Université de Montréal

Un système de programmation Scheme pour micro-contrôleur

par

Danny Dubé

Département d'informatique et
de recherche opérationnelle

Faculté des arts et sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M. Sc.)
en informatique

Avril 1996

Page d'identification du jury

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Un système de programmation Scheme
pour micro-contrôleur

présenté par:

Danny Dubé

a été évalué par un jury composé des personnes suivantes:

Yoshua Bengio
(président du jury)

François Major

Marc Feeley
(directeur de maîtrise)

Mémoire accepté le: 1er août 1996

Sommaire

Notre travail vise à étudier les différents aspects reliés à un système de programmation Scheme compact. Le but principal consiste à montrer qu'on peut faire une implantation suffisamment compacte pour pouvoir fonctionner sur un micro-contrôleur. Les buts secondaires consistent à trouver ou développer un gestionnaire mémoire automatique (GC) temps réel implantable de façon compacte, ainsi que de choisir des techniques portables.

Les différents aspects que nous étudions sont: la réprésentation des données, principalement en ce qui concerne le typage, les symboles, les continuations et les environnements lexicaux; plusieurs techniques de GC temps réel et la recherche d'une technique originale; le développement d'un modèle d'exécution très compact utilisant la compilation.

A priori, il ne semble pas évident d'atteindre l'objectif principal. Plusieurs systèmes Scheme compacts existent déjà et aucun de ceux-ci ne peut se plier aux contraintes sévères imposées par un micro-contrôleur. La contrainte majeure imposée par un micro-contrôleur est la petitesse de l'espace mémoire.

Table des matières

Table des figures

Chapitre 1

Introduction

De plus en plus d'appareils utilisent des micro-contrôleurs pour régler leurs différentes fonctions. Il suffit de penser à divers appareils électro-ménagers, électroniques ou à la robotique. Plus souvent qu'autrement, les micro-contrôleurs sont programmés en assembleur. La programmation peut aussi se faire dans des langages plus évolués, par exemple C ou un autre langage de cet acabit. Ces langages sont plus intéressants puisque le processus de programmation y est plus rapide et plus fiable.

Notre recherche consiste à montrer qu'un langage aussi évolué que Scheme peut aussi être utilisé pour la programmation de ces micro-contrôleurs.

1.1 Objectifs

Ce travail consiste à montrer la possibilité de réaliser une implantation d'un système de programmation Scheme pour micro-contrôleur. Nous ne cherchons pas nécessairement à réaliser un interprète capable d'interagir avec un usager. On peut donc effectuer une compilation des programmes afin de générer l'image mémoire qui pourra être transférée sur le micro-contrôleur.

Un objectif secondaire consiste à doter le système Scheme d'un GC (Garbage Collector) temps réel. En effet, un programme Scheme consomme de la mémoire mais ne la retourne jamais explicitement. Il faut donc inclure un système de récupération automatique de la mémoire. Les GC bloquants ont la facheuse tendance à provoquer des pauses momentanées dans le fil d'exécution du programme. Ce seul défaut peut rendre un système de programmation Scheme beaucoup moins attrayant pour une application. Un GC temps réel évite le problème des pauses.

Enfin, les techniques que nous développons ou pour lesquelles nous optons doivent s'appliquer dans un contexte général. Nous ne sommes pas intéressés par des techniques qui sont dépendantes d'un micro-contrôleur particulier ou d'une caractéristique technique du contexte d'utilisation.

1.2 Motivation

Il y a plusieurs raisons de s'intéresser à un système Scheme compact. Pour autant que nous sachions, il n'a a pas de telle implantation, ou du moins, aucune qui ne soit destinée à fonctionner dans un environnement aussi restrictif qu'un micro-contrôleur. Plusieurs implantations sont relativement compactes, comme Scm ([IntScm]), Siod ([IntSiod]) et surtout Minischeme ([IntMini]). Mais elles sont soit trop volumineuses pour un micro-contrôleur, soit incomplètes. La section 7.2.1 donne des mesures concrètes. Enfin, il existe une machine conçue expressément pour le langage Lisp. Une implantation matérielle n'est toutefois pas très générale et est peu économique.

Comme nous le mentionnons brièvement au début de l'introduction, il existe déjà des systèmes compacts de programmation pour différents langages. Par exemple, il y a des interprètes Basic et Forth contenus sur une mémoire extrêmement réduite et qui peuvent interagir avec l'usager. Il y a aussi des compilateurs pour différents langages qui peuvent générer du code natif pour plusieurs micro-contrôleurs. Il est manifestement intéressant de pouvoir programmer les micro-contrôleurs autrement qu'en assembleur.

