable *v* n'apparaisse pas comme paramètre d'une abstraction dans *C*.

b) les variables libres apparaissant dans *A* n'apparaissent pas comme paramètre d'une abstraction dans *C*.

Cette règle s'apparente aux concepts d'appel de procédure et de référence de variable des langages de programmation traditionnels. Les restrictions sur l'application de la règle de la β-conversion font en sorte qu'une variable réfère au paramètre déclaré dans la plus proche abstraction qui déclare cette variable et qui contient la référence de la variable dans son corps. Ceci correspond aux phénomènes de portée lexicale et de visibilité des identificateurs qu'on retrouve dans certains langages (e.g. Pascal). La deuxième restriction assure que les variables libres dans *A* et dans *C* le seront toujours dans *R* une fois la transformation effectuée. Voici un exemple de β-conversions valides:

$$\mathbf{(}\lambda\mathbf{x(xx)}\lambda\mathbf{yy)} \rightarrow_{\beta} \mathbf{(}\lambda\mathbf{yy}\lambda\mathbf{yy)} \rightarrow_{\beta} \lambda\mathbf{yy}$$

La première consiste à remplacer la variable **x** dans le corps de l'abstraction λ**x(xx)** par λ**yy** et la seconde à remplacer la variable **y** dans le corps de l'abstraction λ**yy** par λ**yy**. Par contre, la

β-conversion suivante n'est pas valide car la variable **y** apparait comme paramètre d'une abstraction dans le corps (i.e. λ**yx**) et elle apparait aussi dans l'argument (i.e. **y**):

$$\mathbf{(\lambda x \lambda y x y)} \rightarrow_\beta \mathbf{\lambda y y}$$

Afin de contourner ces restrictions, il est parfois nécessaire d'appliquer la règle de l'α-conversion quelques fois avant de pouvoir effectuer la β-conversion. Ainsi, l'expression de l'exemple précédent pourrait se réécrire comme suit:

$$\mathbf{(\lambda x \lambda y x y)} \rightarrow_\alpha \mathbf{(\lambda x \lambda z x y)} \rightarrow_\beta \mathbf{\lambda z y}$$

1.1.3 Règles d'évaluation d'une expression

Lorsqu'une expression ne peut plus être réduite à l'aide de la règle de la β-conversion, et qu'aucune séquence d'α-conversions ne rend de β-conversion possible, on dit que l'expression est sous forme *normale*. Évaluer une expression c'est chercher sa forme normale en effectuant une séquence d'α-conversions et de β-conversions. La forme normale obtenue est le résultat de l'évaluation de l'expression. Le théorème de Church-Rosser affirme qu'une même expression réduite par plusieurs séquences différentes d'α-conversions et de β-conversions en plusieurs formes normales résulte en formes normales identiques aux noms des paramètres près [Bare81]. Ceci implique que l'ordre dans lequel s'effectue les conversions n'affecte rien au résultat de l'évaluation d'une expression (sauf, bien sûr, pour le nom des paramètres des abstractions dans le résultat).

Certaines expressions, telle **(λx(xx)λx(xx))**, n'ont pas de formes normales. Quelles que soient les conversions effectuées, il y a toujours une β-conversion qui peut s'appliquer. De façon générale, le fait qu'une expression ait ou non de forme normale est indécidable. Certaines autres expressions ont à la fois des séquences de conversions qui aboutissent à une forme normale et des séquences qui ne terminent pas. Par exemple:

$$\mathbf{(\lambda x \lambda y y (\lambda x (x x) \lambda x (x x)))} \rightarrow_\beta \mathbf{\lambda y y}$$

$$\mathbf{(\lambda x \lambda y y (\lambda x (x x) \lambda x (x x)))} \rightarrow_\beta \mathbf{(\lambda x \lambda y y (\lambda x (x x) \lambda x (x x)))} \rightarrow_\beta \ldots$$

La première évaluation de l'exemple consiste à effectuer une β-conversion sur la combinaison extérieure sans avoir au préalable réduit l'argument à sa forme normale. La deuxième consiste à essayer de réduire l'argument à sa forme normale avant d'effectuer la β-conversion sur la combinaison extérieure. Ces deux méthodes d'évaluation s'appelle respectivement l'évaluation en ordre *normal* et en ordre *applicatif*.

Ces deux méthodes ont leur contrepartie dans certains langages de programmation qui permettent les procédures. L'évaluation en ordre normal correspond au passage de paramètres par *nom* et l'évaluation en ordre applicatif correspond au passage de paramètres par *valeur*. Chaque méthode d'évaluation a ses avantages. En général, l'évaluation en ordre applicatif permet d'obtenir le résultat de l'évaluation d'une expression en effectuant moins de β-conversions que l'évaluation en ordre normal. Par contre, si le paramètre n'est pas utilisé dans le corps de l'abstraction, le nombre de β-conversions à effectuer est moindre pour l'évaluation en ordre normal.

L'évaluation en ordre normal est plus versatile que l'évaluation en ordre applicatif. Il existe un théorème qui affirme que, s'il existe une forme normale pour une expression, l'évaluation en ordre normal terminera [Wegn76]. Ceci n'est pas le cas de l'évaluation en ordre applicatif, tel que montré dans l'exemple précédent.

1.1.4 Programmation en λ-calcul

Le λ-calcul est dans la même classe computationnelle que les machines de Turing. Toute fonction calculable peut être exprimée sous forme d'une expression et le résultat de la fonction, s'il existe, peut être obtenu en évaluant l'expression.

Le fait que le résultat de l'évaluation d'une expression soit lui aussi une expression est un des aspects les plus intéressants du λ-calcul. Puisqu'une expression peut représenter une fonction, rien n'empêche une évaluation de produire une fonction. Il est très commun d'avoir des fonctions qui retournent des fonctions. C'est la seule façon en λ-calcul de faire quoi que ce soit d'utile. En général, si on désire manipuler des valeurs il faut les représenter sous forme de fonctions. La distinction entre une méthode de calcul et une donnée devient ainsi assez floue.

Par exemple, si on désire manipuler des valeurs booléenes, on peut représenter la valeur vraie par l'expression **λaλba** et la valeur faux par l'expression **λaλbb**. L'équivalent de l'instruction conditionnelle **if** *condition* **then** *conséquent* **else** *alternative* des langages de programmation traditionnels s'exprimerait alors par l'expression:

((*condition conséquent***)***alternative***)**

Similairement, on peut représenter les entiers positifs à l'aide de fonctions de la façon suivante:

0	↔	**λaλbb**
1	↔	**λaλb(ab)**
2	↔	**λaλb(a(ab))**
…		

Le nombre N est représenté par une fonction qui, lorsqu'elle est appliquée sur une fonction *F*, retourne une fonction qui calcule la N^ième^ application de *F* sur son argument. Si *X* est une fonction qui calcule le prédécesseur de son argument et que *Y* est une fonction qui représente le nombre N, alors **(***YX***)** est l'équivalent d'une fonction qui calcule N de moins que son argument. En utilisant cette représentation pour les nombres, les opérateurs d'addition, de multiplication et d'exponentiation peuvent aisément s'exprimer comme suit:

$$+ \quad \leftrightarrow \quad \lambda a \lambda b \lambda c \lambda d((ac)((bc)d))$$
$$* \quad \leftrightarrow \quad \lambda a \lambda b \lambda c(a \lambda d((bc)d))$$
$$\wedge \quad \leftrightarrow \quad \lambda a \lambda b(ba)$$

Le calcul de N op M s'exprime par **((***FX***)***Y***)** (où *F* est la fonction qui représente l'opérateur et *X*,*Y* les fonctions qui représentent respectivement les nombres N et M). Par exemple, le calcul de 2^3 se fait comme ceci:

$$((\lambda a \lambda b(ba) \lambda a \lambda b(a(ab))) \lambda a \lambda b(a(a(ab)))) \rightarrow_\beta$$
$$(\lambda b(b \lambda a \lambda b(a(ab))) \lambda a \lambda b(a(a(ab)))) \rightarrow_\alpha$$
$$(\lambda b(b \lambda a \lambda c(a(ac))) \lambda a \lambda b(a(a(ab)))) \rightarrow_\beta$$
$$(\lambda a \lambda b(a(a(ab))) \lambda a \lambda c(a(ac))) \rightarrow_\beta$$
$$\ldots$$
$$\lambda a \lambda b(a(a(a(a(a(a(ab))))))))$$

Les abstractions ont ainsi un usage multiple, que ce soit pour exprimer des méthodes de calcul, pour en créer de nouvelles ou pour représenter des informations. Nous verrons que les λ-expressions de Scheme se comportent de façon similaire aux abstractions. Ceci confère à Scheme les qualités de la portée lexicale et permet d'utiliser des techniques de programmation similaires. La généralité offerte par les λ-expressions est suffisante pour s'en servir presque exclusivement pour implanter les constructions inhérentes au langage Scheme ce qui est décrit dans la prochaine section.

1.2 Scheme

Scheme a été inventé au MIT en 1975 par Sussman et Steele [Suss75]. Ceux-ci l'ont conçu comme une extension du λ-calcul avec une sémantique très claire et simple. Cette première version était basée sur un interprète. En 1978, un compilateur Scheme optimisant fut écrit par Steele [Stee78b]. Le code généré par celui-ci était sous la forme d'un sous-ensemble du langage MacLisp [Pitm83]. Pendant les années qui ont suivi, Scheme a évolué et sa définition a été revisée en 1978 [Stee78a] puis en 1985 [Abel85b] afin de l'épurer encore plus et de le standardiser. Présentement,

les implantation majeures de Scheme sont: MIT Scheme [MITS84], T [Ress82] [Rees84], Scheme 311 [Fess83], Chez Scheme (de l'Université de Caroline du Nord) et Vincennes Scheme (de l'Université Paris-8-Vincennes). Scheme gagne rapidement de la popularité et est utilisé comme langage de support de cours d'introduction à l'informatique dans plusieurs universités (e.g. au MIT [Abel85a], à Université de l'Orégon et à l'Université de l'Indiana).

Scheme est un langage interactif basé sur l'expression. Contrairement aux langages de programmation traditionnels qui nécessitent des phases séparées de composition, compilation et exécution de programmes, l'utilisation du langage Scheme de fait au moyen d'un dialogue (i.e. session) avec le système. Une session est composée d'une série de requêtes entrées par l'usager. Après chaque requête, le système fournit une réponse. Les requêtes sont exprimées sous formes d'expressions et les réponses correspondent au résultat de l'évaluation de ces expressions suivant les règles du langage Scheme. Cette boucle de requêtes-réponses est communément appelée la boucle *read-eval-print* ou *top-level.*

Scheme est un langage de style applicatif. Un calcul s'exprime sous la forme d'une fonction appliquée sur des arguments. Il existe plusieurs fonctions prédéfinies et il est possible d'en définir de nouvelles. De plus, les fonctions sont des objets manipulables comme tout autre type de donnée. Elles peuvent être créées, stockées dans des structures, affectées à des variables, passées en paramètres aux fonctions et retournées comme résultat d'applications de fonctions.

À l'inverse de la majorité des autres dialectes de Lisp, et similairement aux langages comme Algol 60 et Pascal, les fonctions obéissent aux lois de la portée lexicale. Toute évaluation du corps d'une fonction se fait dans le contexte qui existait lorsque la fonction a été créée. Ceci a de fortes conséquences pour un langage où les fonctions sont des objets à part entière, c'est à dire manipulables comme toutes les autres données. Cet aspect est discuté dans la section 1.2.3.4.4.

Les prochaines sections décrivent respectivement les types de données, la syntaxe et les règles d'évaluation des expressions de Scheme.

1.2.1 Données

Scheme possède plusieurs types de donnée: les booléens, les nombres, les caractères, les chaînes de caractères, les symboles, les vecteurs, les listes, les ports et les fonctions. Il existe plusieurs fonctions prédéfinies pour manipuler chacun de ces types de donnée.

Toute donnée a une durée de vie illimitée. Certaines opérations permettent de créer de nouvelles données mais il n'existe pas d'opération explicite pour désallouer l'espace mémoire qu'elles occupent. Généralement, la désallocation des données est effectuée automatiquement

lorsqu'il y a un manque d'espace mémoire. Cette opération, qu'on appelle *garbage collection*, consiste à découvrir puis à désallouer toute donnée qui ne peut plus être accédée par le programme.

Scheme est un langage à type latent. Aucune déclaration de type n'est faite pour les variables, les paramètres et le résultat des fonctions. Les types sont associés aux données plutot qu'aux variables et les variables peuvent contenir des données de n'importe quel type. C'est seulement lors de l'utilisation d'une donnée (e.g. lors de l'exécution d'une fonction prédéfinie) que le type est vérifié.

Chacune des prochaines sections décrit un des types de données et donne des exemples de données de ce type.

1.2.1.1 Booléen

Le type booléen regroupe les données vraie et faux qui sont respectivement représentées par **`#!true`** et **`#!false`**. En général, toute donnée différente de **`#!false`** est traitée comme **`#!true`** par les instructions conditionnelles.

1.2.1.2 Nombre

Le type nombre regroupe plusieurs sous-types: entier, rationnel, réel et complexe. De plus, une donnée de type nombre est soit exacte ou inexacte dépendant de l'exactitude de la représentation interne du nombre. Les sous-types sont complètement compatibles entre eux et il est possible, par exemple, que le produit d'un nombre complexe par un nombre réel donne un nombre entier. La syntaxe des nombres est relativement complexe. Voici un exemple de représentation de nombres:

entier	**`+3`**	**`#B11`**	**`-123456789012345678901`**
rationnel	**`355/113`**	**`1/3`**	**`#B11/22`**
réel	**`3.1415926`**	**`0.333e9`**	**`+1e-10`**
complexe	**`3.14+1.6i`**	**`1/3-1/3i`**	**`1e4@1.57`**

1.2.1.3 Caractère et chaîne de caractères

Une donnée de type chaîne de caractères est une séquence, possiblement nulle, de caractères. Voici un exemple de représentation de caractères et de chaînes de caractères:

caractères	**`#\a`**	**`#\$`**	**`#\newline`**
chaînes de caractères	**`""`**	**`"Z"`**	**`"speciaux: \" et \\"`**

1.2.1.4 Symbole

Une donnée de type symbole est un objet qui a un nom. La seule utilité des symboles provient du fait que deux symboles sont identiques si et seulement si leur noms sont identiques. Les symboles ont une vague ressemblance avec les données de type énuméré de Pascal. La représentation d'un symbole est son nom. Celui-ci est une séquence de caractères quelconques (à l'exception de certains caractères spéciaux tels: **(**,**)**,**'**,**"**,**;**) qui ne peut pas être interprétée comme un autre type de donnée. Par exemple:

```
x    auto    car    +    -1+    symbol->string    set-car!
```

1.2.1.5 Vecteur

Une donnée de type vecteur est une séquence de longueur fixe de données. Un vecteur peut être de longueur nulle et ses éléments ne sont pas nécessairement de même type. Les éléments d'un vecteur sont numérotés, à partir de zéro. Le nombre associé à un élément permet d'indexer le vecteur qui le contient afin de référencer ou de modifier l'élément. Les vecteurs correspondent aux tableaux à une dimension. Les tableaux à plusieurs dimensions peuvent s'obtenir avec des vecteurs contenant des vecteurs. Voici un exemple de représentation de vecteurs:

```
#()    #(ein "deux" #!true)    #(#(11 12) #(21 22))
```

1.2.1.6 Liste

Une donnée de type liste est une séquence de données. Une liste peut être vide et ses éléments ne sont pas nécessairement de même type. Une liste est construite à l'aide de doublets composés de deux champs: le champ **car** contient le premier élément et le champ **cdr** contient la liste qui correspond aux éléments restants. Puisque les doublets d'une liste forment une chaîne, il est possible d'avoir des listes cycliques et plusieurs listes peuvent partager une même queue. Si le champ **cdr** du dernier doublet ne contient pas la liste vide, la liste est dite impropre. Voici quelques exemples de représentation de listes:

#!null ()	les deux représentations de la liste vide
(+ (- 1) 2)	liste contenant le symbole **+**, la liste **(- 1)** et le nombre **2**
(a b c . d)	liste impropre ayant **d** dans le champ **cdr** du dernier doublet

1.2.1.7 Port

Une donnée de type port correspond à une unité d'entrée ou de sortie de caractères (typiquement un fichier de texte). Un port d'entrée fournit des caractères sur demande et un port de sortie accepte des caractères. Il n'existe pas de représentation standard pour les ports.

1.2.1.8 Fonction

Une donnée de type fonction dénote une méthode de calcul qui produit une donnée (i.e. le résultat) lorsqu'elle est appliquée sur d'autres données (i.e. les arguments). Certaines fonctions acceptent un nombre fixe d'arguments, d'autres en acceptent un nombre variable. Les deux opérations principales sur les fonctions sont: la création et l'application (voir les sections 1.2.3.3 et 1.2.3.4.4). Il n'existe pas de représentation standard pour les fonctions.

1.2.2 Syntaxe

Suivant la convention établie par Lisp, les expressions Scheme ont une syntaxe similaire aux données. La grammaire BNF étendue de Scheme est la suivante:

<expression>	::=	<constante> | <variable> | <combinaison>
<constante>	::=	<donnée>
<variable>	::=	<symbole>
<combinaison>	::=	<application> | <forme spéciale>
<application>	::=	**(** <expression> {<expression>} **)**
<forme spéciale>	::=	**(** <mot clé> {<composante syntaxique>} **)**
<mot clé>	::=	**and** | **begin** | **case** | **cond** | **define** | **do** | **if** | **lambda** | **let** | **let*** | **letrec** | **named-lambda** | **or** | **quote** | **rec** | **set!**

La catégorie <donnée> correspond à une donnée telle que décrite dans la section précédente. Une <composante syntaxique> correspond à une donnée avec certaines restrictions qui dépendent du mot clé de la forme spéciale dans laquelle elle apparait. Il est à remarquer que cette grammaire est ambiguë. Les ambiguïtés sont résolues de la façon suivante: une expression qui est ni un symbole ni une liste est une constante et une expression qui est une liste ayant un des mots clés comme premier élément est une forme spéciale.

Il existe également des abréviations qui permettent d'alléger l'écriture d'une expression. Entre autres, une expression préfixée d'une apostrophe (i.e. **'**<expression>) est une abréviation de **(quote** <expression>**)**. Ceci est fort utile pour dénoter une constante quelconque dans une

expression. Une autre abréviation, qu'on nomme notation *backquote*, consiste à préfixer une expression par une apostrophe renversée (i.e. `<expression>). La notation backquote est utile pour abréger les constructions de listes comportant une majorité d'éléments constants. Toute sous-expression qui n'est pas une constante dans la liste à construire est préfixée par une virgule. Par exemple, **`(a (,b d) ,e)** est l'abréviation de **(list 'a (list b 'd) e)**. Finalement, l'utilisation d'un point-virgule dénote un commentaire qui se termine à la fin de la ligne.

1.2.3 Règles d'évaluation des expressions

L'évaluation d'une expression consiste à chercher, d'après un ensemble de règles, la donnée qui lui correspond. La valeur d'une expression c'est la donnée qui résulte de son évaluation. Symboliquement, nous indiquerons que la valeur de l'expression *E* est la donnée *D* par $E \Rightarrow D$. À chaque forme d'expression est associée une règle d'évaluation.

1.2.3.1 Constante

Si *C* est une expression dénotant une constante alors la valeur de *C* est *C*. Par exemple:

3.1416 ⇒ 3.1416

1.2.3.2 Variable

Si *V* est une expression dénotant une variable (i.e. *V* est un symbole) alors la valeur de *V* est la donnée qui est contenue dans l'emplacement présentement associé à la variable *V*. Nous introduisons le concept d'emplacement (au lieu de parler simplement d'associations entre variables et données) car ceci facilite la description de l'affectation. La façon dont s'effectue les associations entre les variables et les emplacements est décrite dans la section 1.2.3.4.4. C'est une erreur si la variable *V* n'est pas associée à un emplacement. Par exemple, si l'emplacement associé à la variable **premiers** contient la liste **(2 3 5 7)** alors:

premiers ⇒ (2 3 5 7)

1.2.3.3 Application

Si *A* est une expression dénotant une application (i.e. *A* est une liste dont le premier élément n'est pas un des mots clés) alors la valeur de *A* est obtenue de la façon suivante:

1) Tous les éléments de *A* sont évalués et ce, dans un ordre indéterminé.

2) La valeur du premier élément doit être une donnée de type fonction. Cette fonction est appelée et les données produites par l'évaluation des éléments restants de *A* lui sont passées comme argument.

3) La donnée produite par la fonction est la valeur de *A*.

Puisque la première expression correspond à la fonction à appliquer, les applications sont exprimées en notation préfixe. Il n'y a pas de particularité d'évaluation des expressions se trouvant en position fonctionnelle d'une application. Ceci simplifie grandement la règle d'évaluation des applications par rapport aux autres dialectes de Lisp qui traitent l'expression en position fonctionnelle de façon spéciale.

De plus, c'est un abus de langage que de parler de la fonction **`sqrt`**. Lorsqu'on écrit **`(sqrt 9)`**, **`sqrt`** dénote une variable dont la valeur est une fonction qui sera appliquée au nombre **`9`**. La valeur initiale (i.e. au début d'une session) de la variable **`sqrt`** est une fonction qui retourne la racine carrée de son argument. Rien n'empêche, au milieu d'une session, d'affecter une nouvelle fonction à la variable **`sqrt`** pour en changer la signification. Voici une liste partielle des variables qui ont initialement une valeur de type fonction (que nous appellerons des *fonctions primitives*):

variable	définition de la fonction primitive
`not`	retourne le complément logique de l'argument
`+`,**`-`**,**`*`**,**`/`**	retourne la somme, différence, produit ou quotient des arguments
`=`,**`<`**,**`>`**,**`<=`**,**`>=`**	indique si la relation numérique est vraie des deux arguments
`eq?`,**`equal?`**	indique si les deux arguments sont identiques ou structurellement équivalents
`list`	retourne une nouvelle liste contenant les arguments
`cons`	retourne un doublet contenant les deux arguments
`car`,**`cdr`**	retourne le contenu du champ **`car`** ou **`cdr`** de l'argument
`set-car!`	remplace le contenu du champ **`car`** du 1er argument par le 2em argument
`read`	lit la représentation d'une donnée d'un port et la retourne
`write`	écrit la représentation d'une donnée sur un port

Par exemple, si l'emplacement associé à la variable **`premiers`** contient la liste **`(2 3 5 7)`** et que les fonctions primitives n'ont pas changées, alors:

```
(+ 1 2 3)                    ⇒ 6
(car premiers)               ⇒ 2
(cons 1 premiers)            ⇒ (1 2 3 5 7)
(list (< 2 1) (- 2 1))       ⇒ (#!false 1)
(= (car (cdr premiers)) 3) ⇒ #!true
```

Certaines fonctions primitives, telles que **`set-car!`** et **`write`**, ne sont utiles que pour l'effet de bord qu'elles produisent et la donnée retournée par celles-ci est indéfinie (i.e. une donnée est retournée mais le langage ne definit pas laquelle). Les particularités de l'application de fonction non-primitive (i.e. définie par l'usager) est décrite dans la section 1.2.3.4.4.

1.2.3.4 Formes spéciales

Vue la similitude entre les expressions et les données Scheme, il est clair que les expressions peuvent être représentées internement sous forme de données. Cette approche a été suivie dans les divers dialectes de Lisp. L'avantage de celle-ci réside dans le fait qu'il est alors possible d'écrire des programmes qui manipulent des représentations de programmes. L'écriture d'interpréteurs et de compilateurs Lisp est d'autant simplifiée puisque la lecture d'une expression se résume à la lecture d'une donnée à l'aide de la fonction primitive **`read`**. Cette donnée peut ensuite être examinée par un interprète pour évaluer l'expression correspondante ou par un compilateur pour la compiler.

De plus, cette approche permet d'implanter les formes spéciales de façon élégante grâce à la méthode de *transformation source à source*. L'idée consiste à implanter un petit sous-ensemble des formes spéciales qui est complet en lui-même puis à définir les autres formes spéciales à l'aide de ce sous-ensemble. Le traitement, par l'interprète ou le compilateur, d'une forme spéciale ne faisant pas partie du sous-ensemble consiste à la traduire en termes du sous-ensemble puis à interpréter ou compiler l'expression résultante. Les formes spéciales implantées de cette façon se nomment *macros* et la traduction de celles-ci se nomme *expansion des macros*. Vue le caractère différent des formes spéciales de base et celles implantées sous forme de macros, nous décrirons ces dernières dans une section séparée.

Les formes spéciales de base que nous utilisons sont celles dénotées par les mots clés: **`quote`**, **`set!`**, **`if`** et **`lambda`**. Celles-ci, combinées avec les constructions de référence de variable et d'application de fonction, forment un sous-ensemble universel du langage Scheme avec lequel les autres formes spéciales peuvent aisément être exprimées.

1.2.3.4.1 **quote**

La forme spéciale **quote** permet de dénoter une donnée constante. La valeur de **(quote** *D***)** est *D*, où *D* est une donnée quelconque incluant les symboles et les listes. Par exemple:

```
(quote premiers)  ⇒ premiers
(quote (+ 1 2 3)) ⇒ (+ 1 2 3)
'(+ 1 2 3)        ⇒ (+ 1 2 3)
```

1.2.3.4.2 **set!**

La forme spéciale **set!** est la construction d'affectation de base. L'évaluation de **(set!** *V E***)**, où *V* est une variable et *E* est une expression, consiste à stocker la valeur de *E* dans l'emplacement associé à la variable *V*. C'est une erreur si la variable *V* n'est pas associée à un emplacement. Cette forme a une valeur indéfinie et n'est utile que pour l'effet de bord qu'elle produit. Par exemple, si un emplacement est associé à la variable **premiers**, alors, après l'évaluation de **(set! premiers '(adam eve))**:

```
premiers ⇒ (adam eve)
```

1.2.3.4.3 **if**

La forme spéciale **if** est la construction conditionnelle de base. La valeur de **(if** *A B C***)**, où *A*, *B* et *C* sont des expressions, est la valeur de *B* si la valeur de *A* est différente de **#!false**, sinon, c'est la valeur de *C*. Il est à noter que seulement une des deux expressions *B* ou *C* est évaluée et que l'expression *A* est évaluée une seule fois.

La forme spéciale **if** existe aussi sous la forme **(if** *A B***)** qui est utilisée uniquement pour son effet de bord. L'expression *B* est évaluée si la valeur de *A* est différente de **#!false**. La valeur de cette forme est indéfinie, quelle que soit la valeur de *A*. Sans perte de généralité, nous ne considérerons plus cette forme de **if** car **(if** *A B***)** peut être réécrit comme **(if** *A B* **'peu-importe)**.

Par exemple, si les fonctions primitives n'ont pas changées:

```
(if (= 1 2) 'adam 'eve)     ⇒ eve
(if 0 (write 1) (write 2)) ⇒ indéfini (écrit 1)
(if (< 1 2) 3)              ⇒ indéfini
```

1.2.3.4.4 **lambda**

La forme spéciale **lambda** permet de créer des données de type fonction. Cette forme, qu'on appelle λ-expression, suit la syntaxe BNF étendue suivante:

<λ-expression> ::= **(** **lambda** <paramètres> <corps> **)**
<paramètres> ::= **(** {<variable>} **)** | <variable> |
(<variable> {<variable>} **.** <variable> **)**
<corps> ::= <expression> {<expression>}

La valeur d'une λ-expression est une fonction qui accepte ses arguments tels que déclarés dans les paramètres et qui retourne le résultat de l'évaluation des expressions de son corps. Lorsqu'elle est appelée, cette fonction crée un nouvel emplacement pour chacune des variables déclarées en paramètre et y stocke des données d'après la règle suivante:

si la fonction est appelée avec M arguments (i.e. $a_1 \ldots a_M$) alors:

paramètre(s)	M doit être	règle
($v_1 \ldots v_N$ **)**	= N	pour $1 \le I \le N$, a_I est stocké dans l'emplacement associé à v_I
v	≥ 0	une nouvelle liste contenant les arguments (i.e. **(** $a_1 \ldots a_M$ **)**) est stockée dans l'emplacement associé à v
($v_1 \ldots v_{N-1}$ **.** v_N **)**	≥ N-1	pour $1 \le I \le N-1$, a_I est stocké dans l'emplacement associé à v_I et une nouvelle liste contenant les arguments restants (i.e. **(** $a_N \ldots a_M$ **)**) est stockée dans l'emplacement associé à v_N

Les deux dernières formes de paramètres permettent de créer des fonctions qui acceptent un nombre variable d'arguments. Dans ces deux cas, on dit que la dernière variable est un paramètre *reste*.

Une fois les emplacements initialisés, les expressions du corps sont évaluées séquentiellement. La donnée retournée par la fonction est la valeur de la dernière expression du corps.

L'ensemble de toutes les associations variable-emplacement à un point d'une expression s'appelle l'environnement à ce point. L'emplacement qui est accédé lors d'une référence de variable ou d'une affectation est déterminé à partir de l'environnement à ce point. L'environnement au top-level s'appelle l'*environnement global* et les variables qui le constituent sont les *variables globales*. L'environnement à l'intérieur du corps d'une fonction est constitué de l'environnement qui existait lors de sa création (i.e. lors de l'évaluation de la λ-expression qui l'a créé) augmenté des associations produites par l'appel de la fonction. Les associations les plus récentes cachent les

associations aux variables de même nom de l'environnement de création en cas de conflit. Le seul endroit où une variable non-globale peut être accédée est dans le corps de la λ-expression qui la déclare en paramètre. Cette stratégie d'association entre les variables et les emplacements correspond à la portée statique (ou lexicale). La majorité des autres dialectes de Lisp, à l'exception de Common Lisp, utilisent la portée dynamique. Cette dernière n'impose aucune limite sur la portée des variables qui peuvent être accédées de n'importe quel point du programme.

Une conséquence de la portée statique est qu'à chaque fonction est associé son environnement de création. Ceci est nécessaire car l'environnement qui existait lors de la création d'une fonction doit toujours être disponible pour évaluer son corps lors de l'appel de cette fonction. De plus, puisque les fonctions ont une durée de vie illimitée, l'environnement d'une fonction ainsi que les emplacements associés aux variables qu'il contient ont aussi une durée de vie illimitée. Ceci n'est pas le cas de la portée dynamique qui limite la durée de vie d'une variable à l'évaluation du corps de la λ-expression qui la déclare en paramètre.

Lorsqu'une λ-expression est évaluée dans un certain environnement, on dit que la fonction résultante est la *fermeture* de la λ-expression dans cet environnement. Par extension, on dit aussi qu'une telle fonction est une fermeture. Nous utiliserons le nom fermeture, plutôt que fonction, pour souligner l'aspect de rétention de l'environnement d'une fonction.

Par exemple, si des emplacements sont associés aux variables **a**, **b**, **c** et **d**, et que les fonctions primitives n'ont pas changées, alors:

i) `(set! a (lambda (x) (* x x)))` ⇒ *indéfini*
`(a 3)` ⇒ **9**

ii) `(set! b (lambda (x y) (x (x y))))` ⇒ *indéfini*
`(b a 3)` ⇒ **81**

iii) `(set! c (lambda (x y) (lambda (z) (x y z))))` ⇒ *indéfini*
`((c + 1) 2)` ⇒ **3**

iv) `(set! d (c b a))` ⇒ *indéfini*
`(d 3)` ⇒ **81**

L'évaluation de l'expression i) affecte à la variable **a** une fonction qui calcule le carré de son argument. L'évaluation de l'expression ii) affecte à la variable **b** une fonction à deux arguments qui calcule la double application du premier argument (qui doit être une fonction à un argument) sur le deuxième argument. L'expression **(b a** *n***)** calcule donc la quatrième puissance de *n*. Ceci démontre l'aise avec laquelle Scheme permet de manipuler des fonctions. L'évaluation de l'expression iii) affecte à la variable **c** une fonction à deux arguments. Celle-ci, lorsqu'elle est appelée avec deux arguments (disons a_1 et a_2), retourne une fermeture à un

argument. Lorsque cette dernière est appelée avec un argument (disons a_3) elle retourne $a_1[a_2,a_3]$. La valeur de l'expression **(c + 1)** est donc une fermeture qui additionne un à son argument. L'évaluation de l'expression iv) affecte à la variable **d** une fermeture à un argument. L'évaluation de **(d 3)** est équivalente à l'évaluation de **(b a 3)** car la fermeture affectée à **d** retient son environnement de création où les variables **x** et **y** valent respectivement la même chose que **b** et **a**.

Une caractéristique importante de Scheme est que la *récursivité terminale* est traitée correctement. On peut distinguer deux classes distinctes d'applications de fonctions: *non-terminales* et *terminales*. Les applications non-terminales sont celles qui sont utilisées comme argument d'une application ou comme condition d'une forme spéciale **if**. Lorsqu'une application est telle que sa valeur est retournée comme résultat de la fonction qui la contient, cette application est terminale. Par exemple dans **(lambda (x) (if (n x) (t x) (t (n x))))** les deux applications de **n** sont non-terminales et les deux applications à **t** sont terminales. Tel que décrit par Steele [Stee76] il n'est pas nécessaire d'utiliser d'espace mémoire supplémentaire sur la pile de contrôle pour effectuer les applications terminales car il suffit d'effectuer un branchement. En Scheme il est défini que les applications terminales ne consomment pas d'espace supplémentaire sur la pile ce qui permet d'exprimer toute forme itérative uniquement à l'aide d'applications terminales récursives. C'est de cette façon que la forme itérative **do** est implantée (voir la section 1.2.4.11).

1.2.4 Macros

Cette section décrit les autres formes spéciales et la façon de les traduire en termes des formes spéciales de base: **quote**, **set!**, **if** et **lambda**. Les transformations décrites proviennent en majeure partie de celles employées dans [Stee78b], [Abel85a] et [Abel85b] auxquelles nous avons apporté certaines améliorations. Pour décrire les transformations, nous utilisons la notation suivante:

$\alpha \equiv \beta$	indique que α est traduit en β
`A,B,C,D,E,F`	représentent des expressions
`a,b,c`	représentent des variables
`x`	représente une donnée
α …	représente une séquence, possiblement nulle, de α
`?`	représente une valeur indéfinie
$\otimes$	représente une variable artificielle créée lors de la traduction et qui est différente de toute autre variable

Il est à remarquer qu'un bon nombre de formes spéciales implantées à l'aide de macros se traduisent en termes d'autres macros et parfois récursivement en termes du macro original. Il n'y a

cependant aucun problème à effectuer les transformations de cette façon car les macros de l'expression produite seront à leur tour traduit et éventuellement il ne restera que les formes spéciales de base dans l'expression finale.

1.2.4.1 **let**

Cette forme spéciale permet d'évaluer des expressions dans l'environnement présent augmenté de nouvelles variables. L'évaluation de la forme **(let ((a A)…) B C…)** consiste à évaluer les expressions A… (dans un ordre indéterminé), puis d'évaluer séquentiellement les expressions B C… dans l'environnement présent augmenté temporairement des associations entre les variables de a… et de nouveaux emplacements contenant les valeurs correspondantes de A…. La valeur de cette forme est la valeur de la dernière expression de B C…. Par exemple:

(let ((a 1) (b 2)) (+ a b)) ⇒ 3

La forme spéciale **let** se traduit en l'application d'une fermeture obtenue grâce à l'évaluation d'une λ-expression. Voici la règle de traduction:

(let ((a A)…) B C…) ≡ ((lambda (a…) B C…) A…)

1.2.4.2 **let***

La forme spéciale **let*** est similaire à la forme spéciale **let** sauf que l'augmentation de l'environnement se fait une variable à la fois à partir de la gauche et que l'expression correspondant à la ième variable est évaluée dans l'environnement présent augmenté des i-1 premières associations. Par exemple:

(let* ((a 1) (b (+ a 1))) (+ a b)) ⇒ 3

La forme spéciale **let*** possède une règle de traduction récursive en termes de la forme spéciale **let**:

```
(let* () A B…)                ≡ (let () A B…)
(let* ((a A) (b B)…) C D…) ≡ (let ((a A))
                                 (let* ((b B)…) C D…))
```

1.2.4.3 **letrec**

La forme spéciale **letrec** est similaire à la forme spéciale **let** et permet de créer des

fonctions mutuellement récursives. L'évaluation de la forme **(letrec ((a A)…) B C…)** consiste à augmenter temporairement l'environnement présent des associations entre les variables a… et des emplacements contenant des valeurs indéfinies, évaluer les expressions A… (dans un ordre indéterminé), affecter ces valeurs aux variables correspondantes puis d'évaluer B C… séquentiellement dans cet environnement. La valeur de cette forme est la valeur de la dernière expression de B C…. Par exemple:

```
(letrec ((a (lambda (x) (* 2 (b (- x 1)))))
         (b (lambda (x) (if (= x 0) 1 (a x)))))
  (b 1)) ⇒ 2
```

La forme spéciale **letrec** se traduit en termes des formes spéciales **let** et **set!**. La dernière forme spéciale **let** permet d'inclure des définitions locales dans le corps du **letrec** tel que discuté dans la section 1.2.4.12. Voici la règle de traduction:

```
(letrec ((a A)…) B C…) ≡ (let ((a ?)…)
                           (let ((⊗ A)…)
                             (set! a ⊗)…
                             (let () B C…)))
```

1.2.4.4 **rec**

La forme spéciale **rec** permet de créer des fonctions auto-récursives. L'évaluation de la forme **(rec a A)** consiste à augmenter temporairement l'environnement présent de la variable a (de valeur indéfinie) puis d'affecter la valeur de l'expression A à cette variable. La valeur de cette forme est la valeur de la variable a. Par exemple:

```
((rec fact (lambda (x)
             (if (< x 2) 1 (* x (fact (- x 1))))))
 3) ⇒ 6
```

La forme spéciale **rec** se traduit en termes de la forme spéciale **letrec**:

```
(rec a A) ≡ (letrec ((a A)) a)
```

1.2.4.5 **named-lambda**

La forme spéciale **named-lambda** a une syntaxe similaire à une λ-expression sauf que le premier élément de la liste de paramètres est une variable dont la valeur, lors de l'évaluation du

corps de la fonction résultante, est la fonction elle même. L'exemple donné pour la forme spéciale **rec** pourrait se réécrire:

```
((named-lambda (fact x)
   (if (< x 2) 1 (* x (fact (- x 1)))))
 3) ⇒ 6
```

La forme spéciale **named-lambda** se traduit en termes de la forme spéciale **rec**:

```
(named-lambda (a b…) A B…) ≡ (rec a (lambda (b…) A B…))
```

1.2.4.6 **begin**

La forme spéciale **begin** permet d'évaluer des expressions en séquence. L'évaluation de la forme **(begin** A B…**)** consiste à évaluer séquentiellement, à partir de la gauche, les expression A B…. La valeur de cette forme est la valeur de la dernière expression de A B…. Puisque la valeur des premières expressions est ignorée, cette forme est surtout utilisée pour effectuer le séquencement d'effets de bord (e.g. les entrées/sorties). Par exemple:

(begin (write 1) (write 2) 3) ⇒ **3** (écrit **1** puis **2**)

La forme spéciale **begin** possède une règle de traduction récursive en termes de la forme spéciale **let**. Le séquencement est dû au principe que la valeur de la variable artificielle ⊗ est calculée avant d'évaluer le corps du **let**:

```
(begin A)       ≡ A
(begin A B C…) ≡ (let ((⊗ A)) (begin B C…))
```

1.2.4.7 **and**

La forme spéciale **and** est une forme de séquencement conditionnel. L'évaluation de la forme **(and** A B…**)** consiste à évaluer séquentiellement, à partir de la gauche, les expression A B… jusqu'à ce qu'une soit **#!false** ou que toutes soient évaluées. La valeur de cette forme est la valeur de la dernière expression évaluée. Par exemple:

(and (< 2 1) (> 2 1)) ⇒ **#!false**

La forme spéciale **and** possède une règle de traduction récursive en termes de la forme spéciale **if**.

```
(and A)       ≡ A
(and A B C…) ≡ (if A (and B C…) #!false)
```

1.2.4.8 **or**

La forme spéciale **or** est une forme de séquencement conditionnel. L'évaluation de la forme **(or** A B…**)** consiste à évaluer séquentiellement, à partir de la gauche, les expression A B… jusqu'à ce qu'une soit différente de **#!false** ou que toutes soient évaluées. La valeur de cette forme est la valeur de la dernière expression évaluée. Par exemple:

```
(or (< 2 1) (> 2 1)) ⇒ #!true
```

La forme spéciale **or** possède une règle de traduction récursive en termes des formes spéciales **if** et **let**. La variable artificielle ⊗ est utilisée pour ne pas évaluer l'expression A deux fois:

```
(or A)       ≡ A
(or A B C…) ≡ (let ((⊗ A)) (if ⊗ ⊗ (or B C…)))
```

1.2.4.9 **cond**

La forme spéciale **cond** est une forme d'évaluation conditionnelle. L'évaluation de la forme **(cond (**A B…**)…)** consiste à évaluer séquentiellement l'expression A de chaque clause **(**A B…**)** jusqu'à ce qu'une soit différente de **#!false**. Les expressions restantes de cette clause sont ensuite évaluées séquentiellement. La valeur de la forme **cond** est la valeur de la dernière expression évaluée. Par contre, si toutes les expressions A valent **#!false**, la valeur de la forme **cond** est indéfinie. Le symbole **else** peut être utilisé à la place de A en quel cas, il est équivalent à **#!true**. Par exemple:

```
(cond ((< 2 1) 'petit)
      ((> 2 1) 'grand)
      (else    'egal)) ⇒ grand
```

La forme spéciale **cond** possède une règle de traduction récursive en termes des formes spéciales **if**, **or** et **begin**:

```
(cond)                      ≡ ?
(cond (A) (B C…)…)          ≡ (or A (cond (B C…)…))
(cond (A B C…) (D E…)…) ≡ (if A
                              (begin B C…)
                              (cond (D E…)…))
```

1.2.4.10 **case**

La forme spéciale **case** est une forme d'évaluation conditionnelle. L'évaluation de la forme **(case** A **((x…)** B C…**)…)** consiste à chercher séquentiellement une des clauses **((x…)** B C…**)** qui contient la valeur de l'expression A (le sélecteur) dans sa partie **(x…)**. Les expressions restantes de la clause trouvée sont ensuite évaluées séquentiellement. La valeur de la forme **case** est la valeur de la dernière expression évaluée. Par contre, si aucune des parties **(x…)** ne contient le sélecteur, la valeur de la forme **case** est indéfinie. Le symbole **else** peut être utilisé à la place de **(x…)** en quel cas, il est équivalent à une liste qui contient le sélecteur. Par exemple:

```
(case (* 2 3) ((2 3 5 7)    'premier)
              ((1 4 6 8 9)  'compose)
              (else         'inconnu)) ⇒ compose
```

La forme spéciale **case** se traduit en termes des formes spéciales **let**, **cond** et de la fonction primitive **memv** qui vérifie l'appartenance de son premier argument dans la liste qui le suit. La variable artificielle ⊗ est utilisée pour mémoriser le sélecteur pendant l'évaluation de la forme **case**:

```
(case A ((x…) B C…)…) ≡ (let ((⊗ A))
                           (cond ((memv ⊗ '(x…)) B C…)…))
```

1.2.4.11 **do**

La forme spéciale **do** est une forme d'évaluation itérative. L'évaluation de la forme **(do ((**a A B**)…) (**C D…**)** E F…**)** consiste à augmenter temporairement l'environnement présent des variables a… initialisées aux valeurs des expressions A… puis, tant que la valeur de l'expression C est **#!false**, à évaluer toutes les expressions E F… séquentiellement. À chaque

nouveau tour de boucle, la valeur de B est affectée à la variable a correspondante (à moins que B n'ait été omis). Lorsque C est différent de **#!false**, les expressions D… sont évaluées séquentiellement. La valeur de la forme **do** est la valeur de la dernière expression évaluée. Par contre, si la partie D… est omise, la valeur de la forme **do** est indéfinie. Voici un exemple d'utilisation de la forme spéciale **do**:

```
(do ((x '(2 3 5 7) (cdr x)) (somme 0))
    ((null? x) somme)
  (if (odd? (car x))
    (set! somme (+ somme (car x))))) ⇒ 15
```

La forme spéciale **do** se traduit en termes des formes spéciales **named-lambda**, **if** et **begin** ainsi que de l'application. L'itération est obtenue à l'aide de la récursion d'une fonction qui est nommée grâce à la variable artificielle ⊗. Cette façon d'implanter l'itération n'entraîne pas de problème de mémoire car la récursivité terminale est traitée correctement en Scheme. La forme a/B représente l'expression B correspondante si elle existe, sinon elle représente la variable a correspondante:

```
(do ((a A B)…) (C D…) E F…) ≡
((named-lambda (⊗ a…)
   (if C (begin ? D…) (begin E F… (⊗ a/B…))))
 A…)
```

1.2.4.12 **define**

La signification de la forme spéciale **define** dépend de l'endroit où elle est évaluée. Néanmoins, quelle que soit sa position, la règle suivante s'applique:

```
(define (x a…) A B…) ≡ (define x (lambda (a…) A B…))
```

Cette règle permet de définir des fonctions sans avoir à utiliser explicitement de forme spéciale **lambda**, ce qui serait un peu lourd. De plus, lorsque x est lui-même une liste, on peut définir des fonctions qui retournent des fonctions. Par exemple:

```
(define ((a b) c) (+ b c)) ≡
(define a (lambda (b) (lambda (c) (+ b c))))
```

Si la forme **(define** a A**)** est au top-level, son évaluation consiste à ajouter à l'environnement global l'association entre la variable a et un nouvel emplacement (à condition que

la variable `a` ne soit pas déjà associée à un emplacement dans l'environnement global) puis à faire l'équivalent de **`(set! a A)`**.

On peut également retrouver une séquence de formes **`define`** au début du corps d'une forme spéciale **`lambda`**, **`named-lambda`**, **`let`**, **`let*`**, **`letrec`** ou **`define`**. À ce moment, les définitions sont locales au corps de la forme en question et peuvent être utilisées à l'intérieur des définitions qui s'y trouvent. L'ordre dans lequel ces définitions sont évaluées est indéterminé. Par exemple:

```
(define (surface diametre)
  (define pi 3.1416)
  (define (carre x) (* x x))
  (define (aire rayon) (* pi (carre rayon)))
  (aire (/ diametre 2))) ⇒ indéfini

(aire 4) ⇒ 12.5664
```

Puisque les formes pouvant contenir des définitions locales se traduisent éventuellement sous forme de λ-expression, une définition locale peut être traduite en termes des formes spéciales **`letrec`** et **`begin`** uniquement à l'aide de la règle suivante:

```
(lambda (a…) (define b A) (define c B)… C D…) ≡
(lambda (a…) (letrec ((b A) (c B)…) C D…))
```

Par contre, si la première expression du corps d'une λ-expression n'est pas une forme spéciale **`define`** alors:

```
(lambda (a…) A B…) ≡ (lambda (a…) (begin A B…))
```

Ces deux dernières règles permettent de traduire les λ-expressions qui possèdent plusieurs expressions dans leurs corps en λ-expression qui n'en possèdent qu'une. La forme spéciale **`lambda`** peut donc être considérée comme un macro qui se traduit en termes de forme spéciale **`lambda`** plus primitive. Par exemple, après toutes les réécritures, l'expression **`(lambda () 1 2 3)`** devient:

```
(lambda () ((lambda (⊗1) ((lambda (⊗2) 3) 2)) 1))
```

1.2.5 Lien avec le λ-calcul

Chaque construction du λ-calcul a son équivalent en Scheme. La référence de variable existe en Scheme comme pour le λ-calcul, l'application de fonction correspond à la combinaison

du λ-calcul et finalement la forme spéciale **`lambda`** correspond à l'abstraction du λ-calcul. L'évaluation d'une expression Scheme correspond à l'évaluation en ordre applicatif puisque les arguments d'une fonction sont évalués avant d'évaluer le corps de celle-ci.

L'évaluation en ordre applicatif de programmes λ-calcul peut s'écrire en Scheme si l'on suit la règle de transformation ℜ[<expression λ-calcul>] → <expression Scheme> suivante:

si `a` est une variable et `A`, `B` sont des expressions λ-calcul alors:

```
ℜ[a]      = a
ℜ[λaA]    = (lambda (ℜ[a]) ℜ[A])
ℜ[(AB)]   = (ℜ[A] ℜ[B])
```

Le phénomène de fermeture de Scheme est étroitement lié à la règle de β-conversion du λ-calcul. Si le corps d'une abstraction contient une abstraction et que cette dernière contient des références au paramètre de l'abstraction externe alors, aussitôt que l'argument associé au paramètre est connu, les références de l'abstraction interne peuvent être remplacées par l'argument. L'expression qui remplace une variable libre d'une abstraction provient donc du paramètre de même nom d'une abstraction qui l'englobe. Conséquemment, si on emprunte les termes utilisés pour décrire Scheme, on peut dire que la valeur d'une variable libre provient de l'environnement en effet au point où se trouve l'abstraction qui la contient. Ainsi, si on ne considère pas l'affectation, la création d'une fermeture en Scheme est l'équivalent de la substitution des variables libres d'une abstraction à l'aide de la règle de β-conversion.

1.3 Rétrospective

Nous avons démontré comment les formes spéciales de Scheme peuvent être implantées à l'aide des formes spéciales de base **`quote`**, **`set!`**, **`if`** et **`lambda`** et des constructions de référence de variable et d'application de fonction. La création et l'application de fermetures sont centrales à cette approche et sont utilisées à profusion.

De plus, les fermetures sont un outil de programmation versatile. Leur existence incite à la programmation fonctionnelle. Cette discipline décourage l'utilisation d'effets de bord en offrant des alternatives aux opérations traditionnellement effectuées à l'aide d'effets de bord. Le traitement des fonctions comme toute autre donnée améliore l'homogénéité du langage et rend possible les fonctions qui calculent et manipulent des fonctions. Comme nous le verrons au prochain chapitre, les fermetures peuvent aussi être utilisées pour exprimer des abstractions de données, pour implanter les acteurs, les classes et les modules.

Les fermetures jouent donc un rôle particulièrement important dans le langage Scheme, que ce soit au niveau de la traduction des formes spéciales par le compilateur ou au niveau des programmes de l'utilisateur. Ainsi, il existe un lien étroit entre la performance d'un système Scheme et l'implantation efficace des fermetures. Pour cette raison, nous avons étudié cet aspect de plus près. Le deuxième chapitre traite de l'implantation des fermetures et décrit une nouvelle méthode efficace que nous avons développée.

Un autre aspect qui influence la performance d'un système Scheme basé sur un compilateur est la qualité du code généré par celui-ci. Nous avons développé une méthode originale de génération de code qui est basée sur la création de fermetures. Cette méthode fait l'objet du troisième chapitre. Finalement, l'intégration de ces deux méthodes dans un compilateur Scheme est discutée au quatrième chapitre.

Chapitre II

L'implantation des fermetures

Dans ce chapitre, nous décrivons une méthode efficace d'implantation des fermetures et la comparons à une autre méthode plus classique. La première section constitue une introduction générale aux fermetures. Nous décrivons une méthode d'implantation 'classique' des fermetures et des environnements. Cette section est suivie d'une section qui décrit la méthode d'implantation des fermetures que nous avons développée et qui la compare à la méthode précédente. Nous montrons aussi comment notre méthode peut être utilisée pour implanter les fermetures dans les dialectes de Lisp qui ne les ont pas.

2.1 Introduction générale aux fermetures

2.1.1 Utilité des fermetures

Les fermetures sont un outil de programmation très versatile. Entre autres, elles peuvent être utilisées pour:

- exprimer des abstractions de données [Abel85a]
- implanter des acteurs [Stee76] [Suss75]
- implanter les classes et fournir la protection des données [Wand80]
- implanter les modules, la compilation séparée et les *database views* [Atki85]
- représenter le code généré par un compilateur (voir le troisième chapitre)

De plus, elles accomodent les méthodologies de *data driven programming* [Stal77], de programmation fonctionnelle et de programmation orientée objet. Nous incluons deux exemples d'utilisation de fermetures. Le premier est un exemple simple de création de fermetures:

```
(define (adder x)
  (lambda (y) (+ x y)))

; utilisation de la fonction adder:

(define add1 (adder 1))
(define add5 (adder 5))
(add1 1) ⇒ 2
(add5 1) ⇒ 6
```

En termes informels, la fonction **adder** permet de générer des fonctions de la classe 'additionner à'. Lorsqu'elle est appelée avec un nombre, **adder** retourne une fermeture à un argument qui calcule, lors de son application, la somme de son argument avec ce nombre. Dans l'exemple, une fonction qui additionne un est affectée à **add1** et une fonction qui additionne cinq est affectée à **add5**. Ce phénomène repose sur le fait que l'environnement de définition qui est retenu par ces fermetures contient le paramètre de **adder** (i.e. la variable **x**). Puisque c'est cet environnement qui est utilisé lors de la référence à **x**, la fermeture résultant de l'appel à **adder** est une fonction qui additionne son argument à l'argument qui a été passé à **adder**. Il est à remarquer qu'il n'est pas nécessaire de stocker les fermetures dans des variables avant de les utiliser, on aurait tout aussi bien pu faire **((adder 2) 3) ⇒ 5**.

Le deuxième exemple, inspiré de [Abel85a], est un exemple classique d'utilisation des fermetures pour exprimer un type abstrait de donnée et les opérations permettant de manipuler des données de ce type. L'exemple démontre comment il est possible d'implanter les doublets de Scheme sans avoir recours à d'autres structures de données:

```
(define (cons the-car the-cdr)
  (lambda (msg)
    (cond ((= msg 1) (lambda () the-car))
          ((= msg 2) (lambda () the-cdr))
          ((= msg 3) (lambda (v) (set! the-car v)))
          ((= msg 4) (lambda (v) (set! the-cdr v)))
          (else      (error "undefined operation")))))

(define (car pair)          ((pair 1)))
(define (cdr pair)          ((pair 2)))
(define (set-car! pair val) ((pair 3) val))
(define (set-cdr! pair val) ((pair 4) val))

; utilisation de fonctions cons, car, cdr et set-car!:

(define a (cons 'x 'y))
(car a) ⇒ x
(cdr a) ⇒ y
(set-car! a 'z)
(car a) ⇒ z
```

La fonction **cons**, lorsqu'elle est appelée avec deux arguments, retourne une fermeture qui retient l'environnement en existence à ce point. Celui-ci contient, entre autres, l'association entre la variable **the-car** et un emplacement contenant le premier argument passé à **cons** et l'association entre la variable **the-cdr** et un emplacement contenant le deuxième argument passé à **cons**. La fermeture résultante représente un doublet. Lorsque celle-ci est appelée avec une des valeurs **1**, **2**, **3** ou **4**, elle retourne une autre fermeture qui permet, selon le cas, d'obtenir la valeur de la variable **the-car** ou **the-cdr** ou de modifier l'emplacement associé à ces variables

dans l'environnement retenu. Les fonctions **`car`**, **`cdr`**, **`set-car!`** et **`set-cdr!`** passent le message approprié à la fermeture représentant le doublet et appellent la fermeture qui en résulte. Dans le cas de **`set-car!`** et **`set-cdr!`**, un argument représentant la nouvelle valeur du champ est passé lors de cette application. À ce moment, la valeur de la variable est modifiée et les références subséquentes à ce champ retourneront la nouvelle valeur. Il est à noter que plusieurs doublets peuvent être créés de cette façon car chaque fermeture représentant un doublet a son propre environnement indépendant des autres.

Notons, ici, une analogie entre les fermetures et les instances de classes (i.e. les objets) des langages orienté-objet tels que Smalltalk [Gold83] et SIMULA 67 [Dahl82]. Les fermetures possèdent, grâce à l'environnement de définition qu'elles retiennent, un état local qui leur est propre. De leur côté, les objets possèdent un état local contenu dans les champs de leur classe. Cependant, les objets peuvent avoir plusieurs méthodes qui leurs sont associées. Les fermetures n'en ont qu'une (i.e. leur corps). Néanmoins, tel que démontré dans l'exemple précédent, il est possible de simuler des méthodes multiples en retournant, comme résultat de la fermeture, une fermeture représentant une méthode particulière. Celle-ci est sélectionnée à l'aide d'un message et est appelée avec les arguments nécessaires par le requérant. Le phénomène d'héritage a aussi sa contrepartie pour les fermetures bien qu'il soit sous une forme moins flexible. Comme un objet hérite des champs et des méthodes de sa super-classe, une fermeture hérite des variables des λ-expressions qui englobent sa définition. L'évaluation d'une λ-expression est donc semblable à l'instanciation d'une classe et les fermetures peuvent être utilisées de façon similaire aux objets.

L'utilité des fermetures est centrée sur le fait qu'elles possèdent un état local. Sans cela, les fermetures seraient simplistes et bien moins versatiles. On ne pourrait définir que des fonctions monolithiques et immuables. Sans fermetures, il ne serait plus possible de créer de nouvelles paramétrisations des fonctions lors de l'exécution du programme réduisant de beaucoup la puissance expressive de Scheme.

2.1.2 Variables

Les variables forment un élément de base des langages de programmation. Elles permettent de nommer les données résultant d'un calcul afin de simplifier leur manipulation subséquente. Une variable désigne un emplacement où une donnée peut être stockée. La valeur d'une variable peut être accédée à l'aide des opérations de référence et d'affectation de variable.

Les emplacements désignés par les variables sont créés par les constructions de liaison. En Scheme, la construction de liaison la plus fondamentale est la λ-expression. Tel qu'indiqué au premier chapitre, les autres constructions de liaison (e.g. **`let`**, **`letrec`** et **`do`**) peuvent être

exprimées à l'aide de λ-expression. Sans perte de généralité, nous considèrerons que les λ-expressions sont la seule construction de liaison disponible.

Suivant la même terminologie que pour le λ-calcul, nous distinguons deux sortes de variables: *liées* et *libres*. Une variable est liée par rapport à une λ-expression si elle apparait dans sa liste de paramètres, sinon, elle est libre. Une variable est dite *intermédiaire* par rapport à une λ-expression si elle apparait dans la liste de paramètres d'une autre λ-expression qui l'englobe. De plus, nous définissons les classes de variables suivantes:

- une variable qui est libre par rapport à toutes les λ-expressions est une variable *globale*
- une variable qui est liée par rapport aux λ-expressions qui entourent directement chaque accès (i.e. référence et affectation) à cette variable est une variable *locale*
- une variable qui est, au moins une fois, intermédiaire par rapport aux λ-expressions qui entourent directement chaque accès à cette variable est une variable *fermée*

Les variables globales sont généralement définies par une forme spéciale **`define`** entrée au top-level tandis que les variables locales et fermées sont déclarées dans la liste de paramètres des λ-expressions. Ces classes sont distinctes et chacune des variables appartient à une seule de ces classes. Par exemple, dans l'expression:

```
(lambda (x y)
  (list x y (lambda (z) (+ x z))))
```

les variables **`list`** et **`+`** sont globales, les variables **`y`** et **`z`** sont locales et la variable **`x`** est fermée (car elle est intermédiaire par rapport à la λ-expression interne). Chacune de ces classes a des particularités qui seront discutées plus loin.

2.1.3 Environnements

L'ensemble de toutes les associations entre variables et emplacements effectives à un point d'exécution du programme est l'*environnement* en effet à ce point. L'environnement est utilisé pour découvrir quel emplacement est couramment associé à une variable lors d'une référence de variable ou d'une affectation.

Ceci n'implique pas que la structure qui représente l'environnement contient nécessairement le nom des variables couramment accessibles. Tout ce qui importe c'est que l'emplacement couramment associé à une variable soit à la même position relative à la structure qui représente l'environnement (i.e. qu'il existe un algorithme fixe pour retrouver l'emplacement à partir de l'environnement).

2.1.3.1 Environnements chaînés

La façon classique d'implanter les environnements pour les langages à portée statique est à l'aide d'une chaîne d'enregistrements [Rand64] [Aho77]. Chaque enregistrement de la chaîne correspond à une des λ-expressions dans laquelle le point d'exécution courant est contenu. Les enregistrements sont ordonnés par ordre décroissant de niveau d'imbrication (i.e. le premier enregistrement de la chaîne correspond à la λ-expression qui entoure directement le point d'exécution). Chaque enregistrement contient les valeurs des variables déclarées dans la λ-expression qui lui est associée et un pointeur vers l'enregistrement suivant (qu'on appelle le *lien statique*). Le dernier enregistrement de la chaîne contient les variables globales. La création de nouveaux emplacements lors de l'application d'une fermeture consiste à allouer un nouvel enregistrement contenant le bon nombre de champs et de le chaîner au devant de l'environnement de définition de la fermeture. À titre d'exemple, considérons l'expression suivante, entrée au top-level:

```
(set! a (lambda (f x)
          (lambda (y)
            ((lambda (z) (f z z)) (+ x y)))))  ⇒ indéfini
```

La valeur de la variable **a** est une fonction qui, lors de son application à deux arguments (disons a_1 et a_2), retourne une fonction à un argument (disons a_3) qui calcule $a_1[a_2+a_3,a_2+a_3]$. Lors de l'évaluation de l'expression **((a * 1) 2)** ⇒ **9**, l'environnement dans lequel le corps de la λ-expression la plus interne est évaluée est représenté par la chaîne d'enregistrements de la figure 2.1.

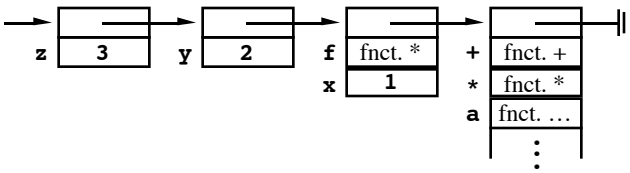


Fig 2.1: Chaîne d'enregistrements

Les variables sont accédées en descendant la chaîne jusqu'au bon enregistrement et en accédant le champ approprié de cet enregistrement. Pour chaque accès particulier à une variable, la profondeur dans la chaîne et le champ accédé ne varie pas. Dans l'exemple, la référence à la variable **f** dans le corps de la λ-expression la plus interne consiste à descendre jusqu'au troisième enregistrement et à accéder le deuxième champ.

Un inconvénient de cette méthode est que le temps requis pour accéder à une variable est proportionnel à la profondeur de l'enregistrement qui la contient. Si le niveau d'imbrication des

λ-expressions est grand et que les variables profondes (e.g. les variables globales) sont accédées fréquemment, cette méthode peut être coûteuse.

2.1.3.2 Environnements à base de *display*

Le coût associé aux environnements chaînés peut être réduit en représentant la séquence d'enregistrements à l'aide d'un vecteur de pointeurs (connu sous le nom de *display*) à la place des liens statiques [Dijk67]. Chaque pointeur du *display* pointe sur un des enregistrements de l'environnement. L'accès à une variable consiste en une indirection du pointeur approprié du *display*. La figure 2.2 illustre l'environnement correspondant à l'exemple précédent en utilisant un *display*.

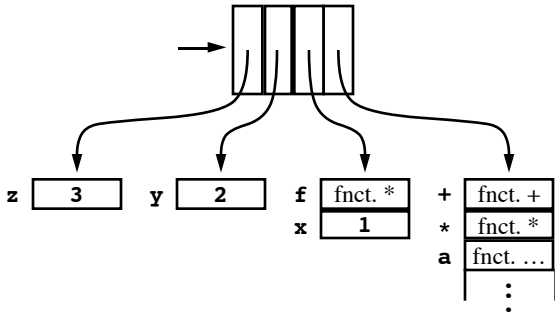


Fig 2.2: Display et enregistrements

De cette façon, la référence à la variable **f** dans le corps de la λ-expression la plus interne consiste à accéder le premier champ de l'enregistrement pointé par la troisième entrée du *display*.

2.1.4 Implantation classique des fermetures

Les deux opérations principales sur les fermetures sont la création et l'application. Les fermetures sont créées par l'évaluation de λ-expressions et sont appelées par l'évaluation d'une application. Le résultat d'une application est le résultat de l'évaluation du corps de la fermeture dans un environnement constitué de l'environnement de définition de la fermeture (i.e. l'environnement en existence au moment où la fermeture a été créée) et de nouvelles associations variables-emplacements. Ces associations proviennent des arguments qui ont été passés lors de l'application. Pour chaque paramètre de la fermeture un emplacement est créé et l'argument correspondant y est stocké. Puisque l'environnement de définition d'une fermeture est nécessaire, plus tard, pour évaluer son corps, il doit y avoir une façon de le récupérer à partir de la fermeture.

La façon classique de représenter une fermeture est à l'aide d'une structure de donnée constituée de deux parties. La première est le code pour évaluer le corps de la fermeture et la seconde est un environnement. La création d'une fermeture consiste à allouer et initialiser les parties de cette structure. De façon à sauver de l'espace, la première partie est en fait un pointeur vers le code car celui-ci est identique pour toutes les fermetures créées pour la même définition. La seconde partie est l'environnement de définition de la fermeture. Par exemple, si l'on suppose que les environnements sont représentés à l'aide d'une chaîne d'enregistrements, la fermeture créée pour la λ-expression la plus interne de l'exemple précédent est représentée par la structure de la figure 2.3.

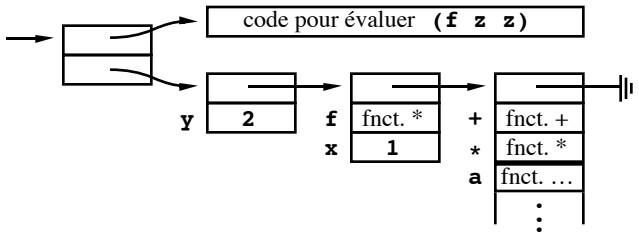


Fig 2.3: Représentation classique d'une fermeture

Lorsque cette fermeture est appelée, un nouvel environnement est construit en allouant un enregistrement à deux champs (pour la variable **z** et le lien statique) à l'avant de l'environnement de définition. Le champ associé à la variable **z** est initialisé à la valeur de l'argument de la fermeture (dans l'exemple, le nombre **3**). Un branchement est alors effectué au code qui évalue le corps en lui passant l'environnement construit et le point de continuation dans le programme (i.e. l'adresse de retour).

L'allocation de l'enregistrement à l'avant de l'environnement peut aussi être effectuée par le code de la fermeture. Pour ce faire, l'appelant doit lui passer chacun des arguments, l'environnement de définition et l'adresse de retour. Cette technique a l'avantage de réduire la longueur des programmes puisque le code d'allocation ne se trouve pas duplicaté pour chacune des applications de fermeture et de permettre un traitement plus simple pour les paramètres reste puisque c'est la fermeture appelée qui en connait l'existence. Par contre, elle nécessite une convention de passage des arguments plus complexe.

Il est à noter que l'environnement dans lequel le corps d'une fermeture est évalué n'a aucun rapport avec l'environnement de l'appelant. Ceci n'est pas le cas des dialectes de Lisp à portée dynamique qui, dans un sens, allouent l'enregistrement associé aux paramètres d'une fonction à l'avant de l'environnement de l'appelant et désallouent celui-ci au retour de la fonction. Dans ce cas, l'allocation et la désallocation des enregistrements suit un régime similaire à une pile ce qui permet d'utiliser la pile d'exécution pour conserver les paramètres. L'opération d'allocation, à

l'entrée d'une fonction, consiste à empiler les paramètres sur la pile et la désallocation, à la sortie de la fonction, consiste à déplacer le pointeur de pile d'une distance égale au nombre de paramètres alloués.

2.1.5 Efficacité

Les fermetures, comme toutes les autres données, ont une durée de vie illimitée. Elles peuvent être actives longtemps après l'exécution de la construction qui les a créées. Généralement, les fermetures ainsi que les environnements qu'elles contiennent doivent donc être alloués d'un monceau. C'est lors de l'opération de *garbage collection* que ceux-ci sont désallouées s'ils ne sont plus nécessaires. Cette opération, qui consiste en une preuve de non-accessibilité des objets, est généralement coûteuse et est souvent la source d'inefficacités dans les implantations qui s'y fient fortement.

Une façon de réduire l'importance de ce problème est d'allouer autant que possible les environnements sur la pile d'exécution. Ceci peut être effectué par un compilateur astucieux lorsqu'il détecte que l'existence d'une portion de l'environnement est limitée [Stee76] [Stee78b]. À titre d'exemple, considérons l'expression suivante:

```
(lambda (x)
  ((lambda (z) z)
   (lambda (y) (+ x y))))
```

Lorsque la fermeture qui résulte de l'évaluation de cette expression est appelée, une fermeture, utilisant la variable **x**, est retournée. Puisque l'on ne peut rien dire sur la durée de vie de cette dernière fermeture, qui pourrait, par exemple, être stockée dans une variable globale et appelée beaucoup plus tard, l'allocation de l'emplacement associé à la variable **x** doit s'effectuer du monceau. Considérons maintenant l'expression suivante obtenue en remplaçant, dans l'expression précédente, la référence à la variable **z** par l'application de fonction **(z 1)**:

```
(lambda (x)
  ((lambda (z) (z 1))
   (lambda (y) (+ x y))))
```

À ce moment, la fermeture qui utilise la variable **x**, c'est à dire la fermeture créée par **(lambda (y) (+ x y)**, n'est plus accessible après l'exécution du corps de la fermeture créée par **(lambda (x)…** car elle ne sert qu'à être appliquée à **1**. Ainsi, le seul moment où la variable **x** est accédée est pendant l'exécution du corps de la fermeture qui l'a allouée et conséquemment, elle peut être désallouée à la sortie de la fermeture. Donc, l'existence de la variable **x** suit un régime similaire à une pile et peut être allouée sur la pile d'exécution. Afin

d'effectuer cette optimisation, le compilateur doit avoir des connaissances sur la sémantique des λ-expressions, de l'application des fermetures et des fonctions primitives.

Une autre optimisation consiste à allouer, de façon consistante, les environnements sur la pile d'exécution et de les copier sur le monceau seulement si cela s'avère nécessaire [McDe80]. Cette méthode est particulièrement bien adaptée pour les interprètes car la détection de cette situation peut se faire à l'exécution.

Il existe plusieurs astuces d'implantation qui permettent de représenter les environnements efficacement. Par exemple, les variables globales peuvent être implantées en utilisant un champ de l'enregistrement qui représente le symbole associé à la variable. Ceci est possible car il n'existe qu'un seul environnement global (i.e. chaque variable globale n'est associée qu'à un seul emplacement). L'allocation d'une variable globale se fait au moment de l'introduction du symbole qui la dénote et l'accès à sa valeur ne nécessite qu'une indirection du pointeur vers le symbole.