Bien sûr, le code natif produit par un compilateur n'est pas toujours d'aussi bonne qualité que celui écrit par un programmeur assembleur chevronné. De même, un programme qui est interprété ne fonctionne pas aussi vite que s'il était traduit en code natif. Toutefois, le dilemme entre l'utilisation d'outils évolués et la réalisation manuelle ne se pose pas uniquement dans la programmation des micro-contrôleurs. Nous considérons que la programmation d'un micro-contrôleur en Scheme reste un problème intéressant malgré que l'on perde de la performance par rapport à la programmation en assembleur.

Un attrait de Scheme vient du fait que c'est un langage plus évolué que ceux utilisés couramment. Des langages comme Basic, Pascal et C sont *grosso modo* du même niveau d'abstraction. De la même façon qu'il est plus facile de programmer dans ces langages qu'en assembleur, il est encore plus facile de programmer en Scheme. L'avantage de Scheme sur ces langages vient surtout du nombre restreint de concepts (donc de sa simplicité), de la gestion automatique de la mémoire et du traitement des erreurs.

Premièrement, le langage Scheme comporte peu de concepts, mais ceux-ci sont suffisamment puissants pour permettre d'exprimer aisément la plupart des programmes que l'on fait en C par exemple. Une faiblesse de Scheme par rapport à C, toutefois, est la distance créée entre le programme et les opérations de très bas niveau. Par exemple, en Scheme, les accès à des adresses particulières en mémoire ne sont possibles que par l'intermédiaire de fonctions spécialisées.

Deuxièmement, un système Scheme fournit un dispositif automatique de récupération de la mémoire. La gestion de la mémoire est l'un des gros problèmes lors du développement des applications dans les langages de plus bas niveau.

Troisièmement, lorsqu'une erreur se produit en Scheme, elle est détectée et reportée en termes de concepts de Scheme. Il n'y a pas d'opérations illicites qui sont acceptées silencieusement ou

détectée par le système d'exploitation. Cette qualité rend le développement de programmes plus facile.

L'intérêt des langages plus évolués est clair. Plus le langage est évolué, plus il est facile de programmer des tâches complexes. Par exemple, en robotique, le travail à effectuer est souvent complexe.

Scheme est donc intéressant pour le développement de programmes. En outre, il est suffisamment simple pour qu'il soit envisageable de l'implanter de façon compacte. Les concepts (environnements lexicaux, fermetures, récursion de queue en espace constant, continuations) sont peu nombreux et ne semblent pas *a priori* demander une utilisation intensive de mémoire. Ensuite, la librairie standard comporte un nombre réduit de fonctions. Ce sont ces raisons qui nous poussent à croire qu'il est possible d'arriver à faire un système Scheme compact.

Enfin, bien que notre étude vise l'implantation sur micro-contrôleur, le développement de techniques portables rend ces dernières utilisables sous d'autres contextes. Dans les situations où l'économie d'espace prime sur l'efficacité, nos résultats peuvent s'avérer utiles.

1.3 Problématique

Bien que la réalisation d'un système Scheme compact semble possible, on ne peut en être sûr *a priori*. Premièrement, beaucoup de recherches ont déjà été faites sur les différents aspects d'un système Scheme. Mais ces recherches portent principalement sur la recherche d'efficacité en temps. Il faut donc comparer les techniques existantes et éventuellement en chercher de nouvelles pour avoir les techniques les plus économes en espace.

Deuxièmement, comme nous le disons plus haut, il n'y a pas, à notre connaissance, d'implantation assez compacte pour être utilisée sur un micro-contrôleur. Il n'est peut-être pas possible de réaliser une telle implantation.

1.4 Méthodologie

Nous avons développé plusieurs versions d'un système Scheme. Les premières sont des interprètes interactifs. Dans ces versions, nous testons plusieurs techniques de GC et des représentations pour les objets. Les dernières versions sont constituées d'un compilateur vers du code-octets (byte-code) et d'un interprète de code-octets. Notre implantation du compilateur est en Scheme et celle du noyau en C. Le compilateur sert à traduire un programme source en un fichier de code-octets destiné à être lié avec l'interprète. Nous y avons adopté une nouvelle représentation des environnements et plusieurs techniques pour améliorer de nombreux points secondaires.

Les chapitres du mémoire ne suivent pas l'ordre chronologique du développement. Ils regroupent plutôt la discussion par aspects. Le prochain chapitre sert à préciser les objectifs tout

en introduisant les notions qui sont les plus utiles pour la compréhension de ce travail. Les chapitres suivants traitent de la représentation des objets, du GC, du compilateur et de l'interprète de code-octets. Quelques résultats expérimentaux sont présentés avant la conclusion.

Chapitre 2

Description précise de l'objectif

Dans ce chapitre, nous décrivons plus en détail le but que nous voulons atteindre dans notre travail. Premièrement, nous précisons les caractéristiques du type de machine qui nous intéresse. Deuxièmement, nous faisons un survol du langage Scheme en insistant sur les points qui sont particulièrement importants du point de vue de l'implanteur. Enfin, nous expliquons plus formellement ce que nous entendons par GC temps réel.

2.1 Type de machine cible

Le type de machine qui nous intéresse pour ce travail est le micro-contrôleur. Il s'agit d'un ordinateur d'usage général et de faible capacité, tant en mémoire qu'en puissance de calcul. Les micro-contrôleurs sont typiquement très bon marché et peuvent être intégrés à toutes sortes d'appareils pour contrôler leur fonctionnement.

Lorsque l'on réalise une application pour un tel micro-contrôleur, on écrit un programme pour décrire la tâche à effectuer. Typiquement, la programmation se fait en assembleur et le programme est assemblé grâce à un logiciel fourni par le fabriquant. Ensuite, le programme compilé est inscrit dans la mémoire morte (ROM) et reproduit à grande échelle. Lors de la fabrication d'un appareil, le micro-contrôleur et le module ROM contenant le programme adéquat y sont inclus et connectés. Ainsi, un micro-contrôleur tout usage peut être utilisé dans une multitude de tâches différentes, simplement en changeant sa programmation.

Le micro-contrôleur que nous avons adopté pour ce travail est le 68HC11 de Motorola. C'est un micro-contrôleur très répandu qui représente bien les micro-contrôleurs qui existent sur le marché. Il possède un processeur 8 bits qui peut effectuer certaines opérations sur 16 bits. Les adresses sont codées sur 16 bits et l'espace d'adressage est donc limité à 64 Koctets. Il dispose d'un très petit nombre de registres (6), dont un seul est d'usage général. L'horloge réglant le déroulement des cycles de la machine fonctionne à 2 MHz. En moyenne, les instructions

requièrent environ 4 cycles pour être exécutées. Ce genre de micro-contrôleurs n'est certes pas fait pour effectuer du calcul intensif. On ne peut raisonnablement espérer que les programmes aient une exécution très rapide sur une telle machine.

La limite de 64 Koctets sur l'espace d'adressage est la contrainte principale imposée à une implantation de Scheme et elle est liée à l'utilisation de la plupart des micro-contrôleurs. Toutes les parties d'un interprète Scheme doivent pouvoir être contenues dans cet espace: noyau de l'interprète (interprète de code-octets), librairie du langage, programme à exécuter et données créées à l'exécution.

2.2 Survol de Scheme

Le langage Scheme est un des dérivés de Lisp. Il s'agit donc d'un langage fonctionnel impur à syntaxe parenthésée qui est bien adapté pour le calcul symbolique. Contrairement à certains dialectes de Lisp, il utilise la portée lexicale. Le passage des paramètres se fait toujours par valeur. Il est typé dynamiquement (ou faiblement typé). Même s'il est souvent utilisé comme un langage fonctionnel, Scheme permet les effets de bords sur les variables et dans les structures de données. La récupération de l'espace mémoire est faite automatiquement. C'est un langage conçu avec une philosophie minimaliste.

Scheme possède les types suivants: les booléens, les nombres, les caractères, les paires, les vecteurs, les chaînes de caractères, les symboles et les fonctions.

Les paires sont les objets à deux champs qui sont utilisés dans tous les dialectes de Lisp pour constituer les listes. Les vecteurs sont similaires aux tableaux à une dimension que l'on retrouve dans les langages plus traditionnels. Les listes et les vecteurs ne sont pas resteints à ne contenir que des objets d'un seul type. Une implantation "normale" permet d'effectuer des accès aléatoires dans un vecteur beaucoup plus rapidement que dans une liste.