Les variables locales peuvent aussi être implantées de façon efficace en les allouant sur la pile d'exécution. Ceci est possible car, par définition, une telle variable n'est jamais accédée par une autre fermeture que celle qui la déclare en paramètre. La durée de vie d'une variable locale est donc limitée à l'exécution du corps de la fermeture qui la déclare en paramètre et peut être désallouée à sa sortie.

Ainsi, en utilisant les astuces précédentes pour implanter les variables globales et locales, l'environnement retenu par une fermeture est uniquement nécessaire pour accéder aux variables fermées. Si de telles variables n'existaient pas, il ne serait pas utile de créer des fermetures car l'environnement de définition ne servirait jamais. Une fermeture serait simplement un pointeur vers le code évaluant son corps. Son application ne nécessiterait pas la manipulation explicite d'un descripteur d'environnement. Il suffirait d'un branchement au code de la fermeture après avoir emmagasiné les arguments et l'adresse de retour sur la pile d'exécution et/ou dans des registres. Cette idée peut d'ailleurs faire partie des optimisations effectuées par un compilateur. Le compilateur pourrait omettre l'environnement de définition d'une fermeture qui ne contient aucun accès à des variables fermées. Néanmoins, de façon générale, une fermeture qui accède à des variables fermées est constituée d'un pointeur vers le code et l'environnement de définition.

Notre approche prend un point de vue différent qui unifie le concept de fermeture et d'environnement. Au lieu d'être représentée par une structure de donnée à deux champs, une fermeture est un bout de code qui contient l'information associée à son environnement de définition. La création d'une fermeture consiste à générer ce bout de code au lieu du couple code-environnement. Cette méthode est présentée dans la section qui suit.

2.2 Notre méthode d'implantation des fermetures

Cette section décrit la méthode d'implantation de fermetures que nous avons développée. Notre méthode est apparentée à la technique des macros et consiste à exprimer les λ-expressions en termes de constructions plus simples du langage source. Dans la première partie nous expliquons comment effectuer cette traduction. Dans un premier temps, nous décrivons la méthode de base et procédons ensuite à des raffinements successifs. La méthode ainsi obtenue est analysée plus en profondeur dans la seconde partie. Dans la dernière partie nous expliquons comment utiliser cette méthode pour simuler les fermetures dans les dialectes de Lisp qui ne les possèdent pas.

2.2.1 La traduction de λ-expressions

2.2.1.1 Méthode de base

Pour l'instant, nous ne considérons pas l'affectation aux variables fermées. Cet aspect, qui rajoute certaines complications, est traité dans la section 2.2.1.4. La traduction de λ-expression se fait en termes des deux constructions cibles suivantes: les ε-expressions et les appels à la fonction **`compile`**.

Les ε-expressions, que nous avons inventé pour les besoins de la cause, sont une forme dégénérée de λ-expressions. Elles ont la même forme que les λ-expression sauf que le mot clé **`lambda`** est remplacé par **`epsilon`** et leur corps ne peut contenir que des accès à des variables globales et locales. Toutes les variables libres d'une expression sont des variables globales (i.e. le concept de variable fermée n'existe pas pour les ε-expressions). D'une certaine façon, les ε-expressions correspondent aux λ-expressions des dialectes de Lisp qui ne possèdent pas les fermetures. Toutes les évaluations d'une même ε-expression résultent en une même fonction qui ne possède pas d'information sur l'environnement de définition. Ainsi, les ε-expressions dénotent des fonctions constantes. Nous utilisons les ε-expressions dans notre approche car leur sémantique est simple et elles peuvent être implantées efficacement. Tel que vu à la section 2.1.5, les paramètres peuvent être allouées, à l'entrée des fonctions créées à partir d'ε-expressions, sur la pile d'exécution et désallouées à la sortie.

La fonction **`compile`**, à un argument, transforme la représentation, sous forme de donnée, d'une ε-expression en la fonction qui lui correspond. À chaque fois qu'elle est appelée, la fonction **`compile`** retourne une nouvelle fonction. Par exemple, l'évaluation des deux expressions suivantes résulte en une fonction équivalente:

```
(compile '(epsilon …)) ≡ (epsilon …)
```

On peut concevoir la fonction **compile** comme étant un constructeur de fonction. Elle a donc un point commun avec les λ-expressions qui créent aussi des fonctions. Cet aspect est à la base de notre méthode d'implantation des fermetures.

La traduction est fondée sur la similitude qui existe entre les λ-expressions et les abstractions du λ-calcul. Plus particulièrement, nous appliquons le principe de la règle de la β-conversion sur les λ-expressions. Rappelons nous la règle de la β-conversion qui stipule que, lorsqu'une abstraction est appliquée à un argument, toutes les références au paramètre, dans le corps de l'abstraction, peuvent être remplacées par l'argument. Les variables fermées des abstractions sont donc remplacées par l'argument correspondant lorsque celui-ci est connu.

La transposition de cette idée à Scheme consiste à effectuer la substitution textuelle de la valeur d'une variable fermée pour chacune de ses références dans les λ-expressions qui s'en servent. Cette substitution peut s'effectuer aussitôt que la valeur de la variable fermée est connue ou au moment de l'évaluation des λ-expressions qui s'en servent. Nous avons opté pour la deuxième qui a l'avantage d'éviter de faire des substitutions qui ne sont pas utilisées ultérieurement.

Ainsi, une λ-expression est transformée en une application de la fonction **compile**. L'argument passé est la représentation sous forme de donnée de la λ-expression avec le mot clé **lambda** remplacé par **epsilon** et les références aux variables fermées remplacées par leur valeur courante. L'expression qui suit (donnée en exemple à la section 2.1.1):

```
(define (adder x)
  (lambda (y) (+ x y)))
```

est traduite en l'expression suivante:

```
(define (adder x)
  (compile `(epsilon (y) (+ ',x y))))
```

qui est l'abréviation de:

```
(define (adder x)
  (compile (list 'epsilon '(y)
                 (list '+ (list 'quote x) 'y))))
```

Chaque fois qu'elle est appelée, la fonction **adder** construit une liste qui représente une ε-expression et la compile. La fonction résultante est parfaitement équivalente à la fermeture qui aurait été obtenue par l'évaluation de la λ-expression. Lorsque ce qui remplace la référence à la variable **x** est évalué, la valeur que **x** avait au moment de l'évaluation de la λ-expression est obtenue. C'est exactement ce que nous voulons. En quelque sorte, la valeur de la variable **x** est

gelée dans le code de la fonction créée. De façon générale, toutes les variables fermées d'une λ-expression sont gelées dans le code des fonctions qui en résulte.

À prime abord, on pourrait douter de l'efficacité de cette méthode. En effet, la création de fermetures est dispendieuse car elle nécessite la construction et la compilation d'une structure représentant une ε-expression. À chaque création d'une fermeture, il faut recompiler une expression, possiblement complexe, ce qui peut prendre beaucoup de temps et d'espace mémoire. Cependant, les fermetures créées de cette façon sont excellentes du point de vue de leur temps d'exécution car les références aux variables fermées ont été remplacées par des références à des constantes. Il n'est donc pas nécessaire d'effectuer une recherche, possiblement coûteuse, pour trouver la valeur des variables fermées. Une telle méthode est justifiable dans un contexte où les fermetures sont créées rarement mais appelées souvent et où l'espace mémoire n'est pas un problème. Par contre, dans un contexte plus raisonnable, on est prêt à perdre un peu de la vitesse d'exécution des fermetures au profit de leur temps de création et de l'espace mémoire utilisé.

2.2.1.2 Réduction du temps de création

Le problème soulevé dans la section précédente peut être résolu en observant que la plupart de la structure à compiler est identique pour les créations de fermetures provenant d'une même λ-expression. La partie commune de la structure peut être compilée une fois pour toute et utilisée, telle une constante, par chaque fermeture. Dans ce but, la partie commune prend la forme d'une fonction (la *fonction commune*) et la partie qui varie d'une fermeture à l'autre est la *fonction d'interface*. La fonction commune est exprimée par une ε-expression qui a le même corps que la λ-expression à traduire et dont la liste de paramètres est la concaténation de la liste de paramètres de la λ-expression avec la liste des variables fermées qu'elle utilise. Par exemple, si la λ-expression **(lambda** ($v_1 \ldots v_p$) corps) utilise les variables fermées $f_1 \ldots f_q$, alors la fonction commune correspondante est exprimée par l'ε-expression suivante:

(epsilon ($v_1 \ldots v_p$ $f_1 \ldots f_q$) corps)

Cette transformation a comme effet de localiser les variables fermées. Les références aux variables fermées contenues dans le corps sont dorénavent des références à des variables locales. Le rôle de la fonction d'interface est double: elle mémorise les valeurs qu'avaient les variables fermées $f_1 \ldots f_q$ au moment de la création de la fermeture et agit comme intermédiaire entre l'appelant de la fermeture et la fonction commune. L'ε-expression correspondant à la fonction d'interface a la même liste de paramètres que la λ-expression à traduire et son corps effectue l'application de la fonction commune. Les arguments de cette application sont ceux reçus par la fonction d'interface augmentés des valeurs des variables $f_1 \ldots f_q$ qu'elle a mémorisées. La création

d'une fermeture consiste en la compilation de la structure correspondant à cette ε-expression. Ainsi, la λ-expression **(lambda (**$v_1 \ldots v_p$**)** corps**)** est traduite en:

```
(compile
  `(epsilon (v1…vp)
     (',(epsilon (v1…vp f1…fq) corps) v1…vp ',f1 … ',fq)))
```

L'application de cette transformation sur la fonction **adder** décrite précédemment donne:

```
(define (adder x)
  (compile `(epsilon (y)
              (',(epsilon (y x) (+ x y)) y ',x))))
```

La fonction commune est compilée une seule fois (lors de la compilation du programme) et est partagée par toutes les fermetures créées. La seule partie qui est recompilée à chaque création de fermeture est la fonction d'interface qui est généralement de taille réduite comparée à la fonction commune. La prochaine section analyse cette méthode plus en profondeur.

2.2.1.3 Aspect interne de notre méthode

Afin de mieux expliquer notre méthode d'implantation des fermetures, voyons quelle est la forme du code généré pour une fermeture. Pour simplifier la description, nous supposons, dans un premier temps, que le code est pour une machine à pile hypothétique avec la convention d'appel de fonction suivante:

- à l'entrée d'une fonction à **n** paramètres, les **n** emplacements au dessus de la pile contiennent les **n** arguments (le dernier argument est au dessus de la pile)
- la continuation de la fonction (i.e. l'adresse de retour) se trouve, sur la pile, directement en dessous des arguments

Donc, à l'entrée de la fonction d'interface, les **p+1** emplacements au dessus de la pile contiennent la continuation et les valeurs des variables $v_1 \ldots v_p$. Par exemple, à l'entrée de la fermeture retournée par l'appel **(adder 3)**, la pile a la structure indiquée à la figure 2.4.

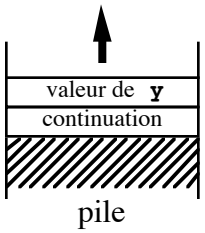


Fig 2.4: Structure de la pile à l'entrée de la fonction d'interface

De son côté, la fonction commune s'attend à recevoir sa continuation et les arguments correspondant aux variables $v_1 \ldots v_p$ et $f_1 \ldots f_q$ dans les **p+q+1** emplacements au dessus de la pile. Cependant, puisque l'application de la fonction commune est la dernière dans le corps de la fonction d'interface, la continuation de la fonction commune est identique à la continuation de la fonction d'interface. Ainsi, la fonction d'interface a peu de choses à faire: il suffit d'empiler les valeurs correspondant aux variables $f_1 \ldots f_q$ et de brancher à la fonction commune. Le code généré pour la fonction d'interface, et conséquemment pour une fermeture, a donc l'aspect suivant:

EMPILER <valeur qu'avait la variable f_1 lorsque ce bout de code a été généré>
...
EMPILER <valeur qu'avait la variable f_q lorsque ce bout de code a été généré>
BRANCHER <adresse de la fonction commune>

Le code généré pour la fermeture qui résulte de l'appel **`(adder 3)`** est donc:

EMPILER 3
BRANCHER `(epsilon (y x) (+ x y))`

et, à l'entrée de la fonction commune, la pile a la structure de la figure 2.5.

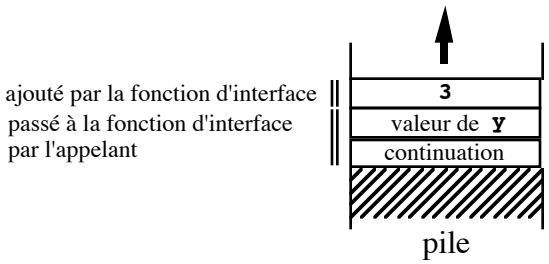


Fig 2.5: Structure de la pile à l'entrée de la fonction commune

Lorsque la variable **`x`** est référencée dans la fonction commune, le nombre **`3`** est obtenu. À la fin de l'exécution de la fonction commune (où, plus précisément lorsque la fonction **+** a terminée son calcul), les deux emplacements au dessus de la pile sont désalloués et l'exécution se poursuit à l'adresse spécifiée par la continuation.

La méthode n'est pas exclusive aux machines à pile. Il est facile d'étendre le même raisonnement à d'autres conventions d'application de fonction. Par exemple, pour une architecture à base de registres qui passe les arguments d'une fonction à **n** paramètres dans les registres

$\mathbf{R_1 \ldots R_n}$ et la continuation dans le registre $\mathbf{R_0}$, le code suivant pourrait être généré pour une fermeture:

CHARGER $\mathbf{R_{p+1}}$,<valeur qu'avait la variable f_1 lorsque ce bout de code a été généré>
…
CHARGER $\mathbf{R_{p+q}}$,<valeur qu'avait la variable f_q lorsque ce bout de code a été généré>
BRANCHER <adresse de la fonction commune>

D'une façon ou d'une autre, le code généré pour une fermeture est très simple et régulier. Ceci est dû au fait qu'il provient de la compilation d'une ε-expression de structure régulière. Ainsi, dans notre méthode, toute la généralité offerte par le compilateur n'est pas nécessaire. Une fonction de compilation spécialisée pourrait être utilisée à sa place. Afin de générer le code approprié, cette fonction, que nous appellerons **`closure`**, a besoin des informations suivantes: la fonction commune et la valeur des variables fermées $f_1 \ldots f_q$. À l'aide de la fonction **`closure`**, la λ-expression **`(lambda (`**$v_1 \ldots v_p$**`) corps)`** se traduit en:

`(closure` $f_1 \ldots f_q$ **`(epsilon (`**$v_1 \ldots v_p$ $f_1 \ldots f_q$**`) corps))`**

2.2.1.4 Traitement de l'affectation

Jusqu'à présent, nous n'avons pas considéré l'affectation aux variables fermées. Si notre méthode n'était pas modifiée, une affectation à une variable fermée changerait seulement la valeur de la variable locale correspondante. En fait, l'affectation à une variable fermée doit influencer de façon permanente toutes les fermetures qui partagent cette variable. Ce comportement est essentiel, par exemple, pour implanter des abstractions de données mutables à l'aide de fermetures (voir [Abel85a]).

Nous appellerons *variable mutable*, une variable fermée qui apparait dans une affectation. Une telle variable peut être implantée en utilisant un emplacement auxiliaire pour emmagasiner la valeur de la variable. À l'entrée d'une fonction qui déclare une variable mutable, une *boîte* (i.e. un enregistrement pouvant contenir une seule donnée) est allouée du monceau et l'argument correspondant à cette variable est déposé dans la boîte. Ainsi, les opérations de référence et d'affectation à une variable mutable consistent respectivement à retirer et à déposer une valeur dans la boîte associée à la variable. Les fermetures qui partagent une variable mutable partageront, en fait, la boîte qui lui est associée.

Des modifications mineures à la méthode de traduction de λ-expressions précédemment décrite sont nécessaires pour traiter l'affectation correctement. Des boîtes sont créées et initialisées, grâce à la fonction **`box`**, à l'entrée des fonctions qui déclarent des variables mutables. Afin de pouvoir les utiliser, l'adresse de chaque boîte est sauvée dans une variable temporaire, de même

nom, déclarée au moyen d'une ε-expression au début de la fonction. Les références et les affectations à ces variables sont transformées repectivement en applications des fonctions **fetch** et **store**. De plus, lorsqu'une fermeture qui utilise une variable mutable est créée, la valeur mémorisée par la fonction d'interface est l'adresse de la boîte correspondante, et non pas son contenu. Par exemple, l'expression:

```
(define (tally x)
  (lambda (y)
    (set! x (+ x y))
    x))
```

est traduite en l'expression suivante:

```
(define (tally x)
  ((epsilon (x)
     (closure
       x
       (epsilon (y x)
         (store x (+ (fetch x) y))
         (fetch x))))
   (box x)))
```

Les fonctions **box**, **fetch** et **store** peuvent être définies en Scheme à partir des doublets:

```
(define (box v)     (list v))
(define (fetch b)   (car b))
(define (store b v) (set-car! b v))
```

Les variables mutables sont donc légèrement plus coûteuses que les autres variables puisqu'elles occupent un plus de mémoire et une indirection supplémentaire est nécessaire pour les accéder. Ceci ne constitue pas un inconvénient majeur surtout si l'on considère que le style de programmation préconisé dans les langages applicatifs comme Scheme décourage l'utilisation d'effets de bords.

2.2.1.5 Algorithme de traduction

Le procédé de traduction de λ-expressions est constitué de deux passes. La première passe consiste à balayer l'expression à traduire et à construire une structure de graphe qui la représente. Chaque noeud du graphe représente une sous-expression de l'expression à traduire et contient des informations additionnelles qui sont calculées lors de cette passe. En particulier, les variables sont représentées par une structure qui indique si elle est globale, locale ou fermée et si elle est utilisée dans une affectation. Pendant la première passe une structure qui décrit l'environnement est

conservée. Celle-ci permet de retrouver la structure qui représente une variable particulière. Les informations d'une variable sont mises à jour à chaque référence ou affectation de variable de l'expression examinée. Une variable déclarée dans une λ-expression est initialement locale puis devient fermée si elle est accédée dans une autre λ-expression. Similairement, les variables sont initialement *non-affectées* et deviennent *affectées* si une affectation à cette variable est trouvée. De plus, chaque noeud représentant une λ-expression contient la liste des variables fermées qui sont utilisées dans son corps. Cette liste est initialement vide et est augmentée à chaque fois qu'un accès à une variable fermée est trouvée dans son corps.

La deuxième passe consiste à balayer la structure de graphe obtenue et de reconstruire l'expression Scheme qui lui correspond en tenant compte des informations accumulées. Il existe trois cas particuliers où la portion d'expression reconstruite n'est pas identique à celle de l'expression originale. Le premier est une référence à une variable fermée qui est aussi utilisée dans une affectation. Il faut alors produire l'expression **`(fetch var)`** au lieu de `var`. Le deuxième cas est une affectation à une variable fermée. Il faut alors produire l'expression **`(store var val)`** au lieu de **`(set! var val)`**. Finalement, dans le cas d'une λ-expression qui utilise des variables fermées, il faut produire l'expression **`(compile `(epsilon …))`** ou **`(closure … (epsilon …))`** tel qu'expliqué précédemment. Il faut aussi créer les boîtes à l'entrée des λ-expressions qui déclarent des variables mutables.

Nous avons implanté le procédé de traduction en Scheme. Celui-ci, qui a une longueur d'à peu près 150 lignes, ne traite que les formes spéciales de base de Scheme, puisque les autres formes spéciales peuvent êtres implantées à l'aide de ces dernières. La version qui produit des formes contenant des ε-expression et des appels à la fonction **`compile`** est donnée en appendice A avec plusieurs exemples de traductions.

2.2.2 Analyse de la performance

La performance générale d'un système Scheme dépend de plusieurs paramètres tels que la représentation interne des données, la fréquence d'utilisation des diverses constructions du langage, le contexte dans lequel elles sont utilisées et une foule d'autres paramètres qui n'ont pas un rapport intime avec la méthode d'implantation des fermetures. Notre but n'est pas d'étudier l'influence de tous ces paramètres sur la performance d'un système Scheme. Nous nous limitons aux aspects qui ont un rapport direct avec les fermetures.

Deux aspects importants à analyser sont l'espace mémoire et le temps d'exécution. En ce qui concerne les fermetures, ces deux aspects se subdivisent comme suit:

- espace mémoire:
    - occupé par fermeture
    - occupé par le code de création de fermeture
    - occupé par le code d'application de fonction
    - occupé par le code d'accès aux variables
- temps:
    - pour la création de fermeture
    - pour l'application de fonction
    - pour l'accès aux variables
    - supplémentaire associé à l'opération de *garbage collection*

Afin de mieux apprécier ses mérites, nous procédons à une analyse comparative entre notre méthode et la méthode classique d'implantation de fermetures.

Avant tout, nous définissons plus précisément la méthode classique. Elle consiste à représenter une fermeture comme une structure de donnée à deux champs. Le premier champ pointe sur le code de la fermeture et le deuxième sur l'environnement de définition. L'environnement est représenté par une chaîne d'enregistrements qui ne contiennent que les variables fermées. Les variables locales sont allouées sur la pile d'exécution et les variables globales sont alloués dans les symboles dénotant ces variables. Ainsi, il n'y a pas d'enregistrement associé à l'environnement global dans la chaîne. C'est le code de la fermeture qui est responsable de l'allocation de ses variables fermées à l'avant de la chaîne d'enregistrements. Comme optimisations supplémentaires, le dernier enregistrement de la chaîne ne contient pas de lien statique et les enregistrements vides ne sont jamais placés à l'avant de la chaîne (ce qui ne ferait qu'augmenter le temps d'accès aux variables). Tout au long de l'exécution, un pointeur global (e.g. un registre) pointe sur l'environnement courant.

La convention d'application de fonctions est similaire pour les deux méthodes et consiste à empiler l'adresse de retour sur la pile d'exécution suivie de chacun des arguments de l'application. La différence majeure réside dans la façon de passer le contrôle à la fermeture. Dans le cas d'une fermeture créée grâce à notre méthode ceci consiste simplement à y effectuer un branchement. Aucun pointeur d'environnement, autre que le pointeur de pile, n'a à être manipulé. Dans le cas des fermetures créées grâce à la méthode classique il faut décortiquer le doublet code-environnement, se placer dans l'environnement de définition de la fermeture appelée, puis effectuer le branchement au code. On peut donc s'attendre à ce que le nombre d'instructions ainsi que le temps d'exécution soit plus faible pour l'application de fermetures implantées à l'aide de

notre méthode.

De plus, lorsque l'application est non-terminale (i.e. lorsqu'elle est effectuée pour évaluer un argument d'une autre application), ce qui est une situation fréquente, la méthode classique doit sauver l'environnement de l'appelant sur la pile avant l'application et le restaurer à son retour. Cette opération pourrait être évitée si le pointeur d'environnement courant était toujours stocké sur la pile au lieu d'être global mais ceci augmenterait le coût de chaque accès aux variables fermées. Cette opération n'est pas nécessaire pour notre méthode qui alloue tout sur la pile.

Également, si toutes les fonctions sont représentées de la même façon (i.e. par un doublet code-environnement), l'application de fonctions primitives, qui ne possèdent pas d'environnement de définition, effectuera le traitement général même si cela n'est pas vraiment nécessaire. Par contre, si les fonctions primitives ne sont pas représentées de la même façon, il sera nécessaire de les différencier à chaque application puisqu'on ne sait pas, à la compilation, quelle sorte de fonction sera appelée. Ceci n'est pas le cas pour notre méthode puisque les fonctions primitives et les fermetures sont représentées par un bout de code et dans chaque cas, l'application consiste simplement en un branchement. Cet aspect est important surtout si l'on considère que l'application de fonctions primitives est de loin la plus fréquente dans les programmes.

Comme nous l'avons indiqué précédemment notre méthode d'implantation de fermetures permet un accès rapide aux variables fermées. Un accès à une variable fermée non-mutable consiste en une indirection avec déplacement par rapport au pointeur de pile. Pour une variable mutable il suffit de faire une indirection supplémentaire. Dans le cas de la méthode classique, l'accès à une variable fermée quelconque consiste en une indirection avec déplacement par rapport au pointeur vers l'enregistrement qui contient cette variable. Ce pointeur est obtenu en descendant la chaîne jusqu'au bon niveau. Si la variable fermée est dans le **n**ième enregistrement du début de la chaîne, **n-1** indirections sont nécessaires. Ces indirections peuvent être réduite à une indirection avec déplacement si l'environnement est représenté grâce à un *display* d'enregistrements. Ainsi, le seul moment où notre méthode requiert plus d'instructions et de temps pour effectuer un accès que la méthode classique est dans le cas de l'accès à une variable mutable se trouvant dans le premier enregistrement de la chaîne, la différence étant d'une indirection supplémentaire. Dans les autres cas, notre méthode est au moins aussi bonne que la méthode classique et de mieux en mieux plus la profondeur de l'enregistrement contenant la variable augmente.

L'espace mémoire occupé par fermeture est plus difficile à analyser. La difficulté provient en partie du fait que certaines portions des fermetures peuvent être partagées entre plusieurs fermetures. De plus, il faut considérer le supplément d'espace associé aux conventions de représentation des données en mémoire (e.g. bit de marquage pour l'opération de *garbage collection*, type de la donnée, longueur, etc...). Pour simplifier les calculs, nous supposerons que

ce supplément est nul et nous ne tiendrons pas compte de l'espace occupé par le corps de la fermeture. Si on examine notre méthode, l'espace mémoire occupé par fermeture dépend du nombre de variables fermées mutables ($\mathbf{N_M}$) et non-mutables ($\mathbf{N_N}$) utilisées par la fermeture, de la taille des codes d'opération pour les instructions **EMPILER** ($\mathbf{E_E}$) et **BRANCHER** ($\mathbf{E_B}$) et de la taille des pointeurs de données ($\mathbf{E_P}$). L'espace total occupé par une fermeture créée par notre méthode est donc:

$$\mathbf{T_N} = (\mathbf{N_M}+\mathbf{N_N})*(\mathbf{E_E}+\mathbf{E_P}) + \mathbf{E_B} + \mathbf{E_P} + \mathbf{N_M}*\mathbf{E_P}$$

Il est à remarquer que le dernier terme de cette équation correspond aux boîtes associées aux variables mutables de la fermeture qui peuvent être partagées entre plusieurs fermetures. Pour la méthode classique, l'espace mémoire occupé par fermeture dépend du nombre de variables fermées dans la chaîne d'enregistrements ($\mathbf{N_F}$), du nombre d'enregistrements dans la chaîne ($\mathbf{N_E}$) et de $\mathbf{E_P}$. L'espace total occupé par une fermeture créée par la méthode classique est donc:

$$\mathbf{T_C} = 2*\mathbf{E_P} + \mathbf{N_F}*\mathbf{E_P} + (\mathbf{N_E}-1)*\mathbf{E_P}$$

Ici, les deux derniers termes de l'équation correspondent à l'espace occupé par l'environnement qui peut être partagé par plusieurs fermetures. Il est bon de signaler que l'environnement peut contenir une quantité arbitraire (**X**) de variables fermées qui ne sont pas utilisées par la fermeture (i.e. $\mathbf{N_F} = \mathbf{N_M}+\mathbf{N_N}+\mathbf{X}, \mathbf{X} \geq 0$). Ce cas, qui survient lorsque plusieurs fermetures utilisant des variables fermées différentes sont créées à partir du même environnement, complique la comparaison entre les deux méthodes.

Le cas le plus favorable pour la méthode classique survient lorsqu'un grand nombre de fermetures partagent le même environnement. À ce moment, pratiquement aucun espace de plus que le doublet code-environnement (i.e. $2*\mathbf{E_P}$) n'est requis par fermeture. Le pire cas survient lorsque **X** est grand, la chaîne est longue et l'environnement de définition n'est pas partagé.

Afin de mesurer et de comparer la performance des deux méthodes, nous avons effectués quelques tests avec chaque méthode. Nous avons traduits trois définitions de fonctions Scheme qui effectuent des créations de fermetures en assembleur MC68000 en utilisant chacune des méthodes d'implantation de fermetures. Ces fonctions, l'appel effectué ainsi que les fermetures créées lors de l'appel sont données à la figure 2.6. La fonction **`f`** à trois arguments utilisée dans les définitions est une fonction qui retourne le troisième argument. Pour chaque définition de fonction nous avons mesuré quatre caractéristiques: l'espace occupé par le code MC68000 correspondant à la fonction, l'espace alloué pour les fermetures lors de l'appel de la fonction, le temps requis pour exécuter la fonction et le temps requis pour exécuter le résultat de l'appel de la fonction (qui est dans chaque cas une fermeture à aucun argument). Les résultats de ces tests

exécutés sur un Macintosh Plus sont donnés à la table 2.1.

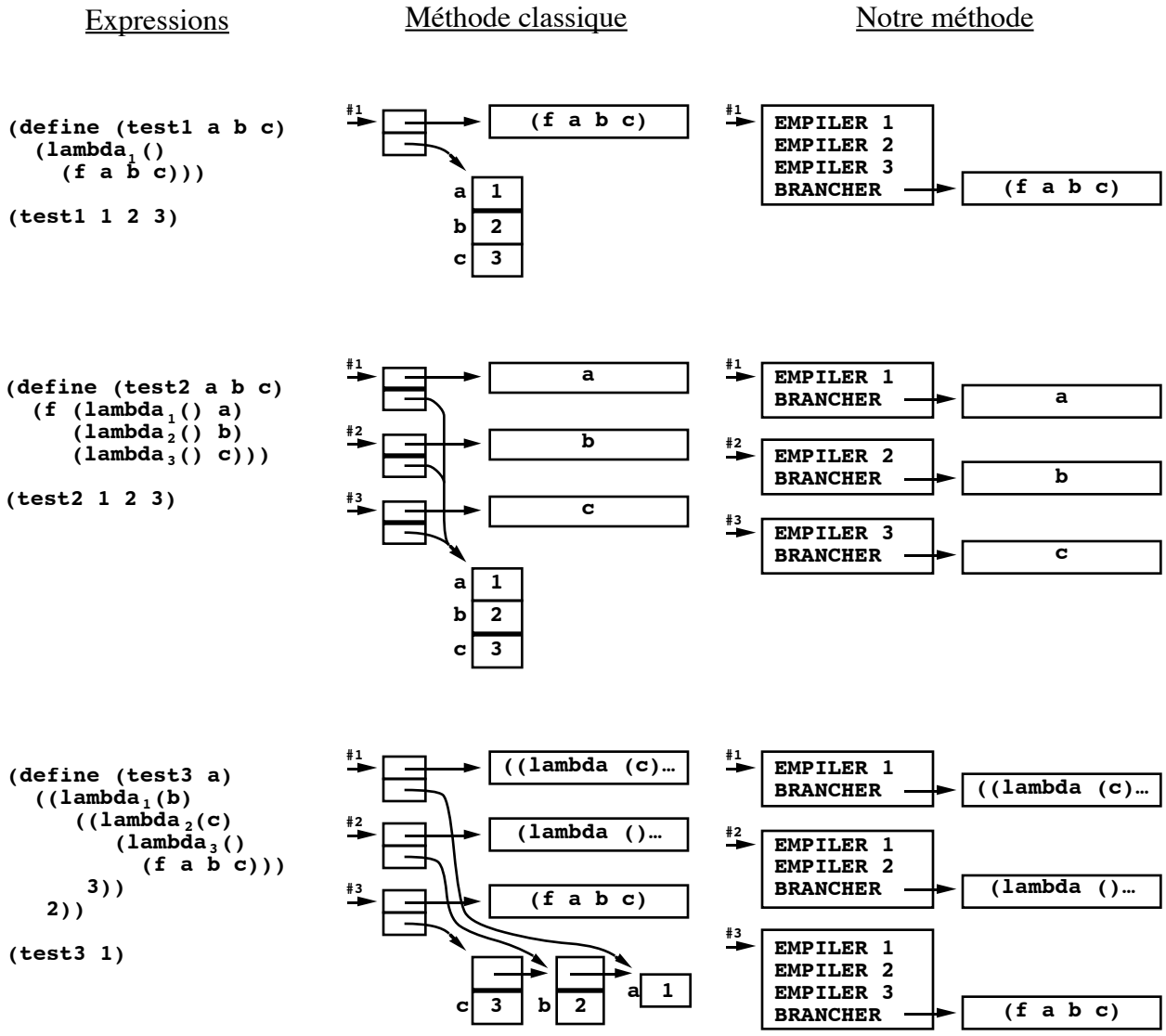


Figure 2.6: Exemples de fermetures créées par la méthode classique et par notre méthode

Méthode et appel	espace occupé par la fonction	espace occupé par les fermetures	temps pour l'appel	temps pour l'appel de la fermeture résultante
Notre méthode:				
(test1 1 2 3)	128 octets	24 octets	50 µsec	40 µsec
(test2 1 2 3)	164 octets	36 octets	80 µsec	21 µsec
(test3 1)	170 octets	54 octets	101 µsec	40 µsec
Méthode classique:				
(test1 1 2 3)	152 octets	20 octets	51 µsec	51 µsec
(test2 1 2 3)	180 octets	36 octets	81 µsec	25 µsec
(test3 1)	192 octets	44 octets	80 µsec	56 µsec

Table 2.1: Résultats des tests pour notre méthode et la méthode classique d'implantation de fermetures

L'espace occupé par les fermetures peut être calculé à partir des informations suivantes: pour le MC68000, $\mathbf{E_P}$ = 4 octets et $\mathbf{E_E}$ = $\mathbf{E_B}$ = 2 octets (car les instructions **EMPILER** et **BRANCHER** correspondent respectivement à **`MOVE.L #pnt,-(SP)`** et **`JMP pnt.L`**). Dans le cas des tests avec notre méthode, nous avons directement inclus le code de création de fermetures dans les fonctions plutôt que d'appeler la fonction **`compile`** ou **`closure`**. Ceci a allongé le code des fonctions et diminué le temps de création des fermetures.

On peut tirer les observations suivantes des résultats. Premièrement, les fonctions implantées à l'aide de notre méthode sont de 10-15% plus courtes et, sauf une exception discutée plus loin, aussi rapide que les fonctions implantées à l'aide de la méthode classique. Puisque ces fonctions exercent en grande partie la création de fermetures, ces observations sont attribuables aux différences du point de vue de la création de fermetures.

Deuxièmement, dans les tests effectués, l'espace occupé par les fermetures créées à l'aide de notre méthode est jusqu'à 25% de plus que l'espace occupé par les fermetures créées par la méthode classique. Néanmoins, si on ne considère que les fermetures actives après l'exécution des fonctions (i.e. la dernière fermeture créée par chacune des fonctions), l'espace occupé est jusqu'à 40% de moins dans le cas de notre méthode. Les fermetures créées par notre méthode ont l'avantage de ne retenir que les variables fermées nécessaires à leur exécution mais cependant, ne partagent pas les variables fermées entre plusieurs fermetures. Ainsi, l'espace occupé dépend beaucoup du contexte et, de ce point de vue, aucune des deux méthodes n'est meilleure que l'autre dans tous les contextes.

Finalement, l'appel des fermetures résultant de l'appel des fonctions, qui teste la référence de variable fermée et l'application de fonction, est de 15-30% plus rapide dans le cas de notre méthode.