Les symboles sont des objets qui sont entièrement décrits par leur nom. Deux symboles nommés exactement de la même façon sont en fait le même symbole. Il est possible d'obtenir le nom d'un symbole sous la forme d'une chaîne de caractères et de faire l'opération inverse à partir d'une chaîne.

Les fonctions sont reconnues comme objets de première classe et jouent un rôle majeur en Scheme. Par objets de première classe, on signifie que les fonctions peuvent être créées, stockées dans des structures de données, passées en paramètre et retournées par d'autres fonctions. La majorité des fonctions manipulées dans un programme sont soit des fonctions primitives de la librairie, soit des fermetures. Les fermetures sont créées par les lambda-expressions, comme en lambda-calcul. Elles ne sont toutefois pas limitées à un seul paramètre. Comme en lambda-calcul, le fait d'exécuter une lambda-expression crée une fermeture, c'est-à-dire que l'environnement lexical des variables "en portée" est conservé avec le corps de la lambda-expression. Enfin, certaines fonctions sont des continuations qui ont été réifiées. Nous discutons des continuations

dans une section à part.

La librairie de Scheme est principalement constituée des fonctions de manipulation des objets des différents types. On y retrouve aussi quelques fonctions d'ordre supérieur d'usage courant. La librairie est relativement réduite et c'est une des raisons qui laisse croire que Scheme peut être implanté de façon compacte.

Scheme comprend quelques formes syntaxiques. Quelques-unes ne sont pas remplaçables, comme les lambda-expressions et les affectations, mais la plupart sont du "sucre syntaxique". Le nombre restreint de formes de base est une autre contribution à la simplicité du langage.

Enfin, le standard ([R^{4}RS]) requiert que les appels en position terminale s'effectuent sans consommation d'espace. Nous discutons de ce point particulier dans une section séparée.

2.3 Scheme sur micro-contrôleur

Certains aspects de Scheme perdent leur intérêt ou leur sens lorsque l'on s'intéresse à la programmation des micro-contrôleurs.

Premièrement, nous ne désirons pas implanter toute la tour des nombres. Les rationnels et les complexes ne sont pas d'usage courant et d'ailleurs ils manquent à plusieurs implantations de Scheme. Par contre, les nombres réels en point flottant sont d'usage courant en programmation. Mais pas nécessairement en programmation de micro-contrôleurs. Un micro-contrôleur typique comme le 68HC11 ne traite pas les nombres en point flottant. Nous préférons éviter d'ajouter des librairies pour simuler des nombres en point flottant. Il en va de même pour les entiers de taille illimitée. Notre système Scheme ne comporte donc que des entiers de taille bornée.

Deuxièmement, les fonctions d'entrées/sorties textuelles perdent leur intérêt dans le contexte d'un micro-contrôleur. L'interprète Scheme n'est pas destiné à fonctionner interactivement, encore moins à être branché à un serveur de disques. En fonction de l'utilisation prévue du micro-contrôleur, les besoins pour des fonctions d'entrées/sorties peuvent changer. Nous ne pouvons pas raisonnablement fixer un modèle particulier de communication avec l'extérieur, donc nous ne nous sommes pas attardés sur les questions d'entrées/sorties.

2.4 Continuations

La présence des continuations comme objets de première classe est un trait de Scheme qui se retrouve dans peu d'autres langages. Même si une grande partie des programmes Scheme n'utilisent pas explicitement les continuations et qu'une seule fonction de la librairie les concernent, les continuations n'en demeurent pas moins un outil puissant. En outre, elles imposent beaucoup de contraintes à l'implanteur d'un système Scheme.

Les continuations sont spécialement utiles sur un micro-contrôleur. En effet, les micro-contrôleurs sont souvent utilisés pour traiter plusieurs signaux et contrôler de nombreux dispositifs simultanément. Or, la programmation concurrente est très utile pour effectuer un tel traitement. On voit justement à la section 2.4.3 comment les continuations permettent de faire de la programmation concurrente. Une telle application constitue, selon nous, une justification suffisante pour permettre l'existence des continuations comme objets de première classe dans notre interprète.

Cette section sert à illustrer ce que sont les continuations et à montrer quels impacts elles ont sur la structure d'un interprète. Premièrement, une définition des continuations est donnée. Deuxièmement, la fonction `call/cc` est introduite. Troisièmement, l'utilité des continuations est illustrée par un exemple. Enfin, nous discutons brièvement de l'impact des continuations sur l'architecture d'un interprète.

2.4.1 Définition

Intuitivement, une continuation est un calcul suspendu qui attend un résultat pour se poursuivre. La continuation "courante" est le calcul qui reste à faire lorsque le calcul courant se termine. Une continuation attend après le résultat du calcul courant.

Exemple

Prenons par exemple cette expression:

```
(set! x (car p))
```

L'évaluation de cette expression se décompose de la façon suivante (sans décomposer l'appel): appel de la fonction `car` sur la paire contenue dans `p`, affectation du résultat à la variable `x`. Avant de pouvoir effectuer l'affectation à la variable `x`, il est impératif de connaître le résultat de l'appel. L'affectation est le calcul à effectuer une fois que l'appel a rendu son résultat. L'affectation est donc la continuation de l'appel.

Si on considère que l'expression fait partie d'un programme, l'affectation elle-même a une continuation. C'est l'exécution du reste du programme. Dans ce cas, la continuation de l'appel est l'affectation suivie de l'exécution du reste du programme.

Conversion CPS

Un programme est normalement écrit en style direct, c'est-à-dire que les expressions ont une valeur de retour. Valeur de retour qui peut être passée en argument, liée à une variable, ... Mais, par conversion CPS, on peut transformer un programme écrit en style direct en un programme écrit en style de passage par continuations (Continuation Passing Style). En style

CPS, les expressions n'ont pas de valeur de retour: elle ne retournent pas. Elles passent plutôt le résultat de leur évaluation à leur continuation. La continuation courante est toujours accessible car elle est passée explicitement en paramètre.

En style CPS, on représente les continuations par des fonctions à un paramètre. Toutes les fonctions de la librairie et les lambda-expressions prennent un paramètre supplémentaire qui est la continuation. Regardons notre exemple une fois converti en style CPS:

```
(lambda (k)
  (car p (lambda (result)
           (set! x result)
           (k #t))))
```

Notre exemple n'est plus une expression prête à retourner une valeur (le résultat indéfini du `set!`, disons `#t`), mais plutôt une fonction qui reçoit une continuation (`k`).

La première chose qui est faite dans cette fonction est d'effectuer l'appel. La fonction `car` prend maintenant deux paramètres: une paire et une continuation. Une fois que le premier champ de la paire est connu, il est passé en paramètre à la continuation (celle reçue par `car`).

Cette continuation représente bel et bien le reste du calcul. Elle consiste à affecter le résultat à `x` et ensuite à poursuivre l'exécution du reste du programme. Ceci est fait en passant le résultat de l'expression `set!` à la continuation `k`.

Il est possible de convertir tout un programme en style CPS grâce à des règles de transformation systématiques. Ce n'est toutefois pas notre but de présenter une telle technique ici. La transformation qui précède vise seulement à illustrer la nature des continuations.

2.4.2 Accès aux continuations en Scheme

Comme nous le disons plus haut, un programme Scheme ne travaille habituellement pas avec les continuations directement. De toute manière, il est toujours possible de réaliser un programme sans utiliser les continuations. Mais l'accès aux continuations permet de simplifier grandement la programmation de certaines tâches.

L'accès aux continuations se fait grâce à la fonction `call-with-current-continuation`. La plupart des implantations lui donnent aussi le nom de `call/cc`. Cette fonction permet de réifier la continuation courante à un point donné du programme. Il est important d'utiliser le terme "réifier" car un interprète ne crée pas nécessairement toutes les continuations qui entrent en jeu conceptuellement.

La fonction `call/cc` reçoit en paramètre une fonction à un argument. Elle l'invoque en lui passant en paramètre sa propre continuation. La continuation en question est "réifiée" sous la forme d'une continuation semblable à celles que l'on fait apparaître lors d'une conversion CPS. Il s'agit d'une véritable fonction à un paramètre. Toutefois, l'invocation de la continuation n'a

pas le même effet que l'invocation d'une fonction ordinaire. L'invocation d'une continuation ne retourne jamais.

```
(+ (* 2 4)
   (call/cc (lambda (k)
              (if (test 8 16)
                  32
                  (begin (k 32) #f)))))
```

Dans l'exemple ci-haut, supposons que l'évaluation se fasse de gauche à droite: la référence à `+`, le produit, l'appel à `call/cc` et enfin la somme. Dans ce cas, la continuation de l'appel à `call/cc` consiste à effectuer la somme du résultat du produit et du résultat de l'appel à `call/cc`.