Les résultats sont donc favorables pour notre méthode d'implantation de fermetures. Cependant, notre méthode a des faiblesses cachées tel que démontré par l'exécution de la fonction **`test3`** qui est 25% plus lente que lorsqu'elle est implantée par la méthode classique. Cette fonction contient des λ-expressions fortement imbriquées et la plus interne utilise des variables qui sont fermées par rapport aux λ-expressions englobantes. Dans cette situation, les fermetures créées pour les λ-expressions englobantes doivent mémoriser ces variables fermées pour pouvoir créer les fermetures qu'elles englobent. Ceci demande plus de temps pour créer les fermetures et pour empiler les valeurs sur la pile lors de l'appel des fermetures. Dans la fonction **`test3`**, c'est seulement la dernière fermeture créée qui utilise les variables fermées **`a`**, **`b`** et **`c`**. L'effort dépensé à empiler la valeur des variables fermées sur la pile afin d'améliorer leur temps d'accès n'est donc pas vraiment utile car aucune des variables fermées n'est accédée lors de l'appel de la fonction **`test3`**.

Cette faiblesse nous suggère une analogie entre notre méthode et une cache [Baer80]. À l'entrée d'une fermeture la valeur des variables fermées est copiée sur la pile. Les valeurs se trouvent à un endroit qui peut être accédé efficacement. Bien que la copie des valeurs coûte en temps, il est espéré que ces valeurs seront accédées fréquemment et qu'ainsi le temps d'exécution global sera diminué. Si plusieurs valeurs sont copiées mais que celles-ci sont rarement accédées, le temps de copie risque de surpasser le temps sauvé. Dans ce cas, une méthode hybride peut être préférable. Celle-ci consiste à représenter les environnements de la même façon que la méthode classique mais de produire le code suivant pour une fermeture:

EMPILER <pointeur vers l'environnement de définition>
BRANCHER <adresse du corps de la fermeture>

Cette méthode d'implantation de fermetures a des caractéristiques similaires à la méthode classique sauf que les fermetures occupent un peu plus d'espace (i.e. $\mathbf{E_E+E_B}$) et sont appelées en effectuant un branchement. Le gros atout sur la méthode classique est que l'application de fonctions ayant un environnement de définition nul (e.g. les fonctions primitives) n'entraine pas de traitement supplémentaire pour décortiquer le doublet code-environnement.

L'avantage des langages à portée statique, tel que Scheme, est que tous les accès à une variable sont situés à l'intérieur de la construction qui déclare cette variable. Conséquemment, le compilateur peut choisir, avec une analyse locale du programme, la façon qu'il juge la meilleure pour représenter les environnements et les fermetures. Notre méthode standard pourrait être utilisée dans certaines situations et la méthode hybride dans les autres. Le prérequis étant que la convention d'application de fonction soit identique pour chaque sorte de représentation, ce qui est le cas de notre méthode standard et hybride.

2.2.3 Simulation des fermetures

Tel que vu dans les sections précédentes, notre méthode peut être utilisée par un compilateur pour implanter les fermetures efficacement. Ce n'est pas sa seule utilité. On peut appliquer la même idée pour simuler les fermetures dans les dialectes de Lisp qui n'ont pas la possibilité de fermetures.

Dans la majorité des dialectes de Lisp sans fermetures, les λ-expressions sont l'équivalent des ε-expressions. De plus, la fonction **`eval`**, lorsqu'elle est appliquée sur une structure représentant une définition de fonction (i.e. une λ-expression), est équivalente à la fonction **`compile`**. Ainsi, en Franz-Lisp [Fode82], l'expression suivante:

```
(eval '(function (lambda …)))
```

est équivalente à:

```
(compile '(epsilon …))
```

En utilisant ces équivalences, il est possible de traduire la fonction **adder**, donnée en exemple précédemment, en Franz-Lisp de la façon suivante:

```
(defun adder (x)
  (eval
    `(function (lambda (y)
                 (',(function (lambda (y x) (+ x y)))
                  y
                  ',x)))))
```

Évidemment, l'usager ne devrait pas être obligé d'entrer cette forme. Il serait préférable que la traduction s'effectue automatiquement grâce à un macro. Par exemple, **lambda** pourrait être redéfini comme étant un macro effectuant la traduction. À ce moment, les formes débutant par le mot clé **lambda** auraient la même sémantique que les λ-expressions de Scheme. L'évaluation d'une λ-expression produirait l'équivalent d'une fermeture sans qu'il soit nécessaire de l'entourer d'une forme spéciale **function**.

Chapitre III

La génération de code

Il y a deux façon traditionnelles d'implanter le langage Lisp: à l'aide d'un interpréteur ou d'un compilateur. Chaque méthode a ses avantages par rapport à l'autre. Souvent les implantations à base d'interpréteur sont plus simples à développer, fournissent de meilleures informations de mise au point de programmes et sont facilement portables. D'un autre côté, les implantations à base de compilateur exécutent les programmes beaucoup plus rapidement ce qui est utile lorsque la vitesse d'exécution est importante. À titre de référence, un interpréteur Scheme écrit en Scheme est donné en appendice B.

Ce chapitre décrit une technique originale de compilation qui possède les avantages d'un interpréteur tout en offrant une exécution rapide. La génération de code est entièrement basée sur la création de fermetures. Puisque le langage Scheme permet de créer et de manipuler des fermetures, il nous a été possible d'écrire un compilateur Scheme totalement en Scheme.

Les deux premières sections décrivent un compilateur simple qui utilise notre méthode de génération de code et indiquent comment chacune des formes primitives de Scheme peut être compilée. Celles-ci sont suivies d'un exemple complet de compilation à l'aide de notre méthode. Nous indiquons ensuite les améliorations qui peuvent être apportées à la méthode de base pour en améliorer l'efficacité. Finalement, nous discutons des autres applications de notre méthode et nous analysons la performance de notre méthode de compilation.

3.1 Notre méthode de génération de code

Le compilateur est une fonction à un argument nommée **`compile`**. Elle accepte comme argument la représentation sous forme de donnée d'une λ-expression Scheme et retourne la fonction qui implante cette λ-expression dans l'environnement global. L'évaluation de l'expression **`(compile '(lambda …))`** résulte en une fonction équivalente à celle obtenue par l'évaluation de l'expression **`(lambda …)`** au *top-level*. La fonction **`compile`** est donc similaire à celle décrite au deuxième chapitre sauf qu'elle est plus générale puisqu'elle compile des λ-expressions. Ce compilateur figure à l'appendice C.

L'idée de base de notre méthode consiste à créer une fermeture élémentaire pour chaque construction primitive contenue dans l'expression à compiler. Ces fermetures ont la propriété que, lorsqu'elles sont appelées, elles effectuent l'évaluation de la partie correspondante de l'expression originale. Les fermetures sont utilisées car elles peuvent retenir, au moyen de leurs variables

fermées, les informations nécessaires pour paramétriser leur comportement. Par exemple, la fermeture créée pour une forme spéciale **`quote`** est la valeur de la constante. Puisque l'environnement d'évaluation varie d'une exécution à l'autre, celui-ci est passé en paramètre à chacune des fermetures créées lorsqu'elles sont appelées. L'environnement est utilisé par les fermetures correspondant aux constructions de référence et d'affectation de variable pour retrouver l'emplacement qui est couramment associé à une variable. On remarque ici un phénomène intéressant: les informations qui sont nécessaires à l'évaluation des constructions primitives et qui sont constantes d'une évaluation à l'autre sont mémorisées dans les variables fermées des fermetures correspondantes et les informations qui varient sont passées en argument aux fermetures au moment de leur exécution. Le compilateur peut tirer profit de ce phénomène en effectuant les traitements invariables au moment de la compilation afin de réduire la quantité d'information à manipuler à l'exécution accélérant ainsi l'exécution du code.

Le coeur du compilateur est une fonction à un argument nommée **`gen`**. Celle-ci accepte la représentation sous forme de donnée d'une expression et retourne la fermeture qui effectue l'évaluation de cette expression lorsqu'elle est appelée. L'expression est classifiée selon la construction primitive qui la constitue et la fonction de génération de code correspondante est appellée avec les informations nécessaires pour paramétriser la fermeture résultante. Dans le cas des constructions primitives composées de sous-expressions, certains des paramètres proviennent de l'appel récursif de la fonction **`gen`** sur les sous-expressions en question. La compilation d'une expression produit donc un réseau de fermetures. Les liens de ce réseau sont mémorisés dans les variables fermées des fermetures et sont suivis lorsque la fermeture correspondante est appelée. Le code ainsi obtenu a une forte ressemblance avec celui habituellement produit pour les *threaded langages* qui forment aussi un réseau de code [Klin81] [Loel81].

Comme on pouvait s'y attendre, le compilateur possède une structure similaire à un interpréteur. Les deux acceptent des représentations d'expressions, les classifient et procèdent à leur balayage récursif dans le cas des primitives composées. La différence majeure est que l'interpréteur effectue l'évaluation de l'expression aussitôt qu'elle est reconnue. De son côté, le compilateur produit uniquement le code qui permet d'évaluer l'expression. L'évaluation comme telle s'effectue seulement lorsque le code est appelé. L'interpréteur doit donc classifier l'expression à chaque fois qu'elle est évaluée. Ceci n'est pas le cas du compilateur qui effectue cette classification une fois pour toute au moment de la compilation. Cet aspect est à l'origine du gain de vitesse d'exécution attribué aux systèmes à base de compilateur car il n'est pas nécessaire de reconnaître l'expression à l'exécution. De plus, le compilateur peut reconnaître, à la compilation, une forme particulière d'évaluation qui peut être traîtée plus efficacement que si elle était compilée de façon générale. Cet avantage ne s'applique pas pour les interpréteurs car il

faudrait détecter ces situations à chaque exécution, ralentissant ainsi le traitement des cas ordinaires.

3.2 Fonctions de génération de code

Dans les sections qui suivent nous décrivons les fonctions de génération de code pour chacune des constructions primitives de Scheme. Tel que vu au premier chapitre, il n'est pas nécessaire de toutes les considérer. On peut se limiter aux constructions de référence de variable, d'application de fonction et aux formes spéciales de base **`quote`**, **`set!`**, **`if`** et **`lambda`** en supposant que l'expression passée à la fonction **`compile`** a déjà subi l'étape d'expansion des macros.

L'implantation de ces fonctions dépend en partie de la représentation des environnements. Pour simplifier la description des fonctions de génération de code, nous utiliserons, dans un premier temps, une liste d'associations symbole-valeur pour représenter les environnements. Plus loin, nous envisagerons d'autres méthodes plus efficaces et indiquerons les changements nécessaires aux fonctions de génération de code.

3.2.1 Forme spéciale **`quote`**

Le code de la forme spéciale **`quote`** est généré par la fonction suivante:

```
(define (gen-cst a)
  (lambda (env) a))
```

L'application de cette fonction sur une donnée particulière retourne une fermeture qui, lors de son application subséquente, retourne la donnée en question. Par exemple:

```
(define code1 (gen-cst 123))
(code1 '()) ⇒ 123
```

3.2.2 Référence de variable

La référence d'une variable consiste à obtenir, à partir de l'environnement, la valeur qui est couramment associée au symbole dénotant la variable. Cette opération peut être effectuée à l'aide de la fonction **`assq`** qui cherche une association particulière dans une liste d'associations. Ainsi, le code d'une référence de variable est généré par la fonction suivante:

```
(define (gen-ref a)
  (lambda (env) (cdr (assq a env))))
```

L'argument passé à cette fonction est le symbole qui dénote la variable à référencer.

3.2.3 Forme spéciale **set!**

L'affectation à une variable consiste à changer la valeur qui est associée au symbole dénotant la variable dans l'environnement courant. De façon similaire à la référence de variable, cette opération peut être effectuée à l'aide de la fonction **assq**. Ainsi, le code d'une affectation est généré par la fonction suivante:

```
(define (gen-set a b)
  (lambda (env) (set-cdr! (assq a env) (b env))))
```

Le premier argument passé à cette fonction est le symbole qui dénote la variable à affecter. Le second argument est la fermeture qui représente le code nécessaire pour calculer la valeur à affecter et provient de l'application de la fonction **gen** sur la sous-expression correspondante. Voici un exemple d'utilisation des fonctions **gen-ref** et **gen-set**:

```
(define code2 (gen-set 'y (gen-cst 123)))
(define code3 (gen-ref 'y))
(define env '((x . 45) (y . 67) (z . 89)))

(code3 env) ⇒ 67
(code2 env) ⇒ indéfini (car le résultat de set-cdr! est indéfini)
env ⇒ ((x . 45) (y . 123) (z . 89))
(code3 env) ⇒ 123
```

3.2.4 Forme spéciale **if**

La fermeture générée pour la forme d'évaluation conditionnelle **if** est paramétrisée par le code correspondant à chacune des sous-expressions de cette forme. Si le résultat de l'appel de la fermeture correspondant à la condition est vrai alors la fermeture correspondant au conséquent est appelée sinon, la fermeture correspondant à l'alternative est appelée. Ainsi, le code d'une évaluation conditionnelle est généré par la fonction suivante:

```
(define (gen-tst a b c)
  (lambda (env) (if (a env) (b env) (c env))))
```

Chacun des arguments passés à cette fonction correspond respectivement à la fermeture produite par l'application de la fonction **gen** sur la sous-expression dénotant la condition, le conséquent et l'alternative de la forme spéciale **if**.

3.2.5 Application de fonction

La fermeture générée pour une application de fonction est paramétrisée par le code des expressions correspondant à la fonction à appliquer et à chacun de ses arguments. Puisque le nombre d'arguments varie d'une application à l'autre, nous avons choisi d'implanter la génération de code à l'aide de plusieurs fonctions, chacune correspondant à une application avec un certain nombre d'arguments. Nous aurions pu utiliser une seule fonction de génération de code basée sur la fonction primitive **`apply`** mais ceci introduirait un autre mécanisme d'application de fonction et serait plus difficile à comprendre. Ainsi, le code d'une application de fonction est généré par la famille de fonctions suivantes:

```
(define (gen-ap0 a)
  (lambda (env) ((a env))))

(define (gen-ap1 a b)
  (lambda (env) ((a env) (b env))))

(define (gen-ap2 a b c)
  (lambda (env) ((a env) (b env) (c env))))

...
```

Les arguments passés à ces fonctions correspondent respectivement à la fermeture produite par l'application de la fonction **`gen`** sur la sous-expression dénotant la fonction à appliquer et chacun de ses arguments.

3.2.6 Forme spéciale **`lambda`**

Le traitement des λ-expressions est un peu plus complexe que pour les autres constructions. L'évaluation d'une λ-expression crée une fermeture qui effectue trois opérations lorsqu'elle est appelée: elle accepte les arguments, alloue les paramètres sur l'environnement de définition de la fermeture et finalement évalue le corps de la fermeture dans ce nouvel environnement. Le code généré pour une évaluation de λ-expression est paramétrisé par le code correspondant au corps de la λ-expression et, dû à notre représentation des environnements, par les symboles dénotant les paramètres de celle-ci. L'allocation des paramètres consiste à ajouter des doublets symbole-valeur à l'avant de la liste d'association représentant l'environnement et l'évaluation du corps consiste à appeler le code correspondant avec l'environnement nouvellement construit. L'opération de création comme telle d'une fermeture est obtenue en évaluant une λ-expression dont la liste de paramètres a la même forme que celle de la λ-expression compilée. Au moment de sa création, cette fermeture mémorise, en plus des informations précédemment décrites, la valeur de l'environnement d'évaluation qui est utilisé pour les applications

subséquentes de la fermeture. Puisque la forme de la liste de paramètre n'est pas fixe d'une λ-expression à l'autre, la génération de code est effectuée par plusieurs fonctions. Ainsi, le code d'une évaluation de λ-expression est généré par la famille de fonctions suivantes:

```
(define (gen-fn0 a)
  (lambda (env)
    (lambda () (a env))))

(define (gen-fn1 a b)
  (lambda (env)
    (lambda (x) (a (cons (cons b x)
                         env)))))

...

(define (gen-fn1/rest a b)
  (lambda (env)
    (lambda x (a (cons (cons b x)
                       env)))))

(define (gen-fn2/rest a b c)
  (lambda (env)
    (lambda (x . y) (a (cons (cons b x)
                             (cons (cons c y)
                                   env))))))

...
```

Le premier argument passé à ces fonctions correspond à la fermeture produite par l'application de la fonction **gen** sur la sous-expression dénotant le corps de la λ-expression et les autres arguments correspondent aux symboles qui dénotent les paramètres de la λ-expression.

3.3 Exemple complet

Nous donnons, ici, un exemple complet de génération de code pour une expression simple. Considérons la définition suivante:

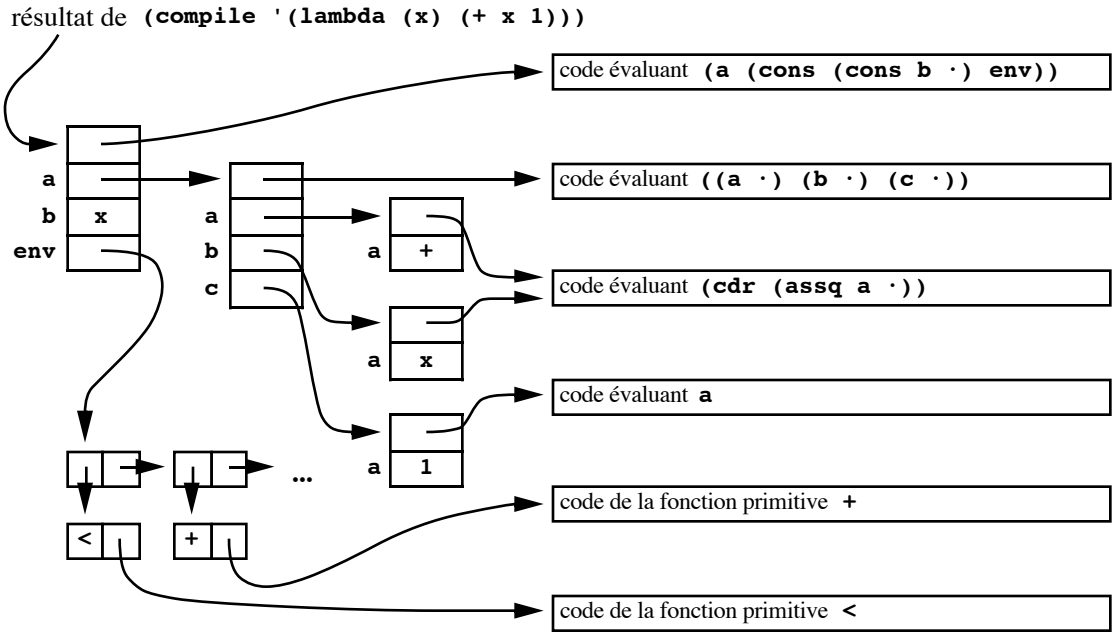
```
(define add1 (compile '(lambda (x) (+ x 1))))
```

Cette définition affecte à la variable globale **add1** une fonction qui retourne un de plus que son argument. Le balayage de la liste **(lambda (x) (+ x 1))** par la fonction

compile consiste à appeler les diverses fonctions de génération de code et correspond à la définition étendue suivante:

```
(define add1 ((gen-fn1
                (gen-ap2
                  (gen-ref '+)
                  (gen-ref 'x)
                  (gen-cst 1))
                'x)
              *env-glo*))
```

La variable ***env-glo*** contient la liste d'association qui représente l'environnement global. Au moment de l'application de la fonction associée à **add1**, l'environnement global doit au moins contenir la variable **+** puisque celle-ci est référencée dans le corps de la fonction. Afin de mieux décrire le code généré, nous utilisons la convention que les fermetures sont représentées graphiquement à l'aide de boîtes à plusieurs champs dont le premier pointe sur le corps de la fonction et les autres correspondent à chacune des variables fermées de la fermeture. Le résultat de la fonction **compile** est indiqué à la figure 3.1.



Note: · représente l'argument passé à la fermeture

Figure 3.1: Exemple de code généré par le compilateur

Comme on peut le voir, le résultat de la compilation est un réseau de fermetures dont les arcs sont mémorisés dans les variables fermées des fermetures générées. Les variables fermées de ces fermetures sont aussi utilisées pour mémoriser d'autres valeurs qui paramétrisent leur

comportement. Il est à remarquer que la première fermeture de toutes n'est pas directement créée par une fonction de génération de code mais plutôt par l'application de la fermeture retournée par la fonction **gen-fn1** sur l'environnement global.

3.4 Améliorations

Globalement, le compilateur décrit est de la même taille et complexité qu'un interpréteur équivalent (on n'a qu'à comparer les programmes de l'appendice B et C pour s'en convaincre). Néanmoins, comme nous le verrons dans la section 3.6, les programmes compilés à l'aide du compilateur exécutent plus rapidement que s'ils avaient été évalués à l'aide d'un interpréteur. Cependant, les qualités essentielles d'un interpréteur ne sont pas perdues. En particulier, les profils d'exécution et les facilités de mise au point peuvent aisément être obtenues en ajoutant le code nécessaire dans le corps des fermetures générées. Par exemple, si l'on désire faire des statistiques sur le nombre d'affectations effectuées, on peut utiliser la fonction de génération de code suivante pour les affectations:

```
(define (gen-set/stat a b)
  (lambda (env)
    (set! nb-affect (1+ nb-affect))
    (set-cdr! (assq a env) (b env))))
```

La variable globale **nb-affect**, initialisée préalablement à zero, vaut, à la fin de l'exécution d'une expression compilée, le nombre d'affectations effectuées. Un drapeau, modifiable par l'usager, peut contrôler la génération de ces facilités offrant ainsi le meilleur des deux mondes. Bien qu'il soit déjà efficace, le compilateur décrit peut être amélioré pour augmenter sa performance.

Un premier aspect qui peut être amélioré est la représentation des environnements. La liste d'associations utilisée présentement est simple mais elle utilise plus de mémoire qu'il n'est vraiment nécessaire et un accès à une variable requiert une recherche dans la liste. Dans un premier lieu, une simple liste de valeurs pourrait être utilisée. Un accès à une variable consisterait à accéder un élément particulier de la liste à partir du début. La profondeur de la variable dans la liste serait calculée au moment de la compilation et serait mémorisée par la fermeture générée pour cet accès. Une autre possibilité pour représenter les environnements serait d'utiliser une chaîne d'enregistrements telle que décrite dans le chapitre précédent. Cette représentation a l'avantage d'utiliser encore moins de mémoire et les accès sont moins profonds que pour la représentation sous forme de liste de valeurs. Finalement, on pourrait simuler une pile à l'aide d'un grand vecteur et utiliser notre méthode d'implantation de fermetures décrite dans le chapitre précédent.

Une autre inefficacité provient du fait que l'environnement est passé en argument aux fermetures lorsqu'elles sont appelées et ce, même si l'environnement n'est pas utilisé par ces dernières. Ce supplément de travail peut être éliminé en affectant l'environnement courant à une variable globale. Cette variable est accédée librement par les fermetures qui en ont besoin. De plus, il faut sauver la valeur de cette variable à l'entrée du code généré pour une fonction et restaurer la valeur de la variable à sa sortie. Ceci est nécessaire car l'environnement d'évaluation change lors d'une application et doit revenir à son état présent pour poursuivre l'évaluation du corps de la fonction qui a effectuée l'application. En fait, pour préserver la sémantique de récursivité terminale de Scheme, si la dernière évaluation du corps de la fermeture est une application, il faut restaurer l'environnement après l'évaluation des arguments et juste avant d'effectuer le branchement au code de la fonction à appliquer. On évite ainsi de conserver des environnements inutiles plus longtemps que nécessaire ce qui permet d'exprimer les itérations sous forme d'appels récursifs sans que cela nécessite une quantité non-bornée de mémoire.

Une autre source d'optimisation provient du fait que certaines formes d'expressions sont utilisées plus souvent que d'autres. Citons deux exemples fréquents: les applications dont la fonction provient d'une variable globale et les constantes communes (e.g. **`1`** et **`#!null`**). Au lieu d'utiliser la fonction générale pour générer le code correspondant, une fonction spécialisée et efficace pourrait être utilisée. Par exemple, pour générer le code de l'évaluation des constantes **`1`** et **`#!null`**, on peut se servir de la fonction correspondante suivante au lieu d'avoir recours à la fonction plus générale **`gen-cst`**:

```
(define (gen-1)    (lambda (env) 1))
(define (gen-null) (lambda (env) #!null))
```

Comme on peut le voir, ces fonctions spécialisées de génération de code sont moins paramétrisées que la fonction générale correspondante. Conséquemment, moins de mémoire est occupé par le code ainsi généré. Cette méthode s'apparente à l'optimisation connue dans la littérature sous le nom de réduction de force des opérations [Aho77]. Cette optimisation, qu'on retrouve surtout dans les compilateurs de code natif, consiste à générer pour la forme à compiler une séquence d'instructions moins générales mais plus efficaces que les instructions qui seraient habituellement générées pour cette forme. L'avantage dans notre cas, c'est qu'on peut concevoir les fermetures à générer qui sont les plus appropriées aux situations particulières qu'on désire optimiser. Cependant, le nombre de fermetures particulières à générer croît très rapidement et cette optimisation ne vaut la peine que pour les situations particulières les plus fréquentes.

Notre méthode de génération de code permet aussi d'économiser de la mémoire en partageant du code. En effet, puisque le code est sous la forme d'un arbre et que chaque sous-arbre correspond au code des sous-expressions de l'expression compilée, on peut réutiliser

un arbre ou sous-arbre de code pour le code d'une expression identique. Ceci est possible car lorsqu'une fermeture représentant un bout de code est appelée, une continuation est passée. Celle-ci indique le point où doit se poursuivre l'exécution une fois que le bout de code est exécuté. Ainsi, il suffit simplement que chaque portion de programme partageant un même bout de code passe sa propre continuation au bout de code qui est partagé. Au lieu de générer une nouvelle fermeture à chaque fois, le compilateur peut vérifier si une fermeture avec les mêmes paramètres n'a pas été générée précédemment et la retourner directement si c'est le cas. Cette opération peut facilement s'intégrer aux fonctions de génération de code. Par exemple, la fonction **gen-ref** peut se réécrire comme suit:

```
(define *ref* '())

(define (gen-ref a)
  (if (not (assq a *ref*))
    (set! *ref*
      (cons (cons a (lambda (env) (cdr (assq a env))))
            *ref*)))
  (cdr (assq a *ref*)))
```

Dans l'exemple, la valeur de la variable globale ***ref*** est une liste d'association qui contient les fermetures précédemment créées par la fonction **gen-ref** et les symboles qui les paramétrisent. Cette liste est examinée à l'entrée de la fonction **gen-ref** et une nouvelle fermeture y est ajoutée si elle ne s'y trouve pas déjà. Finalement la fermeture associée au symbole passé en paramètre dans la liste d'association est retournée. Cette technique est donc similaire au *hash consing* [Alle78] [Goto74] [Tera75] sauf qu'elle s'applique aux fermetures plutôt qu'aux doublets. En fait, cette optimisation pourrait être intégrée au mécanisme de création des fermetures. Il s'agirait d'utiliser une table d'adressage dispersée pour conserver les fermetures. En supposant que les fermetures sont créées au moyen de la fonction **closure** décrite au chapitre précédent, cette dernière n'aurait qu'à vérifier si elle a déjà été appelée avec les mêmes arguments et à retourner la fermeture précédemment créée dans le cas affirmatif et à créer et mémoriser une nouvelle fermeture dans le cas contraire. Les sous-expressions communes à tout le programme seraient ainsi détectées automatiquement et un seul bout de code serait généré pour les évaluer.

Bien que cette méthode permet de réduire la quantité de mémoire occupé par le code, il faut se demander si le surplus de mémoire nécessaire pour mémoriser les fermetures créées et les paramètres correspondants n'est pas plus important que la mémoire sauvée par le partage du code. Ceci ne serait pas un problème important si les phases de compilation et d'exécution des programmes étaient séparées mais puisque Scheme est un langage interactif, le compilateur doit demeurer actif pendant toute la session d'utilisation par l'usager. Un compromis serait de

conserver seulement les fermetures les plus fréquemment utilisées et d'ignorer les autres. Par exemple, les fermetures générées pour une même λ-expression pourraient être mémorisées dans un vecteur de longueur fixe et une technique de remplacement du style *least recently used* (LRU) [Baer80] pourrait être utilisée pour ne garder que les fermetures fréquemment utilisées. Une autre solution serait de borner la rétention de ces informations à la période de compilation de l'expression courante ce qui limiterait la détection des expressions communes à l'expression compilée.

En examinant de plus près les fermetures générées pour une expression, on s'aperçoit que le seul code machine qui est vraiment exécuté est celui correspondant au corps des fermetures et celui des fonctions primitives (i.e. les boîtes à la droite de la figure 1). Puisque le code du corps des fermetures générées est partagé et qu'il n'y en a pas beaucoup, il est envisageable de l'écrire en langage machine (e.g. assembleur) de la même façon que le code des fonction primitives (e.g. **`car`**, **`cons`** et **`+`**) est généralement en langage machine.

L'exécution du code généré consiste en majeure partie en applications de fermetures. L'efficacité des fermetures a donc un impact important sur l'efficacité du code généré. Une méthode d'implantation efficace des fermetures telle que décrite au deuxième chapitre se justifie très bien dans ce contexte. Nous verrons dans le quatrième chapitre comment cette méthode d'implantation de fermetures et la méthode de génération de code décrite dans ce chapitre peuvent être intégrées dans un compilateur Scheme.

Les optimisations que nous avons décrites sont celles qui ont un rapport étroit avec la génération de code basée sur les fermetures. Toutefois, ce ne sont pas les seules optimisations disponibles. Les optimisations indépendentes de la forme du code généré sont toujours possibles. Citons en exemple la factorisation de calculs invariants, l'élimination d'expressions communes, la simplification d'expression et l'évaluation partielle d'expressions [Alle75] [Hara78] [Stee78b]. Les analyses de flôt de contrôle et de flôt des données sont aussi possibles [Hech77] [Jone79]. La plupart de ces optimisations peuvent être effectuées avant de générer le code et on peut les concevoir comme étant simplement des transformations source-source. Par exemple, la phase d'optimisation du compilateur Scheme écrit par Steele [Stee78b] consiste à transformer l'expression Scheme originale en expression Scheme qui peut être évaluée plus efficacement et qui lui est équivalente. On y retrouve une série de règles de transformations telles que:

```
((lambda () corps)) → corps
((lambda (… v …) corps) … a …) → ((lambda (… …) corps) … …)
(if #!true a b) → a
```

La deuxième règle consiste à éliminer le paramètre `v` et l'évaluation de l'argument correspondant si la variable `v` n'est pas utilisée dans le corps et que l'évaluation de l'argument ne

produit pas d'effet de bord. De telles optimisations pourraient être effectuées avant la génération de code que ce soit en représentant le code à l'aide de fermetures ou d'une autre façon.

3.5 Autres applications

Il est bon de noter que la méthode de génération de code décrite ne s'applique pas seulement au langage Scheme. Rien n'empêche d'exprimer les règles d'évaluation d'un autre langage à l'aide des primitives de Scheme (bien qu'il est vrai qu'il est plus simple d'exprimer l'évaluation d'expressions Scheme à l'aide des primitives de Scheme). Par exemple, un compilateur, écrit en Scheme et utilisant notre méthode de génération de code, pourrait être conçu pour le langage Pascal. Il s'agirait de choisir une représentation convenable pour les environnements, de trouver quelle fermeture devrait être générée pour chaque construction primitive du langage Pascal et d'écrire les fonctions primitives et un analyseur lexical. Bien que ce soit possible, le travail n'est pas aussi facile que d'écrire un compilateur Scheme en Scheme.

Une application plus attrayante est d'utiliser notre méthode de génération de code pour implanter les *embedded languages*, c'est à dire les langages qui coexistent avec le langage hôte et qui sont écrits à l'aide de celui-ci. Dans notre cas, le langage hôte est Scheme. Ce genre de langage se retrouve fréquemment dans le domaine de l'intelligence artificielle, en voici quelques exemples: MICRO-PLANNER [Suss71], CONNIVER [McDe73], OPS5 [Forg81] et LCF [Gord79]. Traditionnellement ces langages sont implantés sous la forme d'interpréteurs écrit en Lisp. En utilisant les mêmes techniques que celles énoncées dans ce chapitre, ces interpréteurs pourraient être remplacés par des compilateurs équivalents écrit totalement en Scheme. La taille et la complexité de ces compilateurs seraient comparable à celles des interpréteurs correspondants mais l'exécution des programmes serait sensiblement plus rapide.

Dans un même ordre d'idée, le langage d'implantation du compilateur (i.e. le langage dans lequel le compilateur est écrit) peut être autre que Scheme. Ce qui compte c'est que le langage ait la possibilité de créer des fermetures ou leur équivalent. Citons en exemple les langages T [Rees82] [Rees84], Common Lisp [Stee84] et SIMULA 67 [Dahl82]. Les langages T et Common Lisp sont similaires à Scheme et notre méthode de génération de code s'applique à ceux-ci avec de petites modifications syntaxiques seulement. Pour le langage SIMULA 67, qui est d'une toute autre nature que Scheme, il s'agit d'utiliser des instances de classes à la place des fermetures. Chaque fonction de génération de code est remplacée par une classe qui contient une fonction (**`APPELER`**) qui est utilisée pour simuler l'application de la fermeture. Pour simuler l'appel d'une fonction de génération de code, il suffit de faire **`NEW`** `classe( paramètres )`. Par exemple, si on suppose que les seules données disponibles sont de type entier, on obtiendrait le bout de programme suivant pour simuler la fonction de génération de code **`gen-cst`**:

```
CLASS FONCTION;
  VIRTUAL : INTEGER PROCEDURE APPELER;
BEGIN END;

FONCTION CLASS GEN_CST(A); INTEGER A;
BEGIN
  INTEGER PROCEDURE APPELER(E); REF(ENV) E;
    APPELER :- A;
END;
```

Voici un exemple de génération et d'exécution de code basé sur **GEN_CST**:

```
CODE :- NEW GEN_CST( 123 );
OUTINT( CODE.APPELER( NONE ), 3 );
OUTIMAGE;                                  → affiche 123
```

En fait, tout autre langage orienté objet qui permet de créer des objets à l'exécution peut être utilisé comme langage d'implantation pour le compilateur. Nous donnons à l'appendice F un exemple complet de génération de code, en SIMULA 67, pour la fonction Fibonacci bien connue.

3.6 Performance

Cette section analyse la performance de notre méthode de génération de code. Il y a plusieurs aspects qui nous intéressent. En particulier, nous aimerions savoir de combien l'exécution d'un programme compilé à l'aide de notre méthode est plus rapide que s'il avait été interprété et quel facteur d'amélioration est possible en ajoutant certaines des optimisations énoncées à la section précédente à notre compilateur de base. De plus, il serait intéressant de savoir de combien l'exécution est plus lente que si le programme avait été évalué directement à l'aide du langage d'implantation (i.e. le langage utilisé pour exécuter l'interpreteur et le compilateur).

Pour faire ces comparaisons, nous avons tout d'abord modifié le compilateur de base de l'appendice C pour y incorporer certaines optimisations simples. Dans ce nouveau compilateur sont intégrées les optimisations suivantes:

- Les environnements sont représentés par une liste de valeurs. Le compilateur calcule la profondeur de chaque variable dans la liste car celle-ci ne contient pas le nom des variables.
- Aucun environnement n'est passé aux fermetures générées (i.e. ce sont des fermetures à zéro argument). Une variable globale retient l'environnement d'évaluation courant. Celle-ci est sauvée à l'entrée des fonctions et est restaurée à leur retour.
- Afin de conserver la sémantique de récursivité terminale de Scheme tout en effectuant l'optimisation précédente, toutes les fonctions de génération de code sont dédoublées (sauf pour la forme spéciale **if**). La première des deux formes est utilisée pour les expressions

terminales et la seconde est utilisée pour les expressions non-terminales du corps d'une fonction (i.e. les expressions dont la valeur est ou non le résultat des fonctions qui les contiennent). Les fermetures générées pour les expressions terminales restaurent l'environnement avant de retourner ou, dans le cas d'une application, avant de brancher à la fonction appelée.

- Des fonctions de génération de code spécifiques existent pour les cas fréquents suivants: les constantes **`1`**, **`2`** et **`#!null`**; les références et affectations de variables globales et des trois premières variables de l'environnement; les applications de fonctions ayant une variable globale en position fonctionnelle.

Le compilateur optimisant ainsi obtenu figure en appendice D. Il est à peu près trois fois plus long que le compilateur non-optimisant figurant en appendice C.

Afin de faire des comparaisons entre les diverses méthodes d'évaluation, nous avons évalué trois appels de fonction à l'aide de chacune des méthodes. Deux de ces appels, i.e. **`(fib 20)`** et **`(tak 18 12 6)`**, sont assez classiques pour tester la performance des implantations de Lisp et font beaucoup d'appels récursifs et d'arithmétique entière [Gabr82]. Le troisième appel, i.e. **`(trier '(3 1…))`**, consiste à trier soixante-dix nombres à l'aide d'une méthode de tri par sélection. La fonction **`trier`** alloue beaucoup de doublets et utilise la récursivité terminale pour itérer. La définition de ces fonctions et les appels utilisés figurent à l'appendice E.

Puisque le temps et les caractéristiques d'exécution risquent de varier d'une implantation de Scheme à l'autre, nous avons voulu effectuer ces tests à l'aide de plusieurs implantations de Scheme. Les tests ont été effectués sur les deux implantations de Scheme auxquelles nous avions accès: MIT C Scheme [MITS84] et MacScheme [Sema85]. Les caractéristiques de ces implantations sont comme suit:

MIT C Scheme: Écrit en langage C, compile en pseudo-code (*S-code*), environnement d'exécution: SUN sous Berkeley UNIX 4.2 (MC68000), version utilisée: 6.1

MacScheme: Écrit en assembleur MC68000 et Scheme, compile en pseudo-code (*bytecode*), environnement d'exécution: Macintosh Plus 1MB (MC68000), version utilisée: 1.11

Nous avons effectué les tests avec quatre méthodes d'évaluation: avec le langage d'implantation et avec l'interpréteur, le compilateur non-optimisant et optimisant des appendices B, C et D respectivement. Les temps d'exécution des appels dans chacun des divers contextes

d'évaluation sont indiqués à la table 3.1. Les nombres entre parenthèses correspondent au rapport avec le temps requis pour l'exécution dans le langage d'implantation.

Implantation et appel	Méthode d'évaluation			
	langage d'impl.	comp. opt.	comp. non-opt.	interpréteur
MIT C Scheme:				
`(fib 20)`	190 (1.0)	540 (2.8)	1700 (8.9)	3900 (20.5)
`(tak 18 12 6)`	430 (1.0)	1800 (4.1)	9700 (22.5)	16000 (37.2)
`(trier '(3 1…))`	32 (1.0)	110 (3.4)	850 (26.6)	1300 (40.6)
MacScheme:				
`(fib 20)`	53 (1.0)	120 (2.3)	190 (3.6)	440 (8.3)
`(tak 18 12 6)`	130 (1.0)	370 (2.8)	640 (4.9)	1400 (10.8)
`(trier '(3 1…))`	9 (1.0)	23 (2.6)	44 (4.9)	98 (10.9)

Table 3.1: Temps CPU (sec) requis pour exécuter les fonctions **`fib`**, **`tak`** et **`trier`**

Une première observation qu'on peut faire à partir de ces résultats est que les programmes compilés à l'aide du compilateur non-optimisant exécutent de 30-50% plus rapidement que lorsqu'ils sont évalués par l'interpréteur et de quatre à vingt-cinq fois plus lentement que lorsqu'ils sont évalués à l'aide du langage d'implantation. La performance de notre méthode est donc excellente par rapport à un interpréteur. Signalons ici que l'interpréteur utilisé dans les tests n'effectue pas de vérification syntaxique ni d'expansion de macros. De telles opérations, qui sont certainement souhaitables pour avoir une implantation viable de Scheme, diminueraient la performance de l'interpréteur par un facteur important. Dans le cas du compilateur, ces opérations seraient effectuées une fois pour toute à la compilation ce qui n'affecterait pas la performance du code généré.

De plus, les résultats indiquent que les optimisations simples ajoutées au compilateur permettent de réduire le temps d'exécution du code généré de 40-70%. Les programmes traités par le compilateur optimisant exécutent ainsi seulement de deux à quatre fois plus lentement que s'ils avaient été évalués à l'aide du langage d'implantation. Lorsqu'on y pense, ce facteur est étonnament faible puisqu'il indique qu'en général, l'exécution du code d'une construction primitive particulière de Scheme ne doit pas effectuer plus que de deux à quatre évaluations de cette même primitive dans le langage d'implantation (e.g. le code d'une référence de variable ne doit pas faire plus que l'équivalent de deux à quatre références de variables).

Il aurait été intéressant de mesurer l'espace occupé par le code généré. Malheureusement ceci n'est pas facile à mesurer car l'espace occupé dépend de la représentation interne des fermetures qui nous est inconnue. Dans le quatrième chapitre cet aspect est discuté pour le compilateur qui y est décrit. D'un autre côté, il nous était possible, dans l'implantation MacScheme, de savoir combien de fois l'opération de *garbage collection* a été effectuée ce qui est une bonne indication du rythme d'utilisation de la mémoire. Nous avons observé que l'exécution

des programmes traités par le compilateur optimisant provoquait en moyenne trois fois moins d'opérations de *garbage collection* que l'interpréteur et de deux à trois fois plus que s'ils avaient été évalués à l'aide du langage d'implantation.

Les résultats semblent donc encourageants vue la vitesse d'exécution du code généré par le compilateur optimisant par rapport à la vitesse d'exécution du langage d'implantation. Une question intéressante serait de savoir si le même rapport de vitesse existe si le langage d'implantation est l'assembleur. C'est ce que nous ferons dans le quatrième chapitre qui décrit un compilateur Scheme intégrant notre méthode d'implantation de fermetures discutée au deuxième chapitre et la méthode de génération de code discutée dans le chapitre présent.

Chapitre IV

Un compilateur Scheme

Dans ce chapitre, nous décrivons un compilateur Scheme qui utilise la méthode d'implantation de fermetures décrite au deuxième chapitre et la méthode de génération de code décrite au troisième chapitre.

La première section décrit la structure et le fonctionnement du compilateur lui-même et la seconde section en analyse la performance en le comparant à d'autres systèmes.

4.1 Structure et fonctionnement du compilateur

Notre compilateur, écrit en Scheme et figurant à l'appendice G, est essentiellement une extension du compilateur optimisant du troisième chapitre. Nous avons ajouté le nécessaire pour l'expansion des macros de façon à pouvoir traiter toutes les formes spéciales du langage Scheme et le nécessaire pour traduire les λ-expressions en termes d'ε-expressions et d'appels à la fonction **`closure`**. De plus, les fonctions de génération de code ont été modifiées pour que les fermetures créées soient sous la forme d'un bout de code, tel que décrit au deuxième chapitre. Ainsi, le code généré par le compilateur est sous la forme d'un programme en assembleur. Le compilateur accepte le programme à compiler (i.e. une séquence d'expressions) d'un fichier et produit un autre fichier contenant de l'assembleur MC68000. Ce dernier devra être assemblé avant de pouvoir être exécuté.

La compilation est constitué de cinq phases, les quatre premières étant écrites en Scheme:

1) Expansion des macros
2) Traduction des λ-expressions
3) Génération de code sous forme de fermetures
4) Écriture du code généré en assembleur MC68000
5) Assemblage du code assembleur produit

Les trois premières phases sont exécutées séquentiellement pour chaque expression du programme à compiler. La première phase accepte une expression Scheme et les phases suivantes acceptent le résultat de la phase précédente. Les résultats de la troisième phase sont accumulées et sont traitées par la quatrième phase une fois que toutes les expressions ont été traitées par les trois premières phases.

4.1.1 Expansion des macros

La phase d'expansion des macros consiste à réécrire les expressions en terme des formes de base **`quote`**, **`set!`**, **`if`** et **`lambda`** et des constructions de référence de variable et d'application de fonction. Les transformations effectuées sont identiques à celles figurant au premier chapitre.

La transformation d'une expression se fait de façon récursive. Dans le cas ou l'expression est une liste et le premier élément est le mot clé d'une des formes spéciales de Scheme, une fonction d'expansion correspondante est appelée avec, comme argument, le reste de la liste représentant l'expression. La fonction d'expansion effectue la transformation appropriée à cette forme spéciale. Ce procédé se répète jusqu'à ce que l'expression ne soit plus une des formes spéciales de Scheme.

Une particularité de notre méthode d'expansion est que les formes spéciales **`quote`**, **`set!`**, **`if`** et **`lambda`** sont traitées comme toute autre forme spéciale et possèdent aussi une fonction d'expansion. Ceci permet un traitement plus homogène et relègue le traitement des particularités des formes spéciales **`if`** et **`lambda`** aux fonctions d'expansion correspondantes. Les formes spéciales de base utilisées dans notre méthode sont en fait **`<quote>`**, **`<set!>`**, **`<if>`** et **`<lambda>`**. Celles-ci ont une structure rigide qui est plus simple à traiter par le reste du compilateur. La forme **`<if>`** a toujours deux branches et la forme **`<lambda>`** a toujours une seule expression dans son corps.

Nous effectuons aussi deux transformations supplémentaires sur les expressions durant cette phase, juste après l'expansion des macros. Celles-ci sont:

```
((lambda () corps)) → corps
(if 'a b c) → b ou c selon le cas
```

Ces transformations sont effectuées pour palier à certains inconvénients de la méthode d'expansion des macros. Il arrive que l'une de ces formes d'expressions apparaisse après une expansion plutôt que la forme simplifiée. C'est le cas, par exemple, lorsqu'une clause **`else`** termine une forme spéciale **`cond`**. D'autres simplifications de la sorte pourraient aussi être effectuées pour améliorer le compilateur. En particulier, les simplifications décrites par Steele [Stee78b] pourraient être effectuées à ce moment.

4.1.2 Traduction des λ-expressions

Cette phase consiste à traduire les λ-expressions contenues dans l'expression en termes d'ε-expressions et d'appels à la fonction **`closure`**. L'algorithme utilisé est le même que celui décrit au deuxième chapitre sauf que des appels à la fonction **`closure`** sont produits plutôt que

des appels à la fonction **`compile`**. Après cette phase, l'expression ne contient plus que les formes spéciales **`quote`**, **`set!`**, **`if`** et **`epsilon`**.

4.1.3 Génération de code

La phase de génération de code est similaire à celle du compilateur optimisant du troisième chapitre. Elle consiste à créer un réseau de fermetures qui effectue l'évaluation de l'expression lorsqu'il est appelé. Une différence importante est qu'on n'a plus à traiter les λ-expressions à ce niveau, les ε-expressions les ayant remplacées. Les problèmes de représentation de l'environnement n'existent plus car maintenant, toutes les variables (non-globales) sont allouées sur la pile d'exécution et il suffit d'une indirection avec déplacement par rapport au pointeur de pile pour les accéder. Les fermetures créées pendant la génération de code sont donc des fermetures à aucun argument car l'environnement d'exécution est implicite (i.e. la pile). Une autre différence majeure est que le code généré est sous la forme de structures qui représentent des fermetures plutôt que sous la forme de vraies fermetures Scheme. Cette distinction est nécessaire car ces structures doivent être manipulées lors de l'écriture du code assembleur correspondant. Le corps de ces fermetures est représenté par une séquence d'instructions en assembleur MC68000 qui a été écrit à la main.

La convention d'application de fonction est semblable à celle du deuxième chapitre. Pour appeler une fonction il faut empiler la continuation sur la pile suivie de la valeur de chacun des arguments de l'application. Afin de permettre les fonctions n-aires, telle que **`closure`**, un registre indique le nombre d'arguments passés à la fonction. Un branchement doit alors être effectué à la fonction.

Une difficulté supplémentaire dans la génération de code provient du fait que la pile est utilisée à deux fins. D'un côté elle est utilisée pour les arguments et la continuation de la fonction appelée et de l'autre pour les adresses de retour et les variables fermées des fermetures représentant le code de la fonction. Ainsi, lors de l'exécution du corps d'une fonction, la distance d'un paramètre particulier par rapport au dessus de la pile varie. Cette distance dépend du nombre et de la sorte des constructions dans lesquelles l'accès au paramètre se trouve. Pour cette raison, une variable indique, tout au long de la génération de code pour le corps d'une fonction, le nombre de valeurs supplémentaires qui auront été empilés depuis la continuation de la fonction au moment de l'exécution. La valeur de cette variable est ajustée en fonction de la sorte de construction qui englobe l'expression dont il faut générer le code. Cette variable a une double utilité. Premièrement, elle est utilisée pour déterminer la position d'une variable particulière par rapport au dessus de la pile lorsque le code d'une référence ou affectation de variable est généré. Deuxièmement, elle est utilisée pour savoir combien de valeurs il faut désallouer de la pile avant de

poursuivre l'exécution à la continuation ou d'effectuer une application terminale. Il aurait été possible de garder, à l'exécution, un pointeur indiquant le début des paramètres de la fonction courante sur la pile (i.e. un lien dynamique). Ceci aurait simplifié le travail du compilateur mais aurait demandé un travail supplémentaire à l'exécution pour mettre ce pointeur à jour.

La génération de code pour la forme **`(epsilon`** `paramètres corps`**`)`** est plus simple que pour une λ-expression. Elle est similaire à celle pour **`(quote`** `valeur`**`)`** sauf que `valeur` est en fait le code généré pour l'expression `corps` en ne considérant comme environnement que les variables présentes dans `paramètres`. Ainsi, à l'exécution, l'évaluation d'une forme spéciale **`epsilon`** retournera le code (i.e. la fermeture) qui correspond au corps de l'ε-expression en question. Lorsque cette dernière est appelée, la continuation et les arguments se trouvent sur la pile et conséquemment, elle devra les désallouer avant de retourner son résultat. Pour cette raison, toutes les fermetures créés pendant la génération de code sont paramétrisées par une valeur qui indique combien d'espace il faut désallouer de la pile pour retrouver la continuation. Dans le cas de fermetures correspondant à une expression terminale (e.g. la référence à **`x`** dans **`(epsilon (x) x)`**) cette valeur est simplement le nombre de paramètres de l'ε-expression. Dans les autres cas (e.g. la référence à **`x`** dans **`(epsilon (x) (+ x 1))`**), la valeur est zero car ces fermetures sont appelées avec aucun argument pour évaluer une expression non-terminale du corps de l'ε-expression.

Lors de la génération de code, les optimisations suivantes sont effectuées:

- Création de fermetures spécifiques pour les constantes **`1`**, **`2`** et **`#!null`**; les références et affectations de variables globales et des cinq emplacements au dessus de la pile; les applications de fonction ayant une variable globale en position fonctionnelle.
- Toutes les fonctions de génération de code sont dédoublées (sauf pour la forme spéciale **`if`**) pour traiter le cas particulier où il faut désallouer zero valeurs de la pile pour retrouver la continuation.
- Les fermetures créées sont conservées dans une liste et sont réutilisées si une fermeture avec les mêmes paramètres doit être créée.
- Les variables globales sont allouées dans le symbole qui les dénotent.

4.1.4 Écriture du code généré en assembleur

Cette phase est utilisée pour produire le programme assembleur correspondant au code généré. Elle procède au balayage récursif de la structure représentant le code et produit, pour chaque objet qui s'y trouve, les instructions MC68000 correspondantes préfixées d'une étiquette. L'étiquette est utilisée pour dénoter l'objet dans les objets composés qui le contient. Un problème

survient du fait qu'un même objet peut faire partie de plusieurs objets composés et que la structure peut être circulaire. Pour ne pas produire le même objet plusieurs fois, nous gardons une liste de tous les objets déjà traités avec l'étiquette associée et ne produisons les instructions MC68000 correspondantes que si l'objet n'a pas déjà été traité. Pour améliorer la lisibilité, des macros-instructions sont utilisées dans le programme assembleur produit. Un exemple de compilation, incluant le programme assembleur produit et des commentaires, est donné en appendice I.

4.1.5 Lacunes du compilateur

Le compilateur décrit n'est pas un compilateur complet et viable pour le langage Scheme. Plusieurs aspects ont étés omis afin de ne faire resortir que les points important de la synthèse de nos deux méthodes et d'en faciliter la compréhension:

- Les paramètres reste ne sont pas traités. Leur traitement aurait demandé de générer du code plus complexe à l'entrée des fonctions pour retirer les derniers arguments de la pile et d'en construire la liste.
- Le nombre d'arguments des applications de fonctions est limité à trois. Ce nombre peut facilement être augmenté en ajoutant les fonctions de génération de code nécessaire pour les autres classes d'applications à traiter.
- Les types de données implantés sont: booléen, entier, chaîne de caractères, symbole, liste et fonction et seulement un petit sous-ensemble des fonctions primitives existent. La représentation des données est très simpliste et ne possède pas le nécessaire pour permettre l'opération de *garbage collection* ni la vérification du type d'une donnée à l'exécution.
- Aucune situation d'erreur n'est vérifiée à l'exécution.
- Aucune situation d'erreur n'est vérifiée à la compilation à part les limites particulières du compilateur énoncées précédemment.

De plus, un système Scheme viable devrait intégrer totalement le compilateur au reste du système. Présentement il faut quitter le système Scheme pour procéder à l'assemblage du programme assembleur produit. Cependant, si le compilateur était compilé à l'aide de lui-même (ce qui demanderait de résoudre les lacunes précédentes) et que la génération de code était modifiée de façon à utiliser la fonction **`closure`** primitive plutôt que de produire du texte assembleur, il ne serait plus nécessaire de quitter le système car à ce moment, toutes les phases de la compilation s'effectuerait en Scheme.

4.2 Analyse de la performance

La performance de notre compilateur a été mesurée en compilant et exécutant une batterie de programmes qui exercent fortement un aspect particulier du langage. Ces programmes, qui figurent à l'appendice H, ont les caractéristiques suivantes:

`fib` Calcul récursif de la fonction de Fibonacci. Exerce la récursivité.

`tak` Test classique pour Lisp. Exerce la récursivité.

`trier` Tri par sélection d'une liste. Exerce la manipulation de doublets et la récursivité terminale.

`*trier` Comme **`trier`** sauf que les doublets sont implantées à l'aide de fermetures.

`reines` Calcule le nombre de façons de placer 8 reines sur un échiquier sans qu'elles se menacent. Exerce la récursivité et la manipulation de doublets.

`tally` Calcule la somme des nombres de 1 à 999. Exerce les boucles **`do`** et l'application de fermeture avec variable mutable.

Nous avons aussi effectué les mêmes tests sur d'autres implantations de Scheme et de Lisp de façon à pouvoir les comparer. Les caractéristiques de ces implantations sont:

MacScheme: Compilateur, génère du pseudo-code, écrit en assembleur MC68000 et Scheme, version 1.11 [Sema85]

ExperLisp: Compilateur, génère du code machine MC68000, version 1.0 [Expe84]

XLISP: Interpréteur, écrit en langage C, version 1.5 [Betz85]

Tous les tests ont été exécutés sur un Macintosh Plus avec 1MB de mémoire RAM. Trois caractéristiques ont été mesurées: le temps d'exécution du programme, l'espace alloué du monceau lors de l'exécution du programme et l'espace occupé par le code du programme. Les résultats de ces tests figurent à la table 4.1. Les valeurs entre parenthèses correspondent au rapport des mesures avec celles de notre compilateur. Une série d'étoiles indiquent les situations où nous n'avons pas pu faire exécuter le programme dû à des problèmes de portée lexicale ou à des débordements de pile à l'exécution. Dans le cas de ExperLisp, les mesures reliées à l'espace ne nous étaient pas disponibles.

Du point de vue du temps d'exécution, le code généré par notre compilateur est de 2 à 4 fois plus rapide que ExperLisp (qui génère du code machine), de 5 à 10 fois plus rapide que MacScheme (qui génère du pseudo-code) et de 50 à 130 fois plus rapide que XLISP (qui interprète les expressions). Une irrégularité semble se produire pour le programme **`tally`**. Ceci est attribuable à la forme complexe qui est produite pour la forme spéciale **`do`**, forme qui est probablement traitée de façon particulière dans les autres implantations. Des phases de simplification et d'optimisation plus completes, telles que celles de [Stee78b], seraient souhaitables

pour résoudre cette lacune. Malgré tout, le code généré est considérablement plus rapide que les autres implantations.

Programme	Système utilisé			
	Notre compilateur	MacScheme	ExperLisp	XLISP
Temps d'exécution du programme (sec):				
fib	5.0 (1.0)	53.0 (11.0)	13.0 (2.6)	640 (130)
tak	17.0 (1.0)	130.0 (7.6)	44.0 (2.6)	2200 (130)
trier	1.1 (1.0)	8.6 (7.8)	4.0 (3.6)	**********
*trier	2.3 (1.0)	16.0 (7.0)	**********	**********
reines	17.0 (1.0)	170.0 (10.0)	34.0 (2.0)	2000 (120)
tally	0.8 (1.0)	4.6 (5.8)	**********	41 (51)
Espace alloué du monceau à l'exécution (Koctets):				
fib	0 (1.0)	2200 ( )	----------	2300 ( )
tak	0 (1.0)	7500 ( )	----------	9300 ( )
trier	20 (1.0)	310 (16.0)	----------	**********
*trier	46 (1.0)	850 (19.0)	----------	**********
reines	80 (1.0)	7800 (98.0)	----------	7000 (88.0)
tally	30 (1.0)	190 (6.3)	----------	140 (4.7)
Espace occupé par le programme (octets):				
fib	470 (1.0)	660 (1.4)	----------	390 (0.8)
tak	660 (1.0)	710 (1.1)	----------	610 (0.9)
trier	1500 (1.0)	1600 (1.1)	----------	2400 (1.6)
*trier	2900 (1.0)	3600 (1.2)	----------	4200 (1.4)
reines	2400 (1.0)	2000 (0.8)	----------	2500 (1.0)
tally	1300 (1.0)	980 (0.8)	----------	690 (0.5)

Table 4.1: Résultats des tests de performance

Les résultats sont aussi favorables du point de vue de l'espace. Dans le cas de l'espace alloué du monceau à l'exécution, notre compilateur est beaucoup plus efficace que les autres implantations. Cette différence est probablement dû au fait que les autres implantations allouent toujours les environnements du monceau ce qui n'est pas le cas de notre compilateur. Pour un même espace mémoire, les programmes compilés à l'aide de notre compilateur devraient ainsi effectuer bien moins souvent l'opération de *garbage collection* que les autres implantations.

Dans le cas de l'espace occupé par les programmes, notre compilateur donne des résultats assez similaires aux autres implantations. Il faut cependant remarquer que les mesures pour notre compilateur incluent l'espace occupé par le code des fonctions primitives et que les programmes sont petits n'offrant ainsi que peu de situations pour partager le code généré. Les résultats devraient être légèrements meilleurs pour de plus gros programmes.

Il ne faut pas oublier que le code généré par notre compilateur n'effectue aucunes vérifications à l'exécution et que la représentation des données est trop simpliste pour être viable.

Pour nous assurer que cet aspect n'influence pas trop grandement l'efficacité, nous avons modifié, à la main, le code des programmes **`fib`** et **`tak`** pour inclure le nécessaire pour détecter les erreurs d'exécution et attacher un type aux données. Le temps d'exécution pour ces programmes a seulement augmenté de 20-30%. Ceci s'explique du fait que seul les fonctions primitives et le corps des fermetures générées ont été modifiés. Puisque l'exécution du code consiste en majeure partie en l'exécution des fonctions d'interface et que celles-ci n'ont pas été modifiées, le temps d'exécution global n'a pas beaucoup changé. De plus, puisque les fonctions primitives et le corps des fermetures sont partagés, la taille du code est restée sensiblement la même.

Nous avons aussi réécrit les programmes **`fib`** et **`tak`** à la main, en assembleur MC68000 et en utilisant les mêmes conventions d'appel de fonction que notre compilateur. Les temps obtenus indiquent que le code généré par notre compilateur exécute trois fois plus lentement que si ces fonctions sont écrites en assembleur. Ce résultat est donc cohérent avec celui obtenu au troisième chapitre en comparant le compilateur optimisant avec le langage d'implantation, car en quelque sorte, le langage d'implantation de notre compilateur est le langage machine.

Conclusion

Ce mémoire a proposé de nouvelles approches à l'implantation efficace de deux aspects importants d'un système Scheme. Dans un premier temps, nous avons montré le lien qui existe entre les fermetures du langage Scheme et les abstractions du λ-calcul. Les fermetures sont très versatiles comme outil de programmation et nous avons décrit comment elles peuvent être utilisées pour exprimer un grand nombre de constructions spéciales de Scheme. Ainsi, l'efficacité des fermetures a un impact important sur l'efficacité globale d'un système Scheme conçu de cette façon.

Nous avons décrit une méthode d'implantation de fermetures qui s'apparente à la β-conversion du λ-calcul. Elle intègre à la fois les aspects de rétention d'environnement et de méthode de calcul propres aux fermetures. Elle permet d'exprimer la création de fermetures à l'aide d'autres outils du langage source, notamment un compilateur et des λ-expressions dégénérées. Nous avons raffiné cette méthode et avons montré qu'une fermeture correspond simplement à un bout de code de structure régulière. Celui-ci contient les informations nécessaires de l'environnement de création. La création d'une fermetures revient à la génération de ce bout de code et son application consiste à y brancher en passant les arguments de l'application. Contrairement à la méthode classique d'implantation de fermetures, aucune représentation particulière des environnements n'est nécessaire à l'exception de la pile d'exécution. Les accès aux variables sont ainsi plus simples et ne consistent qu'en une indirection du pointeur de pile. L'application d'une fermeture est aussi plus simple que pour la méthode classique puisqu'elle ne nécessite pas la manipulation explicite d'un doublet code-environnement et d'un descripteur d'environnement. Ceci a l'avantage additionnel d'accélérer l'application de fonctions primitives qui est généralement une opération fréquente. Les tests effectués montrent que dans plusieurs situations notre méthode est plus efficace que la méthode classique. Nous proposons aussi une méthode hybride qui combine les avantages des deux méthodes et qui peut être utilisée lorsque notre méthode est faible. Il serait d'ailleurs interessant d'étudier plus en profondeur comment et sur quelles informations le compilateur pourrait faire ce choix. En plus de pouvoir être utilisée dans un intrerpréteur ou dans un compilateur pour implanter les fermetures, notre méthode peut être utilisée pour simuler les fermetures dans les dialectes de Lisp qui ne les ont pas.

Le deuxième aspect que nous avons étudié est la génération de code pour un compilateur Scheme. Nous avons montré que les fermetures peuvent jouer le rôle de code permettant ainsi d'effectuer la génération de code entièrement à partir de Scheme. Un compilateur utilisant cette

méthode est d'une structure et d'un ordre de complexité similaire à un interpréteur sans pour autant en perdre les qualités essentielles (e.g. informations de mise au point). Pourtant, l'exécution de programmes compilés de cette façon est plus rapide que s'ils étaient interprétés. De plus, il est possible d'intégrer des optimisations au compilateur pour améliorer l'efficacité du code produit et pour partager du code. Ainsi, la méthode décrite permet de remplacer avantageusement les interpréteurs, en particulier dans le cas des *embedded languages*. La même idée s'étend également à d'autres langages et permet, par exemple, d'effectuer la génération de code dans les langages orienté-objet comme SIMULA 67.

Nous croyons qu'un système Scheme peut utiliser à profit les deux approches proposées dans ce mémoire et particulièrement lorsqu'elles sont utilisées conjointement. En effet, nous avons réalisé un compilateur qui combine les deux approches et les temps d'exécution des programmes de tests compilés par celui-ci se comparent avantageusement avec ceux de certains systèmes commerciaux de Scheme et de Lisp couramment disponibles sur le marché.

Remerciements

Par la présente, je tiens à remercier mon directeur de recherche, Monsieur Guy Lapalme, professeur au département d'informatique et de recherche opérationnelle de l'Université de Montréal, pour l'intérêt, l'assistance, les conseils et les critiques qu'il a porté sur mes travaux. Je tiens aussi à remercier mes collègues et amis du laboratoire de robotique, André Foisy, Denis Fortin et Gilles Hurteau, pour le soutien et l'encouragement qu'ils m'ont témoignés tout au long de mes études de maîtrise et particulièrement dans les moments difficiles.

Appendice A

Programme de traduction de λ-expressions

Nous donnons ici le source du programme de traduction de λ-expressions écrit en Scheme. La fonction principale est **`lambda->epsilon+compile`**. Elle prend en paramètre une expression Scheme qui contient des λ-expressions et retourne une expression équivalente qui contient des ε-expressions et des appels à la fonction **`compile`** à la place des λ-expressions.

```
;_____________________________________________________________________________________
;
;                              Traduction de lambda-expressions
;
;_____________________________________________________________________________________

(define (lambda->epsilon+compile expr)
  (pass2 (pass1 expr '())))


; --- Premiere passe ---
;
; Construire une structure qui represente l'expression originale et la decorer avec les
; informations relatives au type d'acces aux variables

(define (pass1 expr env)
  (cond ((symbol? expr)           (pass1-ref expr env))
        ((not (pair? expr))       (pass1-cst expr env))
        ((eq? (car expr) 'quote)  (pass1-cst (cadr expr) env))
        ((eq? (car expr) 'set!)   (pass1-set (cadr expr) (caddr expr) env))
        ((eq? (car expr) 'if)     (pass1-tst (cadr expr) (caddr expr) (cadddr expr) env))
        ((eq? (car expr) 'lambda) (pass1-def (cadr expr) (cddr expr) env))
        (else                     (pass1-app expr env))))

(define (pass1-cst val env)
  (list 'cst val))

(define (pass1-ref var env)
  (list 'ref (acceder var env)))

(define (pass1-set var val env)
  (let ((v (acceder var env)))
    (affectee v)
    (list 'set v (pass1 val env))))

(define (pass1-tst pre con alt env)
  (list 'tst (pass1 pre env) (pass1 con env) (pass1 alt env)))

(define (pass1-def patron corps env)
  (let ((env* (list 'def env (parms patron) '() patron '())))
    (set-car! (cdddddr env*) (map (lambda (x) (pass1 x env*)) corps))
    env*))

(define (parms l)
  (cond ((null? l) #!null)
        ((pair? l) (cons (parametre (car l)) (parms (cdr l))))
        (else      (list (parametre l)))))

(define (pass1-app vals env)
  (cons 'app (map (lambda (x) (pass1 x env)) vals)))


; --- Deuxieme passe ---
;
; Traduire la structure obtenue a la premiere passe en l'expression Scheme correspondante
; en tenant compte du type des variables (i.e. globale/locale/fermee & mutable ou non)
; et de la liste des variables fermees des lambda-expressions

(define (pass2 struct)
  (cond ((eq? (car struct) 'cst) (pass2-cst (cadr struct)))
        ((eq? (car struct) 'ref) (pass2-ref (cadr struct)))
        ((eq? (car struct) 'set) (pass2-set (cadr struct) (caddr struct)))
        ((eq? (car struct) 'tst) (pass2-tst (cadr struct) (caddr struct) (cadddr struct)))
        ((eq? (car struct) 'def) (pass2-def (caddr struct) (cadddr struct) (caddddr struct)
                                                                           (cadddddr struct)))
        ((eq? (car struct) 'app) (pass2-app (cdr struct)))
        (else (error "structure invalide"))))

(define (pass2-cst val)
  (quote1 val))
```

```
(define (pass2-ref var)
  (if (mutable? var)
    (list 'fetch (nom var))
    (nom var)))

(define (pass2-set var val)
  (if (mutable? var)
    (list 'store (nom var) (pass2 val))
    (list 'set! (nom var) (pass2 val))))

(define (pass2-tst pre con alt)
  (list 'if (pass2 pre) (pass2 con) (pass2 alt)))

(define (pass2-def parms fermees patron corps)
  (let ((parametres (map nom parms))
        (boites (ajouter-boites parms)))
    (let ((corps* (if (equal? parametres boites) (map pass2 corps)
                      (list (cons (cons 'epsilon (cons parametres (map pass2 corps))) boites)))))
      (if (null? fermees)
        (cons 'epsilon (cons patron corps*))
        (list 'compile
          (list 'list
            (quote1 'epsilon)
            (quote1 patron)
            (cons 'list
              (cons (quote2 (cons 'epsilon (cons (append parametres fermees) corps*)))
                (append (map quote1 parametres) (map quote2 fermees)))))))))

(define (ajouter-boites parms)
  (if (null? parms) #!null
    (cons (if (mutable? (car parms))
            (list 'box (nom (car parms)))
            (nom (car parms)))
          (ajouter-boites (cdr parms)))))

(define (pass2-app vals)
  (map pass2 vals))

(define (quote1 x) (list 'quote x))   ; retourne la liste: (quote @)
(define (quote2 x) (list 'list (quote1 'quote) x))   ; retourne la liste: (list 'quote @)
(define (cddddr x) (cdr (cddddr x)))
(define (caddddr x) (car (cddddr x)))
(define (cadddddr x) (cadr (cddddr x)))


; --- Fonctions de manipulation de variables et d'environnements ---

(define (acceder nom env)
  (cond ((null? env)
         (var-globale nom))
        ((assq nom (caddr env)))
        (else
         (let ((var (acceder nom (cadr env))))
           (if (not (globale? var))
             (begin
               (fermee var)
               (if (not (memq nom (cadddr env)))
                 (set-car! (cdddr env) (cons nom (cadddr env))))))
           var))))

(define (parametre nom)   (list nom 'locale  #!false))
(define (var-globale nom) (list nom 'globale #!false))
(define (nom x)           (car x))
(define (globale? x)      (eq? (cadr x) 'globale))
(define (fermee? x)       (eq? (cadr x) 'fermee))
(define (fermee x)        (set-car! (cdr x) 'fermee))
(define (affectee? x)     (caddr x))
(define (affectee x)      (set-car! (cddr x) #!true))
(define (mutable? x)      (and (fermee? x) (affectee? x)))


; --- Fonctions utilisees lors de l'evaluation de l'expression transformee ---

; Le macro 'epsilon' permet de simuler les epsilon-expressions a l'aide des lambda-expressions
; de Scheme.  Cependant, la puissance complete des lambda-expressions n'est pas utilisee car
; toute variable accedee dans leur corps est soit locale ou globale.  Ce macro est defini
; suivant la syntaxe de MacScheme.

(macro epsilon (lambda (x) (cons 'lambda (cdr x))))   ; (epsilon ...) --> (lambda ...)

(define (compile x) (eval x))   ; compiler une epsilon-expression c'est comme l'evaluer
(define (box v)     (list v))   ; une boite, c'est un doublet
(define (fetch b)   (car b))
(define (store b v) (set-car! b v))
```

Nous donnons ici quelques exemples de traductions effectuées à l'aide du programme précédent. Afin d'améliorer la lisibilité, les requêtes de l'usager sont en italique. Celles-ci sont suivies du résultat s'il n'est pas indéfini.