L'opération qui effectue la somme est précisément la continuation qui est passée à la fonction via son paramètre `k`. De plus, elle constitue la continuation courante durant l'exécution du corps de la fonction. Donc, que l'exécution du corps se termine normalement ou qu'elle se termine par une invocation explicite de la continuation reçue dans `k`, l'effet est le même. La somme s'effectue sur ce nouveau résultat et sur celui du produit. Dans le premier cas, le résultat est arrivé à la continuation de manière implicite grâce à un retour de valeur. Dans le deuxième, le résultat est transmis explicitement à la continuation par l'invocation de celle-ci. Dans notre exemple, l'appel à `call/cc` retourne 32, quel que soit le résultat de la fonction `test`.

2.4.3 Exemple d'utilisation des continuations

Un exemple classique et simple à comprendre est la création de fonctions d'échappement non-local. C'est une des applications les plus courantes des continuations. Le fait de pouvoir abandonner un calcul lorsque nécessaire permet de simplifier grandement la programmation de certaines applications.

De nombreux langages fournissent un moyen d'effectuer des sorties non-locales. Cet exemple ne permet donc pas d'illustrer le potentiel des continuations. Nous préférons présenter un exemple de programmation concurrente.

Il est important de bien savoir ce nous avons en tête lorsque nous parlons d'exécution concurrente. Il ne s'agit pas de parallélisme. L'exécution du programme se fait sur un seul processeur et il n'y a pas deux calculs faits en même temps. Il s'agit d'avoir plusieurs fils d'exécution dans le programme. A tout moment, un de ces fils d'exécution a le contrôle et effectue du travail. Lorsque c'est jugé approprié, le fil courant laisse temporairement le contrôle aux autres fils d'exécution. Il le retrouve par la suite.

Ce type d'exécution concurrente peut être programmé en Scheme à l'aide des continuations. Le programmeur n'a qu'à insérer dans son programme des appels réguliers à une fonction de changement de fil d'exécution. De plus, il peut faire séparer un fil d'exécution en deux fils distincts et il peut faire terminer un fil d'exécution.

Nous présentons une implantation d'exécution concurrente. Les noms `switch`, `fork` et `terminate` sont ceux des trois fonctions décrites dans le paragraphe précédent, dans l'ordre. Ces trois fonctions ne prennent pas d'argument. Seule la fonction `fork` retourne un résultat significatif, qui est un booléen, et qui permet au programme de distinguer les deux fils d'exécution générés. Voici donc l'implantation de ces fonctions:

```
(define thread-head (cons #f '()))
(define thread-tail thread-head)

(define add-thread
  (lambda (thread)
    (set-car! thread-tail thread)
    (set-cdr! thread-tail (cons #f '()))
    (set! thread-tail (cdr thread-tail))))

(define extract-thread
  (lambda ()
    (if (eq? thread-head thread-tail)
        (begin (display "Erreur: ...") (newline))
        (let ((thread (car thread-head)))
          (set! thread-head (cdr thread-head))
          thread))))

(define switch
  (lambda ()
    (call/cc (lambda (thread)
               (add-thread thread)
               ((extract-thread) #f)))))

(define fork
  (lambda ()
    (call/cc (lambda (thread)
               (add-thread thread)
               #t))))

(define terminate
  (lambda ()
    ((extract-thread) #f)))
```

Le code demande quelques explications. La liste des fils d'exécution est une file d'attente et les deux premières fonctions servent à ajouter à la fin de et à extraire du début de la file. Il est à noter que le fil en cours d'exécution ne se trouve pas dans la file d'attente. Ainsi, la file est

vide s'il n'y a qu'un fil d'exécution.

Comme nous le disons plus haut, la fonction `switch` sert à passer le contrôle aux autres fils d'exécution. Elle capture la continuation courante, la sauve dans la file, extrait une continuation et poursuit l'exécution avec cette dernière. On peut imaginer que chaque continuation représente l'état du calcul d'un fil d'exécution bloqué.

Lorsqu'un fil d'exécution tombe sur une expression `(switch)`, son état est sauvé sous la forme d'une continuation dans la file d'attente. Eventuellement, cette continuation est extraite de la file et est invoquée. Ceci a pour effet de ressortir de l'appel à `switch`. Entre le moment où le fil d'exécution a appelé la fonction `switch` et celui où l'appel se termine, plusieurs fils d'exécution se sont potentiellement succédés. S'il n'y a pas d'autres fils d'exécution toutefois, l'appel à `switch` se termine aussitôt, car la continuation qui vient tout juste d'être stockée est extraite immédiatement.