1) Exemple de traduction de λ-expressions n'ayant pas de variables fermées:

```
==> (lambda->epsilon+compile
      '(lambda (x) (cons x (lambda (y) y))))

(epsilon (x) (cons x (epsilon (y) y)))
```

2) Exemple de traduction d'une λ-expression ayant une variable fermée non-mutable:

```
==> (lambda->epsilon+compile
      '(define adder (lambda (x) (lambda (y) (+ x y)))))

(define adder
  (epsilon (x)
    (compile
      (list 'epsilon '(y)
        (list (list 'quote (epsilon (y x) (+ x y)))
              'y
              (list 'quote x))))))
```

3) Exemple de traduction d'une λ-expression ayant une variable fermée mutable:

```
==> (lambda->epsilon+compile
      '(define tally (lambda (x) (lambda (y) (set! x (+ x y)) x))))

(define tally
  (epsilon (x)
    ((epsilon (x)
       (compile
         (list 'epsilon '(y)
           (list (list 'quote (epsilon (y x) (store x (+ (fetch x) y)) (fetch x)))
                 'y
                 (list 'quote x)))))
     (box x))))
```

4) Exemple de traduction d'une λ-expression ayant une variable fermée mutable et une variable fermée non-mutable:

```
==> (lambda->epsilon+compile
      '(define accumulateur (lambda (f x) (lambda (y) (set! x (f y x)) x))))

(define accumulateur
  (epsilon (f x)
    ((epsilon (f x)
       (compile
         (list 'epsilon '(y)
           (list (list 'quote (epsilon (y f x) (store x (f y (fetch x))) (fetch x)))
                 'y
                 (list 'quote f)
                 (list 'quote x)))))
     f
     (box x))))
```

5) Exemple d'implantation de doublets à l'aide de fermetures:

```
==> (define temp
      (lambda->epsilon+compile
        '(define *cons
           (lambda (the-car the-cdr)
             (lambda (msg)
               (cond ((= msg 1) (lambda () the-car))
                     ((= msg 2) (lambda () the-cdr))
                     ((= msg 3) (lambda (v) (set! the-car v)))
                     ((= msg 4) (lambda (v) (set! the-cdr v)))
                     (else      (error "undefined operation")))))))

==> temp

(define *cons
  (epsilon (the-car the-cdr)
    ((epsilon (the-car the-cdr)
       (compile
         (list 'epsilon '(msg)
           (list
             (list 'quote
               (epsilon (msg the-car the-cdr)
                 (cond ((= msg '1)
                        (compile
                          (list 'epsilon '()
                            (list (list 'quote (epsilon (the-car) (fetch the-car)))
                                  (list 'quote the-car)))))
                       ((= msg '2)
                        (compile
                          (list 'epsilon '()
                            (list (list 'quote (epsilon (the-cdr) (fetch the-cdr)))
                            (list 'quote the-cdr)))))
                       ((= msg '3)
                        (compile
                          (list 'epsilon '(v)
                            (list (list 'quote (epsilon (v the-car) (store the-car v)))
                                  'v
                                  (list 'quote the-car)))))
                       ((= msg '4)
                        (compile
                          (list 'epsilon '(v)
                            (list (list 'quote (epsilon (v the-cdr) (store the-cdr v)))
                                  'v
                                  (list 'quote the-cdr)))))
                       (else
                        (error '"undefined operation")))))
             'msg
             (list 'quote the-car)
             (list 'quote the-cdr)))))
     (box the-car)
     (box the-cdr))))

==> (eval temp)

==> (define (*car pair)          ((pair 1)))

==> (define (*cdr pair)          ((pair 2)))

==> (define (*set-car! pair val) ((pair 3) val))

==> (define (*set-cdr! pair val) ((pair 4) val))

==> (define a (*cons 'x 'y))

==> (*car a)

x

==> (*cdr a)

y

==> (*set-car! a 'z)

==> (*car a)

z

==> (*cdr a)

y
```

Appendice B

Interpréteur Scheme écrit en Scheme

```
;_____________________________________________________________________________________________
;
;                              Interpreteur Scheme ecrit en Scheme
;
; Note: Seulement les formes speciales 'quote', 'set!', 'if' et 'lambda' ainsi que la reference
;       de constante et de variable et l'application de fonction sont traitees car les autres
;       constructions de Scheme peuvent s'exprimer a partir de celles-ci.
;_____________________________________________________________________________________________


(define (interpret expr)
  (int expr *env-glo*))

(define (int expr env)
  (cond ((symbol? expr)          (int-ref expr env))
        ((not (pair? expr))      (int-cst expr env))
        ((eq? (car expr) 'quote) (int-cst (cadr expr) env))
        ((eq? (car expr) 'set!)  (int-set (cadr expr) (caddr expr) env))
        ((eq? (car expr) 'if)    (int-tst (cadr expr) (caddr expr) (cadddr expr) env))
        ((eq? (car expr) 'lambda)
         (let ((p (cadr expr)))
           (cond ((null? p)          (int-fn0      (caddr expr) env))
                 ((symbol? p)        (int-fn1/rest (caddr expr) p env))
                 ((null? (cdr p))    (int-fn1      (caddr expr) (car p) env))
                 ((symbol? (cdr p))  (int-fn2/rest (caddr expr) (car p) (cdr p) env))
                 ((null? (cddr p))   (int-fn2      (caddr expr) (car p) (cadr p) env))
                 ((symbol? (cddr p)) (int-fn3/rest (caddr expr) (car p) (cadr p) (cddr p) env))
                 ((null? (cdddr p))  (int-fn3      (caddr expr) (car p) (cadr p) (caddr p) env))
                 (else               (error "trop de parametres dans une lambda-expression")))))
        ((null? (cdr expr))    (int-ap0 (car expr) env))
        ((null? (cddr expr))   (int-ap1 (car expr) (cadr expr) env))
        ((null? (cdddr expr))  (int-ap2 (car expr) (cadr expr) (caddr expr) env))
        ((null? (cddddr expr)) (int-ap3 (car expr) (cadr expr) (caddr expr) (cadddr expr) env))
        (else                  (error "trop d'arguments dans une application de fonction"))))

; --- interpretation d'une constante ---

(define (int-cst a env)
  a)

; --- interpretation d'une reference de variable ---

(define (int-ref a env)
  (cdr (assq a env)))

; --- interpretation d'une affectation ---

(define (int-set a b env)
  (set-cdr! (assq a env) (int b env)))

; --- interpretation d'une forme speciale 'if' ---

(define (int-tst a b c env)
  (if (int a env) (int b env) (int c env)))

; --- interpretation d'une application de fonction ---

(define (int-ap0 a env)
  ((int a env)))

(define (int-ap1 a b env)
  ((int a env) (int b env)))

(define (int-ap2 a b c env)
  ((int a env) (int b env) (int c env)))

(define (int-ap3 a b c d env)
  ((int a env) (int b env) (int c env) (int d env)))

; --- interpretation d'une forme speciale 'lambda' ---

(define (int-fn0 a env)
  (lambda () (int a env)))

(define (int-fn1 a b env)
  (lambda (x) (int a (cons (cons b x) env))))

(define (int-fn2 a b c env)
  (lambda (x y) (int a (cons (cons b x) (cons (cons c y) env)))))

(define (int-fn3 a b c d env)
  (lambda (x y z) (int a (cons (cons b x) (cons (cons c y) (cons (cons d z) env))))))
```

```
(define (int-fn1/rest a b env)
  (lambda x (int a (cons (cons b x) env))))

(define (int-fn2/rest a b c env)
  (lambda (x . y) (int a (cons (cons b x) (cons (cons c y) env)))))

(define (int-fn3/rest a b c d env)
  (lambda (x y . z) (int a (cons (cons b x) (cons (cons c y) (cons (cons d z) env))))))

; --- definition de variables globales ---

(define (define-global var val)
  (if (assq var *env-glo*)
    (set-cdr! (assq var *env-glo*) val)
    (begin
      (set-cdr! *env-glo* (cons (car *env-glo*) (cdr *env-glo*)))
      (set-car! *env-glo* (cons var val)))))

(define *env-glo* (list (cons 'define define-global)))
(define-global 'cons  cons )
(define-global 'car   car  )
(define-global 'cdr   cdr  )
(define-global 'null? null?)
(define-global 'not   not  )
(define-global '<     <    )
(define-global '-1+   -1+  )
(define-global '+     +    )
(define-global '-     -    )

; --- pour evaluer, on appelle l'interprete ---

(define (evaluer expr)
  (interpret expr))


;_____________________________________________________________________________________________
;
; Exemples d'utilisation de l'interprete:
;
; a) pour evaluer:
;
;      (define fib
;        (lambda (x)
;          (if (< x 2) x (+ (fib (- x 1)) (fib (- x 2))))))
;
;    il faut entrer:
;
;      (evaluer '(define 'fib
;                  (lambda (x)
;                    (if (< x 2) x (+ (fib (- x 1)) (fib (- x 2)))))))
;
; b) pour evaluer:
;
;      (fib 20)
;
;    il faut entrer:
;
;      (evaluer '(fib 20))
```

Appendice C

Compilateur Scheme non-optimisant écrit en Scheme

```
;_____________________________________________________________________________________________
;
;                     Compilateur Scheme non-optimisant ecrit en Scheme
;
; Note: Seulement les formes speciales 'quote', 'set!', 'if' et 'lambda' ainsi que la reference
;       de constante et de variable et l'application de fonction sont traitees car les autres
;       constructions de Scheme peuvent s'exprimer a partir de celles-ci.
;_____________________________________________________________________________________________


(define (compile expr)
  ((gen expr) *env-glo*))

(define (gen expr)
  (cond ((symbol? expr)          (gen-ref expr))
        ((not (pair? expr))      (gen-cst expr))
        ((eq? (car expr) 'quote) (gen-cst (cadr expr)))
        ((eq? (car expr) 'set!)  (gen-set (cadr expr) (gen (caddr expr))))
        ((eq? (car expr) 'if)    (gen-tst (gen (cadr expr)) (gen (caddr expr)) (gen (cadddr expr))))
        ((eq? (car expr) 'lambda)
         (let ((p (cadr expr)))
           (cond ((null? p)          (gen-fn0      (gen (caddr expr))))
                 ((symbol? p)        (gen-fn1/rest (gen (caddr expr)) p))
                 ((null? (cdr p))    (gen-fn1      (gen (caddr expr)) (car p)))
                 ((symbol? (cdr p))  (gen-fn2/rest (gen (caddr expr)) (car p) (cdr p)))
                 ((null? (cddr p))   (gen-fn2      (gen (caddr expr)) (car p) (cadr p)))
                 ((symbol? (cddr p)) (gen-fn3/rest (gen (caddr expr)) (car p) (cadr p) (cddr p)))
                 ((null? (cdddr p))  (gen-fn3      (gen (caddr expr)) (car p) (cadr p) (caddr p)))
                 (else               (error "trop de parametres dans une lambda-expression")))))
        ((null? (cdr expr))    (gen-ap0 (gen (car expr))))
        ((null? (cddr expr))   (gen-ap1 (gen (car expr)) (gen (cadr expr))))
        ((null? (cdddr expr))  (gen-ap2 (gen (car expr)) (gen (cadr expr)) (gen (caddr expr))))
        ((null? (cddddr expr)) (gen-ap3 (gen (car expr)) (gen (cadr expr)) (gen (caddr expr))
                                                                           (gen (cadddr expr))))
        (else                  (error "trop d'arguments dans une application de fonction"))))

; --- generation de code pour une constante ---

(define (gen-cst a)
  (lambda (env) a))

; --- generation de code pour une reference de variable ---

(define (gen-ref a)
  (lambda (env) (cdr (assq a env))))

; --- generation de code pour une affectation ---

(define (gen-set a b)
  (lambda (env) (set-cdr! (assq a env) (b env))))

; --- generation de code pour une forme speciale 'if' ---

(define (gen-tst a b c)
  (lambda (env) (if (a env) (b env) (c env))))

; --- generation de code pour une application de fonction ---

(define (gen-ap0 a)
  (lambda (env) ((a env))))

(define (gen-ap1 a b)
  (lambda (env) ((a env) (b env))))

(define (gen-ap2 a b c)
  (lambda (env) ((a env) (b env) (c env))))

(define (gen-ap3 a b c d)
  (lambda (env) ((a env) (b env) (c env) (d env))))

; --- generation de code pour une forme speciale 'lambda' ---

(define (gen-fn0 a)
  (lambda (env)
    (lambda () (a env))))

(define (gen-fn1 a b)
  (lambda (env)
    (lambda (x) (a (cons (cons b x) env)))))

(define (gen-fn2 a b c)
  (lambda (env)
    (lambda (x y) (a (cons (cons b x) (cons (cons c y) env))))))
```

```
(define (gen-fn3 a b c d)
  (lambda (env)
    (lambda (x y z) (a (cons (cons b x) (cons (cons c y) (cons (cons d z) env)))))))

(define (gen-fn1/rest a b)
  (lambda (env)
    (lambda x (a (cons (cons b x) env)))))

(define (gen-fn2/rest a b c)
  (lambda (env)
    (lambda (x . y) (a (cons (cons b x) (cons (cons c y) env))))))

(define (gen-fn3/rest a b c d)
  (lambda (env)
    (lambda (x y . z) (a (cons (cons b x) (cons (cons c y) (cons (cons d z) env)))))))

; --- definition de variables globales ---

(define (define-global var val)
  (if (assq var *env-glo*)
    (set-cdr! (assq var *env-glo*) val)
    (begin
      (set-cdr! *env-glo* (cons (car *env-glo*) (cdr *env-glo*)))
      (set-car! *env-glo* (cons var val)))))

(define *env-glo* (list (cons 'define define-global)))
(define-global 'cons  cons )
(define-global 'car   car  )
(define-global 'cdr   cdr  )
(define-global 'null? null?)
(define-global 'not   not  )
(define-global '<     <    )
(define-global '-1+   -1+  )
(define-global '+     +    )
(define-global '-     -    )

; --- pour evaluer, on compile une lambda-expression contenant l'expression a evaluer et ---
; --- on appelle la fonction resultante                                                  ---

(define (evaluer expr)
  ((compile (list 'lambda '() expr))))


;_____________________________________________________________________________________________
;
; Exemples d'utilisation du compilateur:
;
; a) pour evaluer:
;
;      (define fib
;        (lambda (x)
;          (if (< x 2) x (+ (fib (- x 1)) (fib (- x 2))))))
;
;    il faut entrer:
;
;      (evaluer '(define 'fib
;                  (lambda (x)
;                    (if (< x 2) x (+ (fib (- x 1)) (fib (- x 2)))))))
;
; b) pour evaluer:
;
;      (fib 20)
;
;    il faut entrer:
;
;      (evaluer '(fib 20))
```

Appendice D

Compilateur Scheme optimisant écrit en Scheme

```
;_____________________________________________________________________________________________
;
;                        Compilateur Scheme optimisant ecrit en Scheme
;
; Note: Seulement les formes speciales 'quote', 'set!', 'if' et 'lambda' ainsi que la reference
;       de constante et de variable et l'application de fonction sont traitees car les autres
;       constructions de Scheme peuvent s'exprimer a partir de celles-ci.
;_____________________________________________________________________________________________


(define (compile expr)
  ((gen expr '() #!false)))

(define (gen expr env term)
  (cond ((symbol? expr)
         (ref (variable expr env) term))
        ((not (pair? expr))
         (cst expr term))
        ((eq? (car expr) 'quote)
         (cst (cadr expr) term))
        ((eq? (car expr) 'set!)
         (set (variable (cadr expr) env) (gen (caddr expr) env #!false) term))
        ((eq? (car expr) 'if)
         (gen-tst (gen (cadr expr) env #!false)
                  (gen (caddr expr) env term)
                  (gen (cadddr expr) env term)))
        ((eq? (car expr) 'lambda)
         (let ((p (cadr expr)))
           (fn p (gen (caddr expr) (allouer p env) #!true) term)))
        (else
         (let ((args (map (lambda (x) (gen x env #!false)) (cdr expr))))
           (let ((var (and (symbol? (car expr)) (variable (car expr) env))))
             (if (global? var)
               (app (cons var args) #!true term)
               (app (cons (gen (car expr) env #!false) args) #!false term)))))))

(define (allouer parms env)
  (cond ((null? parms)   env)
        ((symbol? parms) (cons parms env))
        (else            (cons (car parms) (allouer (cdr parms) env)))))

(define (variable symb env)
  (let ((x (memq symb env)))
    (if x
      (- (length env) (length x))
      (begin
        (if (not (assq symb *env-glo*)) (define-global symb 'indefini))
        (assq symb *env-glo*)))))

(define (global? var)
  (pair? var))

(define (cst val term)
  (cond ((eqv? val 1)      ((if term gen-1*    gen-1   )    ))
        ((eqv? val 2)      ((if term gen-2*    gen-2   )    ))
        ((eqv? val #!null) ((if term gen-null* gen-null)    ))
        (else              ((if term gen-cst*  gen-cst ) val))))

(define (ref var term)
  (cond ((global? var) ((if term gen-ref-glo*   gen-ref-glo  ) var))
        ((= var 0)     ((if term gen-ref-loc-1* gen-ref-loc-1)    ))
        ((= var 1)     ((if term gen-ref-loc-2* gen-ref-loc-2)    ))
        ((= var 2)     ((if term gen-ref-loc-3* gen-ref-loc-3)    ))
        (else          ((if term gen-ref*       gen-ref      ) var))))

(define (set var val term)
  (cond ((global? var) ((if term gen-set-glo*   gen-set-glo  ) var val))
        ((= var 0)     ((if term gen-set-loc-1* gen-set-loc-1)     val))
        ((= var 1)     ((if term gen-set-loc-2* gen-set-loc-2)     val))
        ((= var 2)     ((if term gen-set-loc-3* gen-set-loc-3)     val))
        (else          ((if term gen-set*       gen-set      ) var val))))

(define (fn parms corps term)
  ((cond ((null? parms)          (if term gen-fn0*      gen-fn0     ))
         ((symbol? parms)        (if term gen-fn1/rest* gen-fn1/rest))
         ((null? (cdr parms))    (if term gen-fn1*      gen-fn1     ))
         ((symbol? (cdr parms))  (if term gen-fn2/rest* gen-fn2/rest))
         ((null? (cddr parms))   (if term gen-fn2*      gen-fn2     ))
         ((symbol? (cddr parms)) (if term gen-fn3/rest* gen-fn3/rest))
         ((null? (cdddr parms))  (if term gen-fn3*      gen-fn3     ))
         (else                   (error "trop de parametres dans une lambda-expression")))
   corps))
```

```
(define (app vals glo term)
  (apply (case (length vals)
           ((1) (if glo (if term gen-ap0-glo* gen-ap0-glo) (if term gen-ap0* gen-ap0)))
           ((2) (if glo (if term gen-ap1-glo* gen-ap1-glo) (if term gen-ap1* gen-ap1)))
           ((3) (if glo (if term gen-ap2-glo* gen-ap2-glo) (if term gen-ap2* gen-ap2)))
           ((4) (if glo (if term gen-ap3-glo* gen-ap3-glo) (if term gen-ap3* gen-ap3)))
           (else (error "trop d'arguments dans une application de fonction")))
         vals))

; --- fonctions de generation de code pour les evaluations non-terminales du corps d'une fonction ---

; --- generation de code pour une constante ---

(define (gen-cst a) ; constante quelconque
  (lambda () a))

(define (gen-1)     ; pour la constante 1
  (lambda () 1))

(define (gen-2)     ; pour la constante 2
  (lambda () 2))

(define (gen-null)  ; pour la constante #!null
  (lambda () #!null))

; --- generation de code pour une reference de variable ---

(define (gen-ref-glo a) ; pour une variable globale
  (lambda () (cdr a)))

(define (gen-ref a)     ; pour une variable non-globale quelconque
  (lambda () (do ((i 0 (1+ i)) (env (cdr *env*) (cdr env)))
                 ((= i a) (car env)))))

(define (gen-ref-loc-1) ; pour la premiere variable locale
  (lambda () (cadr *env*)))

(define (gen-ref-loc-2) ; pour la deuxieme variable locale
  (lambda () (caddr *env*)))

(define (gen-ref-loc-3) ; pour la troisieme variable locale
  (lambda () (cadddr *env*)))

; --- generation de code pour une affectation ---

(define (gen-set-glo a b) ; pour une variable globale
  (lambda () (set-cdr! a (b))))

(define (gen-set a b)     ; pour une variable non-globale quelconque
  (lambda () (do ((i 0 (1+ i)) (env (cdr *env*) (cdr env)))
                 ((= i a) (set-car! env (b))))))

(define (gen-set-loc-1 a) ; pour la premiere variable locale
  (lambda () (set-car! (cdr *env*) (a))))

(define (gen-set-loc-2 a) ; pour la deuxieme variable locale
  (lambda () (set-car! (cddr *env*) (a))))

(define (gen-set-loc-3 a) ; pour la troisieme variable locale
  (lambda () (set-car! (cdddr *env*) (a))))

; --- generation de code pour une forme speciale 'if' ---

(define (gen-tst a b c)
  (lambda () (if (a) (b) (c))))

; --- generation de code pour une application de fonction ---

(define (gen-ap0 a)     ; application de fonction quelconque
  (lambda () ((a))))

(define (gen-ap1 a b)
  (lambda () ((a) (b))))

(define (gen-ap2 a b c)
  (lambda () ((a) (b) (c))))

(define (gen-ap3 a b c d)
  (lambda () ((a) (b) (c) (d))))
```

```
(define (gen-ap0-glo a)  ; application de fonction provenant d'une variable globale
  (lambda () ((cdr a))))

(define (gen-ap1-glo a b)
  (lambda () ((cdr a) (b))))

(define (gen-ap2-glo a b c)
  (lambda () ((cdr a) (b) (c))))

(define (gen-ap3-glo a b c d)
  (lambda () ((cdr a) (b) (c) (d))))

; --- generation de code pour une forme speciale 'lambda' ---

(define (gen-fn0 a)      ; pour une lambda-expression sans parametre 'reste'
  (lambda () (let ((def (cdr *env*)))
               (lambda () (set! *env* (cons *env* def)) (a)))))

(define (gen-fn1 a)
  (lambda () (let ((def (cdr *env*)))
               (lambda (x) (set! *env* (cons *env* (cons x def))) (a)))))

(define (gen-fn2 a)
  (lambda () (let ((def (cdr *env*)))
               (lambda (x y) (set! *env* (cons *env* (cons x (cons y def)))) (a)))))

(define (gen-fn3 a)
  (lambda () (let ((def (cdr *env*)))
               (lambda (x y z) (set! *env* (cons *env* (cons x (cons y (cons z def))))) (a)))))

(define (gen-fn1/rest a) ; pour une lambda-expression avec parametre 'reste'
  (lambda () (let ((def (cdr *env*)))
               (lambda x (set! *env* (cons *env* (cons x def))) (a)))))

(define (gen-fn2/rest a)
  (lambda () (let ((def (cdr *env*)))
               (lambda (x . y) (set! *env* (cons *env* (cons x (cons y def)))) (a)))))

(define (gen-fn3/rest a)
  (lambda () (let ((def (cdr *env*)))
               (lambda (x y . z) (set! *env* (cons *env* (cons x (cons y (cons z def))))) (a)))))

; --- fonctions de generation de code pour les evaluations terminales du corps d'une fonction ---

; --- generation de code pour une constante ---

(define (gen-cst* a) ; constante quelconque
  (lambda () (set! *env* (car *env*)) a))

(define (gen-1*)     ; pour la constante 1
  (lambda () (set! *env* (car *env*)) 1))

(define (gen-2*)     ; pour la constante 2
  (lambda () (set! *env* (car *env*)) 2))

(define (gen-null*)  ; pour la constante #!null
  (lambda () (set! *env* (car *env*)) #!null))

; --- generation de code pour une reference de variable ---

(define (gen-ref-glo* a) ; pour une variable globale
  (lambda () (set! *env* (car *env*)) (cdr a)))

(define (gen-ref* a)     ; pour une variable non-globale quelconque
  (lambda () (do ((i 0 (1+ i)) (env (cdr *env*) (cdr env)))
                 ((= i a) (set! *env* (car *env*)) (car env)))))

(define (gen-ref-loc-1*) ; pour la premiere variable locale
  (lambda () (let ((val (cadr *env*))) (set! *env* (car *env*)) val)))

(define (gen-ref-loc-2*) ; pour la deuxieme variable locale
  (lambda () (let ((val (caddr *env*))) (set! *env* (car *env*)) val)))

(define (gen-ref-loc-3*) ; pour la troisieme variable locale
  (lambda () (let ((val (cadddr *env*))) (set! *env* (car *env*)) val)))
```

```
; --- generation de code pour une affectation ---

(define (gen-set-glo* a b) ; pour une variable globale
  (lambda () (set! *env* (car *env*)) (set-cdr! a (b))))

(define (gen-set* a b)     ; pour une variable non-globale quelconque
  (lambda () (do ((i 0 (1+ i)) (env (cdr *env*) (cdr env)))
                 ((= i a) (set! *env* (car *env*)) (set-car! env (b))))))

(define (gen-set-loc-1* a) ; pour la premiere variable locale
  (lambda () (set! *env* (car *env*)) (set-car! (cdr *env*) (a))))

(define (gen-set-loc-2* a) ; pour la deuxieme variable locale
  (lambda () (set! *env* (car *env*)) (set-car! (cddr *env*) (a))))

(define (gen-set-loc-3* a) ; pour la troisieme variable locale
  (lambda () (set! *env* (car *env*)) (set-car! (cdddr *env*) (a))))

; --- generation de code pour une application de fonction ---

(define (gen-ap0* a)     ; application de fonction quelconque
  (lambda () (let ((w (a))) (set! *env* (car *env*)) (w))))

(define (gen-ap1* a b)
  (lambda () (let ((w (a)) (x (b))) (set! *env* (car *env*)) (w x))))

(define (gen-ap2* a b c)
  (lambda () (let ((w (a)) (x (b)) (y (c))) (set! *env* (car *env*)) (w x y))))

(define (gen-ap3* a b c d)
  (lambda () (let ((w (a)) (x (b)) (y (c)) (z (d))) (set! *env* (car *env*)) (w x y z))))

(define (gen-ap0-glo* a)  ; application de fonction provenant d'une variable globale
  (lambda () (set! *env* (car *env*)) ((cdr a))))

(define (gen-ap1-glo* a b)
  (lambda () (let ((x (b))) (set! *env* (car *env*)) ((cdr a) x))))

(define (gen-ap2-glo* a b c)
  (lambda () (let ((x (b)) (y (c))) (set! *env* (car *env*)) ((cdr a) x y))))

(define (gen-ap3-glo* a b c d)
  (lambda () (let ((x (b)) (y (c)) (z (d))) (set! *env* (car *env*)) ((cdr a) x y z))))

; --- generation de code pour une forme speciale 'lambda' ---

(define (gen-fn0* a)      ; pour une lambda-expression sans parametre 'reste'
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda () (set! *env* (cons *env* def)) (a)))))

(define (gen-fn1* a)
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda (x) (set! *env* (cons *env* (cons x def))) (a)))))

(define (gen-fn2* a)
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda (x y) (set! *env* (cons *env* (cons x (cons y def)))) (a)))))

(define (gen-fn3* a)
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda (x y z) (set! *env* (cons *env* (cons x (cons y (cons z def))))) (a)))))

(define (gen-fn1/rest* a) ; pour une lambda-expression avec parametre 'reste'
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda x (set! *env* (cons *env* (cons x def))) (a)))))

(define (gen-fn2/rest* a)
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda (x . y) (set! *env* (cons *env* (cons x (cons y def)))) (a)))))

(define (gen-fn3/rest* a)
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda (x y . z) (set! *env* (cons *env* (cons x (cons y (cons z def))))) (a)))))
```

```
; --- definition de variables globales ---

(define (define-global var val)
  (if (assq var *env-glo*)
    (set-cdr! (assq var *env-glo*) val)
    (set! *env-glo* (cons (cons var val) *env-glo*))))

(define *env-glo* (list (cons 'define define-global)))
(define-global 'cons  cons )
(define-global 'car   car  )
(define-global 'cdr   cdr  )
(define-global 'null? null?)
(define-global 'not   not  )
(define-global '<     <    )
(define-global '-1+   -1+  )
(define-global '+     +    )
(define-global '-     -    )

; --- pour evaluer, on compile une lambda-expression contenant l'expression a evaluer et ---
; --- on appelle la fonction resultante                                                  ---

(define (evaluer expr)
  ((compile (list 'lambda '() expr))))

(define *env* '(bidon)) ; environnement courant

;_____________________________________________________________________________________________
;
; Exemples d'utilisation du compilateur:
;
; a) pour evaluer:
;
;      (define fib
;        (lambda (x)
;          (if (< x 2) x (+ (fib (- x 1)) (fib (- x 2))))))
;
;    il faut entrer:
;
;      (evaluer '(define 'fib
;                  (lambda (x)
;                    (if (< x 2) x (+ (fib (- x 1)) (fib (- x 2)))))))
;
; b) pour evaluer:
;
;      (fib 20)
;
;    il faut entrer:
;
;      (evaluer '(fib 20))
```

Appendice E

Tests du chapitre III

Nous donnons ici la définition des diverses fonctions utilisées pour les tests de performance du troisième chapitre ainsi que l'appel dont le temps d'exécution a été mesuré.

fib

définition:

```
(define fib
  (lambda (x)
    (if (< x 2) x (+ (fib (- x 1)) (fib (- x 2))))))
```

appel:

```
(fib 20)
```

tak

définition:

```
(define tak
  (lambda (x y z)
    (if (not (< y x)) z
      (tak (tak (-1+ x) y z)
           (tak (-1+ y) z x)
           (tak (-1+ z) x y)))))
```

appel:

```
(tak 18 12 6)
```

trier

définition:

```
(define trier
  (lambda (liste)
    (if (null? liste) '() (trier-aux (cdr liste) '() (car liste)))))

(define trier-aux
  (lambda (liste reste min)
    (if (null? liste)
      (cons min (trier reste))
      (if (< (car liste) min)
        (trier-aux (cdr liste) (cons min reste) (car liste))
        (trier-aux (cdr liste) (cons (car liste) reste) min)))))
```

appel:

```
(trier '(3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 4
         3 3 8 3 2 7 9 5 0 2 8     2 7 1 8 2 8 1 8 2 8 4
         5 9 0 4 5 2 3 5 3 6 0 2 8 7 4 7 1 3 5 2 6 6 2 4)) ; pi et e
```

Appendice F

Exemple de génération de code en SIMULA 67

```
!_____________________________________________________________________;
!                                                                     ;
!             Exemple de generation de code en SIMULA 67              ;
!                                                                     ;
!_____________________________________________________________________;

BEGIN

  ! --- Definition des environnements --- ;

  CLASS ENVIRONNEMENT(VAR,VAL,ENV);
    CHARACTER VAR;
    REF(DONNEE) VAL;
    REF(ENVIRONNEMENT) ENV;
  BEGIN

    REF(DONNEE) PROCEDURE VALEUR(X); CHARACTER X;
      VALEUR :- IF X = VAR THEN VAL ELSE ENV.VALEUR(X);

    PROCEDURE AFFECTER(X,Y); CHARACTER X; REF(DONNEE) Y;
      IF X = VAR THEN VAL :- Y ELSE ENV.AFFECTER(X,Y);

  END;


  ! --- Types de donnees (i.e. entier, booleen et fonction) --- ;

  CLASS DONNEE;
  BEGIN
  END;

  DONNEE CLASS ENTIER(VAL); INTEGER VAL;
  BEGIN
  END;

  DONNEE CLASS BOOLEEN(VAL); BOOLEAN VAL;
  BEGIN
  END;

  DONNEE CLASS FONCTION;
    VIRTUAL : REF(DONNEE) PROCEDURE APPELER;
  BEGIN
  END;


  ! --- Fonctions de generation de code --- ;

  FONCTION CLASS GEN_CST(A); REF(DONNEE) A;
  BEGIN
    REF(DONNEE) PROCEDURE APPELER(ENV); REF(ENVIRONNEMENT) ENV;
      APPELER :- A;
  END;

  FONCTION CLASS GEN_REF(A); CHARACTER A;
  BEGIN
    REF(DONNEE) PROCEDURE APPELER(ENV); REF(ENVIRONNEMENT) ENV;
      APPELER :- ENV.VALEUR(A);
  END;

  FONCTION CLASS GEN_TST(A,B,C); REF(FONCTION) A,B,C;
  BEGIN
    REF(DONNEE) PROCEDURE APPELER(ENV); REF(ENVIRONNEMENT) ENV;
      APPELER :- IF A.APPELER(ENV) QUA BOOLEEN.VAL THEN B.APPELER(ENV)
                                                   ELSE C.APPELER(ENV);
  END;

  FONCTION CLASS GEN_AP1(A,B); REF(FONCTION) A,B;
  BEGIN
    REF(DONNEE) PROCEDURE APPELER(ENV); REF(ENVIRONNEMENT) ENV;
      APPELER :- A.APPELER(ENV) QUA FONCTION.APPELER( B.APPELER(ENV) );
  END;

  FONCTION CLASS GEN_AP2(A,B,C); REF(FONCTION) A,B,C;
  BEGIN
    REF(DONNEE) PROCEDURE APPELER(ENV); REF(ENVIRONNEMENT) ENV;
      APPELER :- A.APPELER(ENV) QUA FONCTION.APPELER( B.APPELER(ENV),
                                                      C.APPELER(ENV) );
  END;
```

```
  FONCTION CLASS GEN_FN1(A,B); REF(FONCTION) A; CHARACTER B;
  BEGIN
    REF(DONNEE) PROCEDURE APPELER(ENV); REF(ENVIRONNEMENT) ENV;
      APPELER :- NEW FN1( A, B, ENV );
  END;

  GEN_FN1 CLASS FN1(ENV); REF(ENVIRONNEMENT) ENV;
  BEGIN
    REF(DONNEE) PROCEDURE APPELER(X); REF(DONNEE) X;
      APPELER :- A.APPELER( NEW ENVIRONNEMENT(B,X,ENV) );
  END;


  ! --- Les fonctions predefinies --- ;

  FONCTION CLASS PLUS_PETIT;
  BEGIN
    REF(DONNEE) PROCEDURE APPELER(X,Y); REF(ENTIER) X,Y;
      APPELER :- NEW BOOLEEN( X.VAL < Y.VAL );
  END;

  FONCTION CLASS ADDITIONNER;
  BEGIN
    REF(DONNEE) PROCEDURE APPELER(X,Y); REF(ENTIER) X,Y;
      APPELER :- NEW ENTIER( X.VAL + Y.VAL );
  END;

  FONCTION CLASS SOUSTRAIRE;
  BEGIN
    REF(DONNEE) PROCEDURE APPELER(X,Y); REF(ENTIER) X,Y;
      APPELER :- NEW ENTIER( X.VAL - Y.VAL );
  END;


  ! --- Environnement global --- ;

  REF(ENVIRONNEMENT) ENV_GLO;


! --- Declaration des fonctions dans l'environnement global --- ;

ENV_GLO :- NEW ENVIRONNEMENT( 'F', NONE,
           NEW ENVIRONNEMENT( '<', NEW PLUS_PETIT,
           NEW ENVIRONNEMENT( '+', NEW ADDITIONNER,
           NEW ENVIRONNEMENT( '-', NEW SOUSTRAIRE, NONE ) ) ) );

! --- Generation de code pour la fonction fibonacci:          --- ;
! --- (lambda (x) (if (< x 2) x (+ (f (- x 1)) (f (- x 2))))) --- ;

ENV_GLO.AFFECTER( 'F',
        NEW GEN_FN1(
            NEW GEN_TST(
                NEW GEN_AP2(
                    NEW GEN_REF( '<' ),
                    NEW GEN_REF( 'X' ),
                    NEW GEN_CST( NEW ENTIER(2) ) ),
                NEW GEN_REF( 'X' ),
                NEW GEN_AP2(
                    NEW GEN_REF( '+' ),
                    NEW GEN_AP1(
                        NEW GEN_REF( 'F' ),
                        NEW GEN_AP2(
                            NEW GEN_REF( '-' ),
                            NEW GEN_REF( 'X' ),
                            NEW GEN_CST( NEW ENTIER(1) ) ) ),
                    NEW GEN_AP1(
                        NEW GEN_REF( 'F' ),
                        NEW GEN_AP2(
                            NEW GEN_REF( '-' ),
                            NEW GEN_REF( 'X' ),
                            NEW GEN_CST( NEW ENTIER(2) ) ) ) ) ),
            'X' ).APPELER( ENV_GLO ) );

! --- Calcul de fib(10) --- ;

OUTINT( ENV_GLO.VALEUR('F') QUA
        FONCTION.APPELER( NEW ENTIER(10) ) QUA
        ENTIER.VAL, 4 );

OUTIMAGE;   ! affiche: 55 ;

END;
```

Appendice G

Compilateur Scheme optimisant pour MC68000

```
;_____________________________________________________________________________________________
;
;                          Compilateur Scheme optimisant pour MC68000
;
; Afin de simplifier le compilateur, les parametres 'reste' ne sont pas traites, seulement
; un petit ensemble de fonctions primitives est disponible, les types de donnees sont limites,
; il n'y a pas de 'garbage collector' et pour ainsi dire aucune verification d'erreur n'est
; effectuee (aussi bien a la compilation qu'a l'execution).  La fonction principale du
; compilateur est 'compiler-fichier' a deux arguments.  Le premier designe le fichier
; contenant le programme a compiler et le deuxieme designe le fichier dans lequel sera
; ecrit le code assembleur resultant.  Celui-ci devra etre assemble avec un fichier qui
; contient l'interface necessaire pour la machine cible et la definition de certains macros
; emis dans le code assembleur.
;
; Voici un exemple d'utilisation du compilateur: (compiler-fichier "fib20.scm" "fib20.asm")
;
;_____________________________________________________________________________________________


; --- variables globales ---

(define *fermetures* '()) ; liste d'association des fermetures generees (pour les partager)
(define *env-glo* '())    ; liste d'association representant les variables globales


(define (compiler-fichier entree sortie)
  (set! *fermetures* '())
  (set! *env-glo* '())
  (define-global 'cons    fn/cons   )   ; definition des fonctions primitives
  (define-global 'car     fn/car    )
  (define-global 'cdr     fn/cdr    )
  (define-global 'null?   fn/null?  )
  (define-global 'not     fn/not    )
  (define-global '<       fn/<      )
  (define-global '=       fn/=      )
  (define-global '-1+     fn/-1+    )
  (define-global '+       fn/+      )
  (define-global '-       fn/-      )
  (define-global 'closure fn/closure)
  (define-global 'box     fn/box    )
  (define-global 'fetch   fn/fetch  )
  (define-global 'store   fn/store  )
  (let ((code (call-with-input-file entree lire-et-compiler)))
    (call-with-output-file sortie
      (lambda (fichier) (desassembler code fichier)))))

(define (lire-et-compiler fichier)
  (let ((expr (read fichier)))
    (if (not (eof-object? expr))
      (let ((code (compiler expr)))
        (cons code (lire-et-compiler fichier))))))

(define (compiler expr)
  (gen (lambda->epsilon+closure (expansion expr)) '() 0 0))

(define (gen expr parms emp cont)
  (cond ((symbol? expr)            (ref expr parms emp cont))
        ((not (pair? expr))        (cst expr cont))
        ((eq? (car expr) 'quote)   (cst (cadr expr) cont))
        ((eq? (car expr) 'set!)    (set (cadr expr) (caddr expr) parms emp cont))
        ((eq? (car expr) 'if)      (tst (cadr expr) (caddr expr) (cadddr expr) parms emp cont))
        ((eq? (car expr) 'epsilon) (eps (cadr expr) (caddr expr) cont))
        (else                      (app expr parms emp cont))))

(define (variable symb parms emp)
  (let ((x (memq symb parms)))
    (if x
      (- emp (1+ (- (length parms) (length x)))) ; distance de la variable du dessus de la pile
      (begin
        (if (not (assq symb *env-glo*)) (define-global symb indefini))
        symb))))

(define (global? var)
  (symbol? var))

(define (cst val cont)
  (gen-cst val cont))

(define (ref symb parms emp cont)
  (let ((var (variable symb parms emp)))
    (if (global? var)
      (gen-ref-glo var cont)
      (gen-ref-loc (+ var (if (= cont 0) 0 1)) cont))))
```

```
(define (set symb expr parms emp cont)
  (let ((var (variable symb parms emp))
        (val (gen expr parms (+ emp (if (= cont 0) 2 3)) 0)))
    (if (global? var)
      (gen-set-glo var val cont)
      (gen-set-loc (+ var (if (= cont 0) 0 1)) val cont))))

(define (tst pre con alt parms emp cont)
  (gen-tst (gen pre parms (+ emp 3) 0)
           (gen con parms emp cont)
           (gen alt parms emp cont)))

(define (eps parms corps cont)
  (gen-cst (gen corps parms (length parms) (length parms)) cont))

(define (app expr parms emp cont)
  (let ((emp* (+ emp (length expr) (if (= cont 0) 0 1))))
    (let ((var (and (symbol? (car expr)) (variable (car expr) parms emp*)))
          (args (map (lambda (x) (gen x parms emp* 0)) (cdr expr))))
      (if (global? var)
        (case (length args)
          ((0)  (gen-ap0-glo var                                      cont))
          ((1)  (gen-ap1-glo var (car args)                           cont))
          ((2)  (gen-ap2-glo var (car args) (cadr args)               cont))
          ((3)  (gen-ap3-glo var (car args) (cadr args) (caddr args) cont))
          (else (error "trop d'arguments dans une application de fonction")))
        (case (length args)
          ((0)  (gen-ap0 (gen (car expr) parms emp* 0)                                      cont))
          ((1)  (gen-ap1 (gen (car expr) parms emp* 0) (car args)                           cont))
          ((2)  (gen-ap2 (gen (car expr) parms emp* 0) (car args) (cadr args)               cont))
          ((3)  (gen-ap3 (gen (car expr) parms emp* 0) (car args) (cadr args) (caddr args) cont))
          (else (error "trop d'arguments dans une application de fonction")))))))

; --- definition de variables globales ---

(define (define-global var val)
  (if (assq var *env-glo*)
    (set-cdr! (assq var *env-glo*) val)
    (set! *env-glo* (cons (cons var val) *env-glo*))))

(define (valeur-globale var)
  (cdr (assq var *env-glo*)))
```

```
;_____________________________________________________________________________________________
;
;                          Definition des fonctions primitives en MC68000
;
; Note: on suppose qu'a l'execution les registres contiennent les valeurs suivantes:
;          D0 = #!false
;          D1 = #!true
;          D2 = #!null
;          A0 = pointeur d'allocation des donnees en memoire (haut de la memoire en descendant)
;
;       a l'entree d'une fonction:
;          D3        = nb d'arguments passes a la fonction
;          (SP)      = dernier argument
;          (SP+4)    = avant dernier argument
;          ...
;          (SP+4*D3) = adresse de continuation de la fonction
;
;       a la sortie d'une fonction:
;          A1 = resultat de la fonction
;          (la continuation et les arguments de la fonction ne sont plus sur la pile)
;
;       @1, @2,... sont des etiquettes locales pour l'assembleur
;_____________________________________________________________________________________________


(define (primitive . code)
  (cons '<primitive> code))

(define fn/cons
  (primitive
    "        MOVE.L  (SP)+,-(A0)" ; allouer le champ cdr et depiler le deuxieme argument
    "        MOVE.L  A0,A1"       ; garder le pointeur vers le millieu du doublet
    "        MOVE.L  (SP)+,-(A0)" ; allouer le champ car et depiler le premier argument
    "        RTS"                 ; brancher a la continuation
  ))

(define fn/car
  (primitive
    "        MOVE.L  (SP)+,A1"  ; depiler l'argument
    "        MOVE.L  -4(A1),A1" ; resultat = champ car du doublet
    "        RTS"               ; brancher a la continuation
  ))

(define fn/cdr
  (primitive
    "        MOVE.L  (SP)+,A1" ; depiler l'argument
    "        MOVE.L  (A1),A1"  ; resultat = champ cdr du doublet
    "        RTS"              ; brancher a la continuation
  ))

(define fn/null?
  (primitive
    "        CMP.L   (SP)+,D2" ; depiler et comparer l'argument avec #!null
    "        BEQ.S   @1"       ; si l'argument est #!null brancher a @1
    "        MOVE.L  D0,A1"    ; resultat faux (D0 contient toujours #!false)
    "        RTS"              ; brancher a la continuation
    "@1      MOVE.L  D1,A1"    ; resultat vrai (D1 contient toujours #!true)
    "        RTS"              ; brancher a la continuation
  ))

(define fn/not
  (primitive
    "        CMP.L   (SP)+,D0" ; depiler et comparer l'argument avec #!false
    "        BEQ.S   @1"       ; si l'argument est #!false brancher a @1
    "        MOVE.L  D0,A1"    ; resultat faux (D0 contient toujours #!false)
    "        RTS"              ; brancher a la continuation
    "@1      MOVE.L  D1,A1"    ; resultat vrai (D1 contient toujours #!true)
    "        RTS"              ; brancher a la continuation
  ))

(define fn/<
  (primitive
    "        CMPM.L  (SP)+,(SP)+" ; comparer et depiler les deux arguments
    "        BLT.S   @1"          ; si arg1 < arg2 brancher a @1
    "        MOVE.L  D0,A1"       ; resultat faux (D0 contient toujours #!false)
    "        RTS"                 ; brancher a la continuation
    "@1      MOVE.L  D1,A1"       ; resultat vrai (D1 contient toujours #!true)
    "        RTS"                 ; brancher a la continuation
  ))
```

```
(define fn/=
  (primitive
    "        CMPM.L  (SP)+,(SP)+" ; comparer et depiler les deux arguments
    "        BEQ.S   @1"          ; si arg1 = arg2 brancher a @1
    "        MOVE.L  D0,A1"       ; resultat faux (D0 contient toujours #!false)
    "        RTS"                 ; brancher a la continuation
    "@1      MOVE.L  D1,A1"       ; resultat vrai (D1 contient toujours #!true)
    "        RTS"                 ; brancher a la continuation
  ))

(define fn/-1+
  (primitive
    "        MOVE.L  (SP)+,A1" ; depiler l'argument
    "        SUBQ.L  #1,A1"    ; soustraire 1
    "        RTS"              ; brancher a la continuation
  ))

(define fn/+
  (primitive
    "        MOVE.L  (SP)+,A1" ; depiler le deuxieme argument
    "        ADD.L   (SP)+,A1" ; additionner le premier argument
    "        RTS"              ; brancher a la continuation
  ))

(define fn/-
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler le deuxieme argument
    "        MOVE.L  (SP)+,A1" ; depiler le premier argument
    "        SUB.L   A2,A1"    ; soustraire le deuxieme argument
    "        RTS"              ; brancher a la continuation
  ))

(define fn/closure ; fonction de generation de fermetures (note: c'est une fonction n-aire)
  (primitive
    "        MOVE.L  (SP)+,-(A0)"  ; mettre l'adresse du corps de la fonction
    "        MOVE.W  #$4EF9,-(A0)" ; et generer le code pour un JMP xxx
    "        SUBQ.W  #2,D3"        ; un argument de moins
    "        BMI.S   @2"           ; ne pas generer de MOVE.L s'il n'y a pas de valeurs
    "@1      MOVE.L  (SP)+,-(A0)"  ; mettre la valeur presente de la variable fermee correspondante
    "        MOVE.W  #$2F3C,-(A0)" ; et generer le code pour un MOVE.L #xxx,-(SP)
    "        DBRA    D3,@1"        ; repeter pour chaque variable fermee
    "@2      MOVE.L  A0,A1"        ; resultat = debut du bout de code genere
    "        RTS"                  ; brancher a la continuation
  ))

(define fn/box
  (primitive
    "        MOVE.L  (SP)+,-(A0)" ; allouer la boite et depiler le deuxieme argument
    "        MOVE.L  A0,A1"       ; garder le pointeur vers la boite
    "        RTS"                 ; brancher a la continuation
  ))

(define fn/fetch
  (primitive
    "        MOVE.L  (SP)+,A1" ; depiler l'argument
    "        MOVE.L  (A1),A1"  ; resultat = contenu de la boite
    "        RTS"              ; brancher a la continuation
  ))

(define fn/store
  (primitive
    "        MOVE.L  (SP)+,A1" ; depiler le deuxieme argument (la valeur a deposer)
    "        MOVE.L  (SP)+,A2" ; depiler le premier argument (la boite)
    "        MOVE.L  A1,(A2)"  ; affecter la valeur a la boite
    "        RTS"              ; brancher a la continuation
  ))
```

```
;_____________________________________________________________________________________________
;
;                          Fonctions de generation de code
;
;_____________________________________________________________________________________________


(define (closure . parms)
  (if (null? (cdr parms))
    (car parms)
    (let ((x (assoc parms *fermetures*))) ; verifier si cette fermeture n'a pas deja ete creee
      (if x
        (cdr x)
        (begin
          (set! *fermetures* (cons (cons parms (cons '<closure> parms)) *fermetures*))
          (cdr (assoc parms *fermetures*)))))))

(define (po x) ; nb pointeurs --> nb octets
  (* x 4))


; --- generation de code pour une constante ---

(define (gen-cst a cont)
  (case a
    ((1)      (if (= cont 0) (closure   corps/1   ) (closure (po cont)   corps/1*   )))
    ((2)      (if (= cont 0) (closure   corps/2   ) (closure (po cont)   corps/2*   )))
    ((#!null) (if (= cont 0) (closure   corps/null) (closure (po cont)   corps/null*)))
    (else     (if (= cont 0) (closure a corps/cst ) (closure (po cont) a corps/cst* )))))

(define corps/1
  (primitive
    "        MOVE.L  #1,A1"    ; resultat = entier 1
    "        RTS"              ; brancher a la continuation
  ))

(define corps/1*
  (primitive
    "        MOVE.L  #1,A1"    ; resultat = entier 1
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        RTS"              ; brancher a la continuation de la fonction
  ))

(define corps/2
  (primitive
    "        MOVE.L  #2,A1"    ; resultat = entier 2
    "        RTS"              ; brancher a la continuation
  ))

(define corps/2*
  (primitive
    "        MOVE.L  #2,A1"    ; resultat = entier 2
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        RTS"              ; brancher a la continuation de la fonction
  ))

(define corps/null
  (primitive
    "        MOVE.L  D2,A1"    ; resultat = #!null
    "        RTS"              ; brancher a la continuation
  ))

(define corps/null*
  (primitive
    "        MOVE.L  D2,A1"    ; resultat = #!null
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        RTS"              ; brancher a la continuation de la fonction
  ))

(define corps/cst
  (primitive
    "        MOVE.L  (SP)+,A1" ; resultat = variable fermee 'a'
    "        RTS"              ; brancher a la continuation
  ))

(define corps/cst*
  (primitive
    "        MOVE.L  (SP)+,A1" ; resultat = variable fermee 'a'
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        RTS"              ; brancher a la continuation de la fonction
  ))
```

```
; --- generation de code pour une reference de variable ---

(define (gen-ref-glo a cont)
  (if (= cont 0) (closure a corps/ref-glo) (closure (po cont) a corps/ref-glo*)))

(define corps/ref-glo
  (primitive
    "        MOVE.L  (SP)+,A1" ; depiler la variable fermee 'a' dans A1
    "        MOVE.L  (A1),A1"  ; resultat = champ cdr du doublet associe a la variable
    "        RTS"              ; brancher a la continuation
  ))

(define corps/ref-glo*
  (primitive
    "        MOVE.L  (SP)+,A1" ; depiler la variable fermee 'a' dans A1
    "        MOVE.L  (A1),A1"  ; resultat = champ cdr du doublet associe a la variable
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        RTS"              ; brancher a la continuation de la fonction
  ))

(define (gen-ref-loc a cont)
  (case a
    ((1)  (if (= cont 0) (closure        corps/ref-loc-1) (closure (po cont)        corps/ref-loc-1*)))
    ((2)  (if (= cont 0) (closure        corps/ref-loc-2) (closure (po cont)        corps/ref-loc-2*)))
    ((3)  (if (= cont 0) (closure        corps/ref-loc-3) (closure (po cont)        corps/ref-loc-3*)))
    ((4)  (if (= cont 0) (closure        corps/ref-loc-4) (closure (po cont)        corps/ref-loc-4*)))
    ((5)  (if (= cont 0) (closure        corps/ref-loc-5) (closure (po cont)        corps/ref-loc-5*)))
    (else (if (= cont 0) (closure (po a) corps/ref-loc  ) (closure (po cont) (po a) corps/ref-loc*  )))))

(define corps/ref-loc-1
  (primitive
    "        MOVE.L  4(SP),A1"  ; resultat = 1er pointeur sur la pile
    "        RTS"               ; brancher a la continuation
  ))

(define corps/ref-loc-1*
  (primitive
    "        MOVE.L  4(SP),A1"  ; resultat = 1er pointeur sur la pile
    "        ADD.L   (SP)+,SP"  ; desallouer les parametres de la fonction
    "        RTS"               ; brancher a la continuation de la fonction
  ))

(define corps/ref-loc-2
  (primitive
    "        MOVE.L  8(SP),A1"  ; resultat = 2ieme pointeur sur la pile
    "        RTS"               ; brancher a la continuation
  ))

(define corps/ref-loc-2*
  (primitive
    "        MOVE.L  8(SP),A1"  ; resultat = 2ieme pointeur sur la pile
    "        ADD.L   (SP)+,SP"  ; desallouer les parametres de la fonction
    "        RTS"               ; brancher a la continuation de la fonction
  ))

(define corps/ref-loc-3
  (primitive
    "        MOVE.L  12(SP),A1" ; resultat = 3ieme pointeur sur la pile
    "        RTS"               ; brancher a la continuation
  ))

(define corps/ref-loc-3*
  (primitive
    "        MOVE.L  12(SP),A1" ; resultat = 3ieme pointeur sur la pile
    "        ADD.L   (SP)+,SP"  ; desallouer les parametres de la fonction
    "        RTS"               ; brancher a la continuation de la fonction
  ))

(define corps/ref-loc-4
  (primitive
    "        MOVE.L  16(SP),A1" ; resultat = 4ieme pointeur sur la pile
    "        RTS"               ; brancher a la continuation
  ))

(define corps/ref-loc-4*
  (primitive
    "        MOVE.L  16(SP),A1" ; resultat = 4ieme pointeur sur la pile
    "        ADD.L   (SP)+,SP"  ; desallouer les parametres de la fonction
    "        RTS"               ; brancher a la continuation de la fonction
  ))
```

```
(define corps/ref-loc-5
  (primitive
    "        MOVE.L  20(SP),A1" ; resultat = 5ieme pointeur sur la pile
    "        RTS"               ; brancher a la continuation
  ))

(define corps/ref-loc-5*
  (primitive
    "        MOVE.L  20(SP),A1" ; resultat = 5ieme pointeur sur la pile
    "        ADD.L   (SP)+,SP"  ; desallouer les parametres de la fonction
    "        RTS"               ; brancher a la continuation de la fonction
  ))

(define corps/ref-loc
  (primitive
    "        MOVE.L  (SP)+,A2"     ; depiler la valeur de 'a' dans A2
    "        MOVE.L  (SP,A2.L),A1" ; resultat = 'a' ieme pointeur sur la pile
    "        RTS"                  ; brancher a la continuation
  ))

(define corps/ref-loc*
  (primitive
    "        MOVE.L  (SP)+,A2"     ; depiler la valeur de 'a' dans A2
    "        MOVE.L  (SP,A2.L),A1" ; resultat = 'a' ieme pointeur sur la pile
    "        ADD.L   (SP)+,SP"     ; desallouer les parametres de la fonction
    "        RTS"                  ; brancher a la continuation de la fonction
  ))


; --- generation de code pour une affectation ---

(define (gen-set-glo a b cont)
  (if (= cont 0) (closure a b corps/set-glo) (closure (po cont) a b corps/set-glo*)))

(define corps/set-glo
  (primitive
    "        MOVE.L  (SP)+,A1" ; depiler la valeur de 'b' dans A1
    "        JSR     (A1)"     ; evaluer la valeur a affecter
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'a' dans A2
    "        MOVE.L  A1,(A2)"  ; affecter le resultat dans la variable globale
    "        RTS"              ; brancher a la continuation
  ))

(define corps/set-glo*
  (primitive
    "        MOVE.L  (SP)+,A1" ; depiler la valeur de 'b' dans A1
    "        JSR     (A1)"     ; evaluer la valeur a affecter
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'a' dans A2
    "        MOVE.L  A1,(A2)"  ; affecter le resultat dans la variable globale
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        RTS"              ; brancher a la continuation de la fonction
  ))

(define (gen-set-loc a b cont)
  (if (= cont 0) (closure (po a) b corps/set-loc) (closure (po cont) (po a) b corps/set-loc*)))

(define corps/set-loc
  (primitive
    "        MOVE.L  (SP)+,A1"      ; depiler la valeur de 'b' dans A1
    "        JSR     (A1)"          ; evaluer la valeur a affecter
    "        MOVE.L  (SP)+,A2"      ; depiler la valeur de 'a' dans A2
    "        MOVE.L  A1,(SP,A2.L)"  ; affecter au 'a' ieme pointeur sur la pile
    "        RTS"                   ; brancher a la continuation
  ))

(define corps/set-loc*
  (primitive
    "        MOVE.L  (SP)+,A1"      ; depiler la valeur de 'b' dans A1
    "        JSR     (A1)"          ; evaluer la valeur a affecter
    "        MOVE.L  (SP)+,A2"      ; depiler la valeur de 'a' dans A2
    "        MOVE.L  A1,(SP,A2.L)"  ; affecter au 'a' ieme pointeur sur la pile
    "        ADD.L   (SP)+,SP"      ; desallouer les parametres de la fonction
    "        RTS"                   ; brancher a la continuation de la fonction
  ))
```

```
; --- generation de code pour une forme speciale 'if' ---

(define (gen-tst a b c)
  (closure c b a corps/tst))

(define corps/tst
  (primitive
    "        MOVE.L  (SP)+,A1" ; depiler la valeur de 'a' dans A1
    "        JSR     (A1)"     ; evaluer le predicat
    "        MOVE.L  (SP)+,A2" ; depiler la fermeture du consequent (i.e. 'b')
    "        MOVE.L  (SP)+,A3" ; depiler la fermeture de l'alternative (i.e. 'c')
    "        CMP.L   D0,A1"    ; comparer le resultat avec #!false
    "        BEQ.S   @1"       ; si le resultat est faux, evaluer l'alternative
    "        JMP     (A2)"     ; evaluer le consequent
    "@1      JMP     (A3)"     ; evaluer l'alternative
  ))


; --- generation de code pour une application de fonction ---

; --- application de fonction quelconque ---

(define (gen-ap0 a cont)
  (if (= cont 0) (closure a corps/ap0) (closure (po cont) a corps/ap0*)))

(define corps/ap0
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'a' dans A2
    "        JSR     (A2)"     ; evaluer la position fonctionnelle
    "        MOVEQ.L #0,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A1)"     ; brancher a la fonction
  ))

(define corps/ap0*
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'a' dans A2
    "        JSR     (A2)"     ; evaluer la position fonctionnelle
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        MOVEQ.L #0,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A1)"     ; brancher a la fonction
  ))

(define (gen-ap1 a b cont)
  (if (= cont 0) (closure a b corps/ap1) (closure (po cont) a b corps/ap1*)))

(define corps/ap1
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'b' dans A2
    "        JSR     (A2)"     ; evaluer le 1er argument
    "        MOVE.L  (SP),A2"  ; mettre la valeur de 'a' dans A2
    "        MOVE.L  A1,(SP)"  ; remplacer la valeur de 'a' par le 1er argument
    "        JSR     (A2)"     ; evaluer la position fonctionnelle
    "        MOVEQ.L #1,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A1)"     ; brancher a la fonction
  ))

(define corps/ap1*
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'b' dans A2
    "        JSR     (A2)"     ; evaluer le 1er argument
    "        MOVE.L  (SP),A2"  ; mettre la valeur de 'a' dans A2
    "        MOVE.L  A1,(SP)"  ; remplacer la valeur de 'a' par le 1er argument
    "        JSR     (A2)"     ; evaluer la position fonctionnelle
    "        MOVE.L  (SP)+,A2" ; depiler le 1er argument
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        MOVE.L  A2,-(SP)" ; empiler le 1er argument
    "        MOVEQ.L #1,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A1)"     ; brancher a la fonction
  ))
```

```
(define (gen-ap2 a b c cont)
  (if (= cont 0) (closure a b c corps/ap2) (closure (po cont) a b c corps/ap2*)))

(define corps/ap2
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'c' dans A2
    "        JSR     (A2)"     ; evaluer le 2ieme argument
    "        MOVE.L  (SP),A2"  ; mettre la valeur de 'b' dans A2
    "        MOVE.L  A1,(SP)"  ; remplacer la valeur de 'b' par le 2ieme argument
    "        JSR     (A2)"     ; evaluer le 1er argument
    "        MOVE.L  4(SP),A2" ; mettre la valeur de 'a' dans A2
    "        MOVE.L  A1,4(SP)" ; remplacer la valeur de 'a' par le 1er argument
    "        JSR     (A2)"     ; evaluer la position fonctionnelle
    "        MOVEQ.L #2,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A1)"     ; brancher a la fonction
  ))

(define corps/ap2*
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'c' dans A2
    "        JSR     (A2)"     ; evaluer le 2ieme argument
    "        MOVE.L  (SP),A2"  ; mettre la valeur de 'b' dans A2
    "        MOVE.L  A1,(SP)"  ; remplacer la valeur de 'b' par le 2ieme argument
    "        JSR     (A2)"     ; evaluer le 1er argument
    "        MOVE.L  4(SP),A2" ; mettre la valeur de 'a' dans A2
    "        MOVE.L  A1,4(SP)" ; remplacer la valeur de 'a' par le 1er argument
    "        JSR     (A2)"     ; evaluer la position fonctionnelle
    "        MOVE.L  (SP)+,A3" ; depiler le 2ieme argument
    "        MOVE.L  (SP)+,A2" ; depiler le 1er argument
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        MOVE.L  A2,-(SP)" ; empiler le 1er argument
    "        MOVE.L  A3,-(SP)" ; empiler le 2ieme argument
    "        MOVEQ.L #2,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A1)"     ; brancher a la fonction
  ))

(define (gen-ap3 a b c d cont)
  (if (= cont 0) (closure a b c d corps/ap3) (closure (po cont) a b c d corps/ap3*)))

(define corps/ap3
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'd' dans A2
    "        JSR     (A2)"     ; evaluer le 3ieme argument
    "        MOVE.L  (SP),A2"  ; mettre la valeur de 'c' dans A2
    "        MOVE.L  A1,(SP)"  ; remplacer la valeur de 'c' par le 3ieme argument
    "        JSR     (A2)"     ; evaluer le 2ieme argument
    "        MOVE.L  4(SP),A2" ; mettre la valeur de 'b' dans A2
    "        MOVE.L  A1,4(SP)" ; remplacer la valeur de 'b' par le 2ieme argument
    "        JSR     (A2)"     ; evaluer le 1er argument
    "        MOVE.L  8(SP),A2" ; mettre la valeur de 'a' dans A2
    "        MOVE.L  A1,8(SP)" ; remplacer la valeur de 'a' par le 1er argument
    "        JSR     (A2)"     ; evaluer la position fonctionnelle
    "        MOVEQ.L #3,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A1)"     ; brancher a la fonction
  ))

(define corps/ap3*
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'd' dans A2
    "        JSR     (A2)"     ; evaluer le 3ieme argument
    "        MOVE.L  (SP),A2"  ; mettre la valeur de 'c' dans A2
    "        MOVE.L  A1,(SP)"  ; remplacer la valeur de 'c' par le 3ieme argument
    "        JSR     (A2)"     ; evaluer le 2ieme argument
    "        MOVE.L  4(SP),A2" ; mettre la valeur de 'b' dans A2
    "        MOVE.L  A1,4(SP)" ; remplacer la valeur de 'b' par le 2ieme argument
    "        JSR     (A2)"     ; evaluer le 1er argument
    "        MOVE.L  8(SP),A2" ; mettre la valeur de 'a' dans A2
    "        MOVE.L  A1,8(SP)" ; remplacer la valeur de 'a' par le 1er argument
    "        JSR     (A2)"     ; evaluer la position fonctionnelle
    "        MOVE.L  (SP)+,A4" ; depiler le 3ieme argument
    "        MOVE.L  (SP)+,A3" ; depiler le 2ieme argument
    "        MOVE.L  (SP)+,A2" ; depiler le 1er argument
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        MOVE.L  A2,-(SP)" ; empiler le 1er argument
    "        MOVE.L  A3,-(SP)" ; empiler le 2ieme argument
    "        MOVE.L  A4,-(SP)" ; empiler le 3ieme argument
    "        MOVEQ.L #3,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A1)"     ; brancher a la fonction
  ))
```

```
; --- application de fonction provenant d'une variable globale ---

(define (gen-ap0-glo a cont)
  (if (= cont 0) (closure a corps/ap0-glo) (closure (po cont) a corps/ap0-glo*)))

(define corps/ap0-glo
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'a' dans A2
    "        MOVE.L  (A2),A2"  ; obtenir la valeur de la variable globale
    "        MOVEQ.L #0,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A2)"     ; brancher a la fonction
  ))

(define corps/ap0-glo*
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'a' dans A2
    "        MOVE.L  (A2),A2"  ; obtenir la valeur de la variable globale
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        MOVEQ.L #0,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A2)"     ; brancher a la fonction
  ))

(define (gen-ap1-glo a b cont)
  (if (= cont 0) (closure a b corps/ap1-glo) (closure (po cont) a b corps/ap1-glo*)))

(define corps/ap1-glo
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'b' dans A2
    "        JSR     (A2)"     ; evaluer le 1er argument
    "        MOVE.L  (SP),A2"  ; mettre la valeur de 'a' dans A2
    "        MOVE.L  A1,(SP)"  ; remplacer la valeur de 'a' par le 1er argument
    "        MOVE.L  (A2),A2"  ; obtenir la valeur de la variable globale
    "        MOVEQ.L #1,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A2)"     ; brancher a la fonction
  ))

(define corps/ap1-glo*
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'b' dans A2
    "        JSR     (A2)"     ; evaluer le 1er argument
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'a'
    "        MOVE.L  (A2),A2"  ; obtenir la valeur de la variable globale
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        MOVE.L  A1,-(SP)" ; empiler le 1er argument
    "        MOVEQ.L #1,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A2)"     ; brancher a la fonction
  ))

(define (gen-ap2-glo a b c cont)
  (if (= cont 0) (closure a b c corps/ap2-glo) (closure (po cont) a b c corps/ap2-glo*)))

(define corps/ap2-glo
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'c' dans A2
    "        JSR     (A2)"     ; evaluer le 2ieme argument
    "        MOVE.L  (SP),A2"  ; mettre la valeur de 'b' dans A2
    "        MOVE.L  A1,(SP)"  ; remplacer la valeur de 'b' par le 2ieme argument
    "        JSR     (A2)"     ; evaluer le 1er argument
    "        MOVE.L  4(SP),A2" ; mettre la valeur de 'a' dans A2
    "        MOVE.L  A1,4(SP)" ; remplacer la valeur de 'a' par le 1er argument
    "        MOVE.L  (A2),A2"  ; obtenir la valeur de la variable globale
    "        MOVEQ.L #2,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A2)"     ; brancher a la fonction
  ))

(define corps/ap2-glo*
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'c' dans A2
    "        JSR     (A2)"     ; evaluer le 2ieme argument
    "        MOVE.L  (SP),A2"  ; mettre la valeur de 'b' dans A2
    "        MOVE.L  A1,(SP)"  ; remplacer la valeur de 'b' par le 2ieme argument
    "        JSR     (A2)"     ; evaluer le 1er argument
    "        MOVE.L  (SP)+,A3" ; depiler le 2ieme argument
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'a'
    "        MOVE.L  (A2),A2"  ; obtenir la valeur de la variable globale
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        MOVE.L  A1,-(SP)" ; empiler le 1er argument
    "        MOVE.L  A3,-(SP)" ; empiler le 2ieme argument
    "        MOVEQ.L #2,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A2)"     ; brancher a la fonction
  ))
```

```
(define (gen-ap3-glo a b c d cont)
  (if (= cont 0) (closure a b c d corps/ap3-glo) (closure (po cont) a b c d corps/ap3-glo*)))

(define corps/ap3-glo
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'd' dans A2
    "        JSR     (A2)"     ; evaluer le 3ieme argument
    "        MOVE.L  (SP),A2"  ; mettre la valeur de 'c' dans A2
    "        MOVE.L  A1,(SP)"  ; remplacer la valeur de 'c' par le 3ieme argument
    "        JSR     (A2)"     ; evaluer le 2ieme argument
    "        MOVE.L  4(SP),A2" ; mettre la valeur de 'b' dans A2
    "        MOVE.L  A1,4(SP)" ; remplacer la valeur de 'b' par le 2ieme argument
    "        JSR     (A2)"     ; evaluer le 1er argument
    "        MOVE.L  8(SP),A2" ; mettre la valeur de 'a' dans A2
    "        MOVE.L  A1,8(SP)" ; remplacer la valeur de 'a' par le 1er argument
    "        MOVE.L  (A2),A2"  ; obtenir la valeur de la variable globale
    "        MOVEQ.L #3,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A2)"     ; brancher a la fonction
  ))

(define corps/ap3-glo*
  (primitive
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'd' dans A2
    "        JSR     (A2)"     ; evaluer le 3ieme argument
    "        MOVE.L  (SP),A2"  ; mettre la valeur de 'c' dans A2
    "        MOVE.L  A1,(SP)"  ; remplacer la valeur de 'c' par le 3ieme argument
    "        JSR     (A2)"     ; evaluer le 2ieme argument
    "        MOVE.L  4(SP),A2" ; mettre la valeur de 'b' dans A2
    "        MOVE.L  A1,4(SP)" ; remplacer la valeur de 'b' par le 2ieme argument
    "        JSR     (A2)"     ; evaluer le 1er argument
    "        MOVE.L  (SP)+,A4" ; depiler le 3ieme argument
    "        MOVE.L  (SP)+,A3" ; depiler le 2ieme argument
    "        MOVE.L  (SP)+,A2" ; depiler la valeur de 'a'
    "        MOVE.L  (A2),A2"  ; obtenir la valeur de la variable globale
    "        ADD.L   (SP)+,SP" ; desallouer les parametres de la fonction
    "        MOVE.L  A1,-(SP)" ; empiler le 1er argument
    "        MOVE.L  A3,-(SP)" ; empiler le 2ieme argument
    "        MOVE.L  A4,-(SP)" ; empiler le 3ieme argument
    "        MOVEQ.L #3,D3"    ; mettre le nombre d'arguments dans D3
    "        JMP     (A2)"     ; brancher a la fonction
  ))
```

```
;_____________________________________________________________________________________________
;
;                                Definition et expansion des macros
;
;_____________________________________________________________________________________________


(macro define-macro   ; macro de MacScheme pour faire le 'bootstrap' des macros du compilateur
  (lambda (forme)
    (let ((patron (cadr forme)) (corps (cddr forme)))
      (list 'define-macro-fn
        (list 'quote (car patron))
        (cons 'lambda (cons (cdr patron) corps))))))

(define (define-macro-fn mot-cle fn-transf)
  (put mot-cle 'macro fn-transf)
  mot-cle)


; --- definition des fonctions d'expansion pour chaque forme speciale de Scheme ---


; Le macro 'define-macro' effectue la transformation
;   (define-macro (a b…) A B…)
; -->
;   (define-macro-fn 'a (lambda (b…) A B…))

(define-macro (define-macro patron . corps)
  (list 'define-macro-fn
    (list 'quote (car patron))
    (cons 'lambda (cons (cdr patron) corps))))


; Le macro 'quote' effectue la transformation
;   (quote A)  -->  (<quote> A)

(define-macro (quote x)
  (list '<quote> x))


; Le macro 'set!' effectue la transformation
;   (set! a A)  -->  (<set!> a A)

(define-macro (set! var expr)
  (list '<set!> var expr))


; Le macro 'if' effectue les transformations
;   (if A B)  -->  (<if> A B ?)
; et
;   (if A B C)  -->  (<if> A B C)

(define-macro (if condition consequent . alternative)
  (list '<if> condition consequent
    (if (null? alternative) indefini (car alternative))))


; Le macro 'lambda' effectue les transformations
;   (lambda (a…) (define b A)… (define c B)… C D…)  -->
;   (<lambda> (a…) (letrec ((b A) (c B)…) C D))
; et
;   (lambda (a…) A B…)  -->  (<lambda> (a…) (begin A B…))

(define-macro (lambda parms . corps)
  (define (integrer-defines decl corps)
    (let ((x (define? (car corps))))
      (cond (x            (integrer-defines (cons x decl) (cdr corps)))
            ((null? decl) (cons 'begin corps))
            (else         (cons 'letrec (cons decl corps))))))
  (list '<lambda> parms (integrer-defines '() corps)))


; Le macro 'let' effectue la transformation
;   (let ((a A)…) B C…)  -->  ((lambda (a…) B C…) A…)

(define-macro (let decl . corps)
  (define (var x) (car x))
  (define (init x) (cadr x))
  (cons (cons 'lambda (cons (map var decl) corps)) (map init decl)))
```

```
; Le macro 'let*' effectue les transformations
;   (let* () A B…)  -->  (let () A B…)
; et
;   (let* ((a A)) B C…)  -->  (let ((a A)) B C…)
; et
;   (let* ((a A) (b B) (c C)…) D E…)  -->  (let ((a A)) (let* ((b B) (c C)…) D E…)

(define-macro (let* decl . corps)
  (if (or (null? decl) (null? (cdr decl)))
    (cons 'let (cons decl corps))
    (list 'let (list (car decl))
      (cons 'let* (cons (list (cdr decl)) corps)))))


; Le macro 'letrec' effectue la transformation
;   (letrec ((a A)…) B C…)  -->
;   (let ((a ?)…) (let ((@ A)…) (set! a @)… (let () B C…)))

(define-macro (letrec decl . corps)
  (define (var x) (car x))
  (define (init x) (cadr x))
  (define (var-temp x) (cons (genvar) x))
  (define (indef x) (list (var x) indefini))
  (define (evaluer x) (list (car x) (init (cdr x))))
  (define (set x) (list 'set! (var (cdr x)) (car x)))
  (let ((decl* (map var-temp decl)))
    (list 'let (map indef decl)
      (cons 'let (cons (map evaluer decl*)
        (append (map set decl*)
                (list (cons 'let (cons '() corps)))))))))


; Le macro 'rec' effectue la transformation
;   (rec a A)  -->  (letrec ((a A)) a)

(define-macro (rec var expr)
  (list 'letrec (list (list var expr)) var))


; Le macro 'named-lambda' effectue la transformation
;   (named-lambda (a b…) A B…)  -->  (rec a (lambda (b…) A B…))

(define-macro (named-lambda patron . corps)
  (list 'rec (car patron) (cons 'lambda (cons (cdr patron) corps))))


; Le macro 'begin' effectue les transformations
;   (begin A)  -->  A
; et
;   (begin A B C…)  -->  (let ((@ A)) (begin B C…))

(define-macro (begin premier . reste)
  (if (null? reste)
    premier
    (list 'let (list (list (genvar) premier)) (cons 'begin reste))))


; Le macro 'and' effectue les transformations
;   (and A)  -->  A
; et
;   (and A B C…)  -->  (if A (and B C…) #!false)

(define-macro (and premier . reste)
  (if (null? reste)
    premier
    (list 'if premier (cons 'and reste) #!false)))


; Le macro 'or' effectue les transformations
;   (or A)  -->  A
; et
;   (or A B C…)  -->  (let ((@ A) (if @ @ (or B C…)))

(define-macro (or premier . reste)
  (if (null? reste)
    premier
    (let ((var-temp (genvar)))
      (list 'let (list (list var-temp premier))
        (list 'if var-temp var-temp (cons 'or reste))))))
```

```
; Le macro 'cond' effectue les transformations
;   (cond)  -->  ?
; et
;   (cond (A) (B C…)…)  -->  (or A (cond (B C…)…))
; et
;   (cond (A B C…) (D E…)…)  -->  (if A (begin B C…) (cond (D E…)…))

(define-macro (cond . clauses)
  (if (null? clauses)
    indefini
    (let ((guarde (if (eq? (caar clauses) 'else) #!true (caar clauses)))
          (consequents (cdar clauses)))
      (if (null? consequents)
        (list 'or guarde (cons 'cond (cdr clauses)))
        (list 'if guarde (cons 'begin consequents) (cons 'cond (cdr clauses)))))))


; Le macro 'case' effectue la transformation
;   (case A ((x…) B C…)…)  -->  (let ((@ A)) (cond ((memv @ '(x…)) B C…)…))

(define-macro (case selecteur . clauses)
  (let ((var-temp (genvar)))
    (define (clause x)
      (cons (list 'memv var-temp (car x)) (cdr x)))
    (list 'let (list (list var-temp selecteur))
      (cons 'cond (map clause clauses)))))


; Le macro 'do' effectue la transformation
;   (do ((a A B)…) (C D…) E F…)  -->
;   ((named-lambda (@ a…)
;      (if C (begin ? D…) (begin E F… (@ a/B…))))
;    A…)

(define-macro (do vars sortie . corps)
  (define (var x) (car x))
  (define (init x) (if (null? (cdr x)) indefini (cadr x)))
  (define (pas x) (if (or (null? (cdr x)) (null? (cddr x))) (car x) (caddr x)))
  (define (test x) (car x))
  (define (exprs x) (cdr x))
  (let ((var-temp (genvar)))
    (cons (list 'named-lambda (cons var-temp (map var vars))
            (list 'if (test sortie)
              (cons 'begin (cons indefini (exprs sortie)))
              (cons 'begin (append corps (list (cons var-temp (map pas vars)))))))
          (map init vars))))


; Le macro 'define' effectue les transformations
;   (define (x a…) A B…)  -->  (define x (lambda (a…) A B…))
; et
;   (define a A)  -->  (set! a A)

(define-macro (define patron . corps)
  (cons 'set! (definition patron corps)))


; La fonction 'define?' prend une donnee representant une expression et retourne
;
; - #!false si la donnee ne represente pas une definition,
; - la liste '(a <forme>)' si la donnee suit la syntaxe d'une definition:
;
;           <definition> ::= (define <patron> A B…)
;           <patron>     ::= a  |  (<patron> b…)
;
; la liste retournee est telle que '(define a <forme>)' est equivalente a la
; definition passee a 'define?'

(define (define? x)
  (if (and (pair? x) (eq? (car x) 'define))
    (definition (cadr x) (cddr x))
    #!false))

(define (definition patron corps)
  (define (def patron expr)
    (if (symbol? patron)
      (list patron expr)
      (def (car patron) (list 'lambda (cdr patron) expr))))
  (if (symbol? patron)
    (list patron (car corps))
    (def (car patron) (cons 'lambda (cons (cdr patron) corps)))))

(define indefini #!false)    ; valeur utilisee a tout endroit ou le resultat est indefini
(define (genvar) (gensym))   ; creation d'une variable temporaire
```

```
; --- expansion des macros ---


(define (expansion expr)
  (cond ((symbol? expr)
         expr)
        ((not (pair? expr))
         (list 'quote expr))
        ((eq? (car expr) '<quote>)
         (list 'quote (cadr expr)))
        ((eq? (car expr) '<set!>)
         (list 'set! (cadr expr)
                     (expansion (caddr expr))))
        ((eq? (car expr) '<if>)
         (let ((pre (expansion (cadr expr)))
               (con (expansion (caddr expr)))
               (alt (expansion (cadddr expr))))
           (if (and (pair? pre) (eq? (car pre) 'quote)) ; predicat = constante?
             (if (cadr pre) con alt) ; generer seulement une des deux branches
             (list 'if pre con alt))))
        ((eq? (car expr) '<lambda>)
         (list 'lambda (cadr expr)
                       (expansion (caddr expr))))
        (else
         (let ((fn-transf (and (symbol? (car expr)) (get (car expr) 'macro))))
           (if fn-transf
             (expansion (apply fn-transf (cdr expr)))
             (let ((vals (map expansion expr)))
               (if (and (pair? (car vals)) (eq? (caar vals) 'lambda) (null? (cdr vals)))
                                ; lambda-expression a 0 parametres en position fonctionnelle?
                   (caddar vals) ; remplacer l'application par le corps de la lambda-expression
                   vals)))))))
```

```
;_____________________________________________________________________________________________
;
;                              Traduction de lambda-expressions
;
;_____________________________________________________________________________________________


(define (lambda->epsilon+closure expr)
  (pass2 (pass1 expr '())))


; --- Premiere passe ---
;
; Construire une structure qui represente l'expression originale et la decorer avec les
; informations relatives au type d'acces aux variables

(define (pass1 expr env)
  (cond ((symbol? expr)           (pass1-ref expr env))
        ((not (pair? expr))       (pass1-cst expr env))
        ((eq? (car expr) 'quote)  (pass1-cst (cadr expr) env))
        ((eq? (car expr) 'set!)   (pass1-set (cadr expr) (caddr expr) env))
        ((eq? (car expr) 'if)     (pass1-tst (cadr expr) (caddr expr) (cadddr expr) env))
        ((eq? (car expr) 'lambda) (pass1-def (cadr expr) (caddr expr) env))
        (else                     (pass1-app expr env))))

(define (pass1-cst val env)
  (list 'cst val))

(define (pass1-ref var env)
  (list 'ref (acceder var env)))

(define (pass1-set var val env)
  (let ((v (acceder var env)))
    (affectee v)
    (list 'set v (pass1 val env))))

(define (pass1-tst pre con alt env)
  (list 'tst (pass1 pre env) (pass1 con env) (pass1 alt env)))

(define (pass1-def patron corps env)
  (let ((env* (list 'def env (parms patron) '() patron '())))
    (set-car! (cdddddr env*) (pass1 corps env*))
    env*))

(define (parms l)
  (cond ((null? l) #!null)
        ((pair? l) (cons (parametre (car l)) (parms (cdr l))))
        (else      (error "parametre reste non implante"))))

(define (pass1-app vals env)
  (cons 'app (map (lambda (x) (pass1 x env)) vals)))


; --- Deuxieme passe ---
;
; Traduire la structure obtenue a la premiere passe en l'expression Scheme correspondante
; en tenant compte du type des variables (i.e. globale/locale/fermee & mutable ou non)
; et de la liste des variables fermees des lambda-expressions

(define (pass2 struct)
  (cond ((eq? (car struct) 'cst) (pass2-cst (cadr struct)))
        ((eq? (car struct) 'ref) (pass2-ref (cadr struct)))
        ((eq? (car struct) 'set) (pass2-set (cadr struct) (caddr struct)))
        ((eq? (car struct) 'tst) (pass2-tst (cadr struct) (caddr struct) (cadddr struct)))
        ((eq? (car struct) 'def) (pass2-def (caddr struct) (cadddr struct) (caddddr struct)
                                                                           (cadddddr struct)))
        ((eq? (car struct) 'app) (pass2-app (cdr struct)))
        (else (error "structure invalide"))))

(define (pass2-cst val)
  (quote1 val))

(define (pass2-ref var)
  (if (mutable? var)
    (list 'fetch (nom var))
    (nom var)))

(define (pass2-set var val)
  (if (mutable? var)
    (list 'store (nom var) (pass2 val))
    (list 'set! (nom var) (pass2 val))))

(define (pass2-tst pre con alt)
  (list 'if (pass2 pre) (pass2 con) (pass2 alt)))
```

```
(define (pass2-def parms fermees patron corps)
  (let ((parametres (map nom parms))
        (boites (ajouter-boites parms)))
    (let ((corps* (if (equal? parametres boites) (pass2 corps)
                      (cons (list 'epsilon parametres (pass2 corps)) boites))))
      (if (null? fermees)
        (list 'epsilon patron corps*)
        (cons 'closure
          (append fermees (list (list 'epsilon (append parametres fermees) corps*))))))))

(define (ajouter-boites parms)
  (if (null? parms) #!null
    (cons (if (mutable? (car parms))
            (list 'box (nom (car parms)))
            (nom (car parms)))
          (ajouter-boites (cdr parms)))))

(define (pass2-app vals)
  (map pass2 vals))

(define (quote1 x) (list 'quote x))   ; retourne la liste: (quote @)  (note: @=argument)

(define (quote2 x) (list 'list (quote1 'quote) x))   ; retourne la liste: (list 'quote @)

(define (cdddddr x) (cdr (cddddr x)))
(define (caddddr x) (car (cddddr x)))
(define (cadddddr x) (cadr (cddddr x)))


; --- Fonctions de manipulation de variables et d'environnements ---

(define (acceder nom env)
  (cond ((null? env)
         (var-globale nom))
        ((assq nom (caddr env)))
        (else
         (let ((var (acceder nom (cadr env))))
           (if (not (globale? var))
             (begin
               (fermee var)
               (if (not (memq nom (cadddr env)))
                 (set-car! (cdddr env) (cons nom (cadddr env))))))
           var))))

(define (parametre nom)   (list nom 'locale  #!false))
(define (var-globale nom) (list nom 'globale #!false))
(define (nom x)           (car x))
(define (globale? x)      (eq? (cadr x) 'globale))
(define (fermee? x)       (eq? (cadr x) 'fermee))
(define (fermee x)        (set-car! (cdr x) 'fermee))
(define (affectee? x)     (caddr x))
(define (affectee x)      (set-car! (cddr x) #!true))
(define (mutable? x)      (and (fermee? x) (affectee? x)))
```

```
;_____________________________________________________________________________________________
;
;             Ecriture d'une structure de donnee Scheme en assembleur MC68000
;
;_____________________________________________________________________________________________


(define (desassembler valeurs fichier)
  (let ((etiquettes (etiquetter valeurs)))
    (define (etiquette val)
      (cond ((integer? val)        val)
            ((eq? val #!null)      "Null")
            ((eq? val #!false)     "False")
            ((eq? val #!true)      "True")
            ((assq val etiquettes) (cdr (assq val etiquettes)))
            (else                  (error "type de donnee non implante"))))
    (define (ecrire x)
      (let ((val (car x)))
        (cond ((and (pair? val) (eq? (car val) '<closure>))
               (display "Fonction " fichier) (display (cdr x) fichier) (newline fichier)
               (do ((p (cdr val) (cdr p)))
                   ((null? (cdr p))
                    (display "        JMP     " fichier)
                    (display (etiquette (car p)) fichier)
                    (newline fichier))
                 (display "        MOVE.L  #" fichier)
                 (display (etiquette (car p)) fichier)
                 (display ",-(SP)" fichier)
                 (newline fichier)))
              ((and (pair? val) (eq? (car val) '<primitive>))
               (display "Fonction " fichier) (display (cdr x) fichier) (newline fichier)
               (for-each (lambda (y) (display y fichier) (newline fichier)) (cdr val)))
              ((pair? val)
               (display "Doublet " fichier) (display (cdr x) fichier)
               (display "," fichier)        (display (etiquette (car val)) fichier)
               (display "," fichier)        (display (etiquette (cdr val)) fichier)
               (newline fichier))
              ((symbol? val)
               (display "Symbole " fichier) (display (cdr x) fichier)
               (display ",'" fichier)       (display val fichier)
               (display "'," fichier)       (display (etiquette (valeur-globale val)) fichier)
               (newline fichier))
              ((string? val)
               (display "Chaine " fichier) (display (cdr x) fichier)
               (display ",'" fichier)      (display val)
               (display "'" fichier)
               (newline fichier))
              (else
               (error "type de donnee non implante")))
        (newline fichier)))
    (display "Debut_Code" fichier) (newline fichier)
    (newline fichier)
    (display "Fonction Depart" fichier) (newline fichier)
    (for-each (lambda (x)
                (display "        JSR     " fichier)
                (display (etiquette x) fichier)
                (newline fichier))
              valeurs)
    (display "        RTS" fichier) (newline fichier)
    (newline fichier)
    (for-each ecrire etiquettes)
    (display "Fin_Code" fichier) (newline fichier)))

(define (etiquetter valeurs)
  (let ((etiquettes '()))
    (define (etiqu val)
      (if (and (memoire val) (not (assq val etiquettes)))
        (begin
          (set! etiquettes (cons (cons val (gensym)) etiquettes))
          (cond ((and (pair? val) (eq? (car val) '<closure>))
                 (for-each etiqu (cdr val)))
                ((and (pair? val) (eq? (car val) '<primitive>)))
                ((pair? val)
                 (etiqu (cdr val))
                 (etiqu (car val)))
                ((symbol? val)
                 (etiqu (valeur-globale val)))))))
    (for-each etiqu valeurs)
    (reverse etiquettes)))
```

```
(define (memoire val) ; indique si la valeur passee occupe un espace en memoire
  (or (string? val)                                  ; une chaine de caracteres
      (symbol? val)                                  ; un symbole
      (and (pair? val) (eq? (car val) '<primitive>)) ; une fonction primitive
      (and (pair? val) (eq? (car val) '<closure>))   ; une fermeture
      (pair? val)                                    ; un doublet
      (vector? val)))                                ; un vecteur
```