La fonction `fork` est construite sensiblement de la même façon. Mais celle-ci sert à dupliquer un fil d'exécution. Lorsqu'elle est invoquée, elle capture la continuation et la sauve dans la file d'attente, mais elle renvoit immédiatement le contrôle à l'appelant. Un appel à `fork` ressort donc presque aussitôt. Toutefois, l'état du calcul a été sauvé au cours de l'appel. Celui-ci constitue donc la copie du fil d'exécution qui se poursuit. Ainsi, un appel à `fork` retourne deux fois!

Il est important de constater que la fonction `fork` retourne correctement les valeurs `#t` et `#f` servant à différencier les deux fils. Nous pouvons voir que le fil "original", celui qui est rétabli aussitôt après l'appel à `fork`, continue avec le résultat `#t`. L'autre fil, quant à lui, sera nécessairement rétabli par la fonction `switch` ou `terminate`, donc il continuera avec le résultat `#f`. Il est important de ne pas confondre les choses ici. Ce deuxième fil, même s'il sera rétabli par `switch` ou `terminate`, est décrit par une continuation qui a été capturée lors d'un appel à `fork` et poursuivra donc son exécution immédiatement après le site d'appel à `fork`.

Finalement, il reste à décrire la fonction `terminate`. Celle-ci sert à faire disparaître un fil d'exécution, chose qui ne peut être faite avec les fonctions `switch` et `fork`. Un appel à `terminate` ne fait qu'extraire une continuation et lui passer le contrôle. La continuation courante, c'est-à-dire l'état du calcul du fil, n'est pas sauvée. Un appel à la fonction `terminate` ne ressort donc pas. Il y a un cas spécial avec cette fonction et c'est le cas où le programme essaie de terminer son unique fil d'exécution. Ce cas est détecté dans la fonction d'extraction de la file d'attente. Dans l'exemple, nous ne faisons qu'écrire un message lorsque cela se produit.

Ces fonctions que nous présentons sont réduites à leur plus simple expression. Le programmeur d'une application pourrait vouloir plus de facilités rattachées à la gestion des fils d'exécution. Par exemple, donner des numéros distincts à tous les fils, laisser un fil dormir jusqu'à ce qu'un signal indique de le remettre dans la file d'attente, etc. Toutefois, nous voulons montrer la possibilité de faire de l'exécution concurrente le plus simplement possible.

Il est possible de bâtir une application qui gère de multiples fils d'exécution sans l'aide des continuations explicites. Mais cette tâche est terriblement difficile, surtout lorsqu'on permet des programmes récursifs.

2.4.4 Impact sur l'interprète

Connaissant les effets des continuations, il est clair qu'elles imposent des contraintes assez fortes sur un interprète. Une pile d'exécution contiguë comme on en retrouve dans les langages plus conventionnels comme le C ne suffit pas à réaliser les continuations de Scheme. En effet, en C, lorsqu'un calcul s'est terminé, on ne peut y revenir. Les informations concernant le calcul ont été dépilées ou écrasées et il n'y a pas de moyen portables de prévenir cela. Le mécanisme d'appel de C ne peut donc pas simuler le mécanisme d'appel de Scheme.

Malgré les contraintes qu'imposent nécessairement les continuations sur l'architecture d'un interprète, nous tenons tout de même à les inclure dans notre système.

2.5 Traitement des appels terminaux

Une caractéristique de Scheme est le traitement des appels terminaux. Les implantations de Scheme sont tenues de les implanter de façon appropriée, ceci afin de permettre au programmeur Scheme d'exprimer des itérations à l'aide de fonctions récursives terminales et d'assurer que le traitement se fasse sans consommation d'espace.

Afin d'illustrer ce que ceci signifie, nous présentons tout d'abord une définition d'appel terminal, ensuite nous montrons la façon correcte de l'implanter et, enfin, nous discutons brièvement de l'impact de cette contrainte sur une implantation de Scheme.

2.5.1 Définition d'appel terminal

Toutes les expressions d'un programme sont soit en position terminale, soit en position non-terminale. Intuitivement, une expression est en position terminale si elle est le dernier calcul à effectuer dans une fonction. En particulier, un appel est terminal s'il est le dernier calcul à faire dans le corps d'une fonction.