Appendice H

Tests du chapitre IV

Nous donnons ici la liste des divers programmes utilisés pour les tests de performance du quatrième chapitre.

fib

```
(define (fib x)
  (if (< x 2) x (+ (fib (- x 1)) (fib (- x 2)))))

(fib 20)
```

tak

```
(define (tak x y z)
  (if (not (< y x)) z
    (tak (tak (-1+ x) y z)
         (tak (-1+ y) z x)
         (tak (-1+ z) x y))))

(tak 18 12 6)
```

trier

```
(define (trier liste)
  (if (null? liste) '() (trier-aux (cdr liste) '() (car liste))))

(define (trier-aux liste reste min)
  (if (null? liste)
    (cons min (trier reste))
    (if (< (car liste) min)
      (trier-aux (cdr liste) (cons min reste) (car liste))
      (trier-aux (cdr liste) (cons (car liste) reste) min))))

(trier '(3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 4
         3 3 8 3 2 7 9 5 0 2 8     2 7 1 8 2 8 1 8 2 8 4
         5 9 0 4 5 2 3 5 3 6 0 2 8 7 4 7 1 3 5 2 6 6 2 4)) ; pi et e
```

*trier

```
(define (*cons the-car the-cdr)
  (lambda (msg) (if (= msg 1) the-car the-cdr)))

(define (*trier liste)
  (if (null? liste) '() (*trier-aux (liste 2) '() (liste 1))))

(define (*trier-aux liste reste min)
  (if (null? liste)
    (*cons min (*trier reste))
    (if (< (liste 1) min)
      (*trier-aux (liste 2) (*cons min reste) (liste 1))
      (*trier-aux (liste 2) (*cons (liste 1) reste) min))))

(*trier (*cons 3 (*cons 1 (*cons 4 (*cons 1 (*cons 5 (*cons 9 (*cons 2
        (*cons 6 (*cons 5 (*cons 3 (*cons 5 (*cons 8 (*cons 9 (*cons 7
        (*cons 9 (*cons 3 (*cons 2 (*cons 3 (*cons 8 (*cons 4 (*cons 6
        (*cons 2 (*cons 6 (*cons 4 (*cons 3 (*cons 3 (*cons 8 (*cons 3
        (*cons 2 (*cons 7 (*cons 9 (*cons 5 (*cons 0 (*cons 2 (*cons 8
        (*cons 2 (*cons 7 (*cons 1 (*cons 8 (*cons 2 (*cons 8 (*cons 1
        (*cons 8 (*cons 2 (*cons 8 (*cons 4 (*cons 5 (*cons 9 (*cons 0
        (*cons 4 (*cons 5 (*cons 2 (*cons 3 (*cons 5 (*cons 3 (*cons 6
        (*cons 0 (*cons 2 (*cons 8 (*cons 7 (*cons 4 (*cons 7 (*cons 1
        (*cons 3 (*cons 5 (*cons 2 (*cons 6 (*cons 6 (*cons 2 (*cons 4
        #!null))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

```
          )))))))))))))) ; pi et e
```

reines

```
(define (reines nb)
  (essayer (nombres nb) '() '()))

(define (nombres n)
  (do ((i n (-1+ i)) (liste '() (cons i liste)))
      ((= i 0) liste)))

(define (essayer x y z)
  (if (null? x)
    (if (null? y) 1 0)
    (+ (if (ok? (car x) 1 z)
         (essayer (ajouter (cdr x) y) '() (cons (car x) z))
         0)
       (essayer (cdr x) (cons (car x) y) z))))

(define (ok? rangee dist placees)
  (if (null? placees)
    #!true
    (and (not (= (car placees) (+ rangee dist)))
         (not (= (car placees) (- rangee dist)))
         (ok? rangee (+ dist 1) (cdr placees)))))

(define (ajouter liste1 liste2)
  (if (null? liste1) liste2
    (cons (car liste1) (ajouter (cdr liste1) liste2))))

(reines 8)
```

tally

```
(define (tally x)
  (lambda (y) (set! x (+ x y)) x))

(define somme (tally 0))

(do ((i 1 (+ i 1)))
    ((= i 1000))
  (somme i))

(somme 0)
```

Appendice I

Exemple de sortie du compilateur

Nous présentons ici, un exemple de sortie produite par le compilateur du chapitre IV. Celle-ci provient de la compilation du programme suivant, contenant deux expressions:

```
(define (adder x) (lambda (y) (+ x y)))
((adder 2) 3)
```

Le programme assembleur produit, avec des commentaires ajoutés par l'auteur, est le suivant:

```
Debut_Code

Fonction Depart                     point d'entrée du programme
        JSR     G0000               évaluer (define (adder x) (lambda (y) (+ x y)))
        JSR     G0015               évaluer ((adder 2) 3)
        RTS                         c'est fini (résultat dans le registre A1)

Fonction G0000                      code de (set! adder (epsilon (x) (closure x…
        MOVE.L  #G0001,-(SP)        empiler le pointeur vers le symbole adder
        MOVE.L  #G0002,-(SP)        empiler le pointeur vers le code de (epsilon (x)…
        JMP     G0014               effectuer l'affectation

Symbole G0001,'adder',Null          le symbole adder (valeur initiale = #!null)

Fonction G0002                      code de (epsilon (x) (closure x…
        MOVE.L  #G0003,-(SP)        empiler le pointeur vers le code de (closure x…
        JMP     G0013               faire comme si c'était une constante

Fonction G0003                      code de (closure x (epsilon (y x) (+ x y)))
        MOVE.L  #4,-(SP)            1 paramètre, donc il faudra désallouer 4 octets de la pile
        MOVE.L  #G0004,-(SP)        empiler le pointeur vers le symbole closure
        MOVE.L  #G0006,-(SP)        empiler le pointeur vers le code de x
        MOVE.L  #G0007,-(SP)        empiler le pointeur vers le code de (epsilon (y x)…
        JMP     G0012               effectuer une application à 2 arguments

Symbole G0004,'closure',G0005       le symbole closure (valeur initiale = Fonction G0005)

Fonction G0005                      code de la fonction primitive closure
        MOVE.L  (SP)+,-(A0)
        MOVE.W  #$4EF9,-(A0)
        SUBQ.W  #2,D3
        BMI.S   @2
@1      MOVE.L  (SP)+,-(A0)
        MOVE.W  #$2F3C,-(A0)
        DBRA    D3,@1
@2      MOVE.L  A0,A1
        RTS

Fonction G0006                      code pour les deux références à x
        MOVE.L  16(SP),A1           obtenir la valeur qui est à 16 octets du dessus de la pile
        RTS

Fonction G0007                      code de (epsilon (y x) (+ x y))
        MOVE.L  #G0008,-(SP)        empiler le pointeur vers le code de (+ x y)
        JMP     G0013               faire comme si c'était une constante

Fonction G0008                      code de (+ x y)
        MOVE.L  #8,-(SP)            2 paramètres, donc il faudra désallouer 8 octets de la pile
        MOVE.L  #G0009,-(SP)        empiler le pointeur vers le symbole +
        MOVE.L  #G0006,-(SP)        empiler le pointeur vers le code de x
        MOVE.L  #G0011,-(SP)        empiler le pointeur vers le code de y
        JMP     G0012               effectuer une application à 2 arguments

Symbole G0009,'+',G0010             le symbole + (valeur initiale = Fonction G0010)

Fonction G0010                      code de la fonction primitive +
        MOVE.L  (SP)+,A1
        ADD.L   (SP)+,A1
        RTS
```

```
Fonction G0011                  code pour la référence à y
        MOVE.L  20(SP),A1       obtenir la valeur qui est à 20 octets du dessus de la pile
        RTS

Fonction G0012                  corps des fermetures pour les applications à 2 arguments
        MOVE.L  (SP)+,A2        dont la position fonctionnelle est une variable globale
        JSR     (A2)
        MOVE.L  (SP),A2
        MOVE.L  A1,(SP)
        JSR     (A2)
        MOVE.L  (SP)+,A3
        MOVE.L  (SP)+,A2
        MOVE.L  (A2),A2
        ADD.L   (SP)+,SP
        MOVE.L  A1,-(SP)
        MOVE.L  A3,-(SP)
        MOVEQ.L #2,D3
        JMP     (A2)

Fonction G0013                  corps des fermetures pour les évaluations de constantes
        MOVE.L  (SP)+,A1
        RTS

Fonction G0014                  corps des fermetures pour les affectations globales
        MOVE.L  (SP)+,A1
        JSR     (A1)
        MOVE.L  (SP)+,A2
        MOVE.L  A1,(A2)
        RTS

Fonction G0015                  code de ((adder 2) 3)
        MOVE.L  #G0016,-(SP)    empiler le pointeur vers le code de (adder 2)
        MOVE.L  #G0019,-(SP)    empiler le pointeur vers le code de 3
        JMP     G0020           effectuer une application à 1 argument

Fonction G0016                  code de (adder 2)
        MOVE.L  #G0001,-(SP)    empiler le pointeur vers le symbole adder
        MOVE.L  #G0017,-(SP)    empiler le pointeur vers le code de 2
        JMP     G0018           effectuer une application à 1 argument

Fonction G0017                  code pour la constante 2
        MOVE.L  #2,A1
        RTS

Fonction G0018                  corps des fermetures pour les applications à 1 argument
        MOVE.L  (SP)+,A2        dont la position fonctionnelle est une variable globale
        JSR     (A2)
        MOVE.L  (SP),A2
        MOVE.L  A1,(SP)
        MOVE.L  (A2),A2
        MOVEQ.L #1,D3
        JMP     (A2)

Fonction G0019                  code pour la constante 3
        MOVE.L  #3,-(SP)        empiler la valeur 3
        JMP     G0013           évaluer la constante

Fonction G0020                  corps des fermetures pour les applications à 1 argument
        MOVE.L  (SP)+,A2
        JSR     (A2)
        MOVE.L  (SP),A2
        MOVE.L  A1,(SP)
        JSR     (A2)
        MOVEQ.L #1,D3
        JMP     (A1)

Fin_Code
```

Références

[Abel85a] Abelson, H., Sussman, G. J., Sussman, J., *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1985.

[Abel85b] Abelson, H., et al, *The Revised Revised Report on Scheme or an UnCommon Lisp*. MIT Artificial Intelligence Memo 848, Cambridge, Massachusetts, August 1985.

[Aho77] Aho, A. V., Ullman, J. D., *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts, 1977.

[Alle75] Allen, F. E., *Bibliography on program optimization*. IBM Research Report RC-5767, Technical Journal Watson Research Center, Yorktown Heights, New York, 1975.

[Alle78] Allen, J., *Anatomy of Lisp*. McGraw-Hill, New York, New York, 1978.

[Atki85] Atkinson, M. P., Morrison, R., Procedures as Persistent Data Objects. In *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, pages 539-559, October 1985.

[Baer80] Baer, J.-L., *Computer Systems Architecture*. Computer Science Press, Rockville, Maryland, 1980.

[Bare81] Barendregt, H. P., *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1981.

[Betz85] Betz, D., An XLISP Tutorial. In *Byte*, vol. 10, no. 3, page 221, Peterborough, New Hampshire, March 1985.

[Broo84] Brooks, R. A., Gabriel, R. P., A Critique of Common Lisp. In *Conference record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[Chur41] Church, A., *The Calculi of Lambda-Conversion*. Annals of Mathematics Studies Number 6, Princeton University Press, Princeton, New Jersey, 1941.

[Dahl82] Dahl, O.-J., Myhrhaug, B., Nygaard, K., *SIMULA 67 Common Base Language*. Norwegian Computing Center Report 725, December 1982.

[Dijk67] Dijkstra, E. W., Recursive Programming. In *Programming Systems and Languages*, McGraw-Hill, New York, New York, 1967.

[Expe84] ExperTelligence, *ExperLisp Reference Manual*. Santa Barbara, California, 1984.

[Fess83] Fessenden, C., Clinger, W., Friedman, D. P., Haynes, C., *Scheme 311 Version 4 Reference Manual*. Indiana University Computer Science Technical Report 137, February 1983.

[Fode82] Foderaro, J. K., Sklower, K. L., *The FRANZ LISP Manual*. University of California, Berkeley, California, April 1982.

[Forg81] Forgy, C. L., *The OPS5 User's Manual*. Technical Report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1981

[Gabr82] Gabriel, R. P., Masinter, L. M., Performance of Lisp Systems. In *Conference record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 123-142, Pittsburgh, Pennsylvania, August 1982.

[Gold83] Goldberg, A., Robson, D., *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[Gord79] Gordon, M., Milner, R., Wadsworth, C., *Edinburgh LCF*. Lecture Notes in Computer Science, vol. 78, Springer-Verlag, New York, New York, 1979.

[Goto74] Goto, E., *Monocopy and Associative Algorithms in an Extended Lisp*. University of Tokyo, Japan, May 1974.

[Hara78] Haraldsson, A., A Partial Evaluator, and Its Use for Compiling Iterative Statements in Lisp. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978.

[Hech77] Hecht, M. S., *Data Flow Analysis of Computer Programs*. American Elsevier, New York, New York, 1977.

[Jone79] Jones, N. D., Muchnick, S. S., Flow Analysis and Optimisation of LISP-like Structures. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1979.

[Klin81] Klint, P., Interpretation Techniques. In *Software-Practice and Experience*, vol. 11, pages 963-973, 1981.

[Loel81] Loeliger, R. G., *Threaded Interpretive Languages*. Byte Books, Peterborough, New Hampshire, 1981.

[McCa60] McCarthy, J., Recursive functions of symbolic expressions and their computation by machine. In *Communications of the ACM*, vol. 3, no. 4, pages 184-195, 1960.

[McDe73] McDermott, D., Sussman, G. J., *The CONNIVER Reference Manual*. MIT Artificial Intelligence Memo 259A, Cambridge, Massachusetts, 1973.

[McDe80] McDermott, D., An efficient Environment Allocation Scheme in an Interpreter for a Lexically-scoped Lisp. In *Conference record of the 1980 ACM Symposium on Lisp and Functional Programming*, pages 154-162, Stanford, California, August 1980.

[MITS84] *MIT Scheme Manual*. Seventh Edition, Cambridge, Massachusetts, September 1984.

[Pitm83] Pitman, K., *The revised MacLisp Manual*. MIT Computer Science Technical report 295, Cambridge, Massachusetts, 1983.

[Rand64] Randell, B., Russell, L. J., *Algol 60 implementation*. Academic Press, New York, New York, 1964.

[Rees82] Rees, J. A., Adams, N. I., 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, Pittsburgh, Pennsylvania, August 1982.

[Rees84] Rees, J. A., Adams, N. I., Meehan, J. R., *The T Manual*. Computer Science Department, Yale University, New Haven, Connecticut, January 1984.

[Rich83] Rich, E., *Artificial Intelligence*. McGraw-Hill, New York, New York, 1983.

[Sema85] Semantic Microsystems, *MacScheme Reference Manual*. Sausalito, California, 1985.

[Stal77] Stallman, R. M., Sussman, G. J., Forward Reasoning and Dependency-directed Backtracking in a System for Computer-aided Circuit Analysis. In *Artificial Intelligence*, vol. 9, no. 2, pages 135-196, October 1977.

[Stee76] Steele, G. L., *Lambda: the ultimate declarative*. MIT Artificial Intelligence Memo 379, Cambridge, Massachusetts, November 1976.

[Stee78a] Steele, G. L., Sussman, G. J., *The Revised Report on Scheme: A Dialect of Lisp*. MIT Artificial Intelligence Memo 452, Cambridge, Massachusetts, January 1978.

[Stee78b] Steele, G. L., *Rabbit: a compiler for Scheme*. MIT Artificial Intelligence Memo 474, Cambridge, Massachusetts, May 1978.

[Stee84] Steele, G. L., *Common Lisp: the Language*. Digital Press, 1984.

[Stoy84] Stoyan, H., Early LISP History. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[Suss71] Sussman, G. J., Winograd, T., Charniak, E., *MICRO-PLANNER Reference Manual*. MIT Artificial Intelligence Memo 203A, Cambridge, Massachusetts, 1971.

[Suss75] Sussman, G. J., Steele, G. L., *Scheme: An Interpreter for extended Lambda Calculus*. MIT Artificial Intelligence Memo 349, Cambridge, Massachusetts, December 1975.

[Tera75] Terashima, M., *Algorithms Used in an Implementation of HLISP*. Information Sciences Laboratory Technical Report 75-03, University of Tokyo, Japan, January 1975.

[Wand80] Wand, M., Continuation-Based Multiprocessing. In *Conference record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 19-28, Stanford, California, August 1980.

[Wegn76] Wegner, P., Lambda Calculus. In *Encyclopedia of Computer Science*, pages 752-755, Van Nostrand Reinhold Co., New York, New York, 1976.