Prenons par exemple l'appel `(+ x y)` dans l'expression suivante:

```
(define counted-sum
  (lambda (x y)
    (set! sum-counter (+ sum-counter 1))
    (+ x y)))
```

L'appel `(+ x y)` est le dernier calcul à effectuer dans le corps de la fonction et est donc en position terminale.

Toutefois, dans l'expression `(set! z (+ x y))`, le même appel ne peut être en position terminale car il reste l'affectation à effectuer. L'appel n'est donc pas terminal, quelle que soit la position de l'expression `set!` qui l'englobe.

On peut donner pour chaque forme syntaxique primitive les sous-expressions qui sont en position terminale si et seulement si la forme elle-même l'est. L'exception est la lambda-expression: la dernière expression de son corps est en position terminale dans tous les cas. Les sous-expressions qui nous intéressent sont en gras.

- Les références de variables et les constantes littérales ne comportent pas de sous-expressions.
- Appel de fonction: `(`$\langle E_0 \rangle$ $\langle E_1 \rangle$ `...` $\langle E_n \rangle$`)`. Aucune des sous-expressions dans l'appel n'est le dernier calcul à effectuer. Même après l'évaluation du dernier argument, il reste à invoquer la fonction sur ses arguments.
- Lambda-expression: `(lambda` ⟨formels⟩ $\langle E_1 \rangle$ `...` $\langle E_{n-1} \rangle$ $\langle \mathbf{E_n} \rangle$`)`. La dernière expression du corps est le dernier calcul à effectuer et est toujours en position terminale.
- Expression conditionnelle: `(if` $\langle E_1 \rangle$ $\langle \mathbf{E_2} \rangle$`)` ou `(if` $\langle E_1 \rangle$ $\langle \mathbf{E_2} \rangle$ $\langle \mathbf{E_3} \rangle$`)`. Le test n'est jamais le dernier calcul à effectuer dans une condition. Le résultat est repris et détermine quelle branche sera exécutée. La branche choisie est toutefois le dernier calcul à effectuer dans la condition.
- Affectation et définition: `(set!` ⟨variable⟩ ⟨E⟩`)` et `(define` ⟨variable⟩ ⟨E⟩`)`. La sous-expression présente dans les affectations et les définitions n'est jamais en position terminale à cause du fait que son résultat doit être placé dans une variable avant que l'expression complète ne retourne son résultat.

Nous ne tenons pas à énumérer chacune des formes syntaxiques dérivées qui existent en Scheme et décrire lesquelles de leurs sous-expressions sont en position terminale. Le standard décrit les transformations source à source qui permettent de ramener les formes dérivées en des formes primitives. Une fois la forme réduite, il suffit de suivre les règles que nous avons données pour trouver les expressions en position terminale.

2.5.2 Implantation correcte

Les continuations permettent de comprendre rapidement ce qui doit être fait dans une implantation correcte. Pour ce, prenons l'exemple suivant:

```
(define write-and-decode
  (lambda (n)
    (write n)
    (- 200 n)))
```

Le corps de la fonction comporte deux appels. Le premier n'est pas en position terminale et le second l'est. Voici une façon de convertir la fonction en style CPS:

```
(define write-and-decode
```

```
  (lambda (n k)
    (write n (lambda (result1)
               (- 200 n (lambda (result2)
                          (k result2)))))))
```

Si l'on observe attentivement l'exemple converti, on remarque qu'il est inutile de créer une nouvelle continuation pour le deuxième appel. En effet, elle ne fait qu'intercepter le résultat de la soustraction avant de le passer à la continuation `k`. Si on avait plutôt converti la fonction de la façon suivante:

```
(define write-and-decode
  (lambda (n k)
    (write n (lambda (result)
               (- 200 n k)))))
```

elle aurait le même comportement sauf qu'elle ne créerait pas de continuation inutilement.

La deuxième conversion représente en fait la façon correcte d'implanter l'appel terminal. Comme l'appel terminal est le dernier calcul de la fonction, c'est lui qui doit se charger de retourner le résultat (le passer à `k`). C'est la fonction `-` qui se charge de retourner le résultat à la place de `write-and-decode`. Ceci contraste avec les langages plus traditionnels où les fonctions retournent toujours elles-mêmes leur résultat. Dans ces langages, le mécanisme d'appel est représenté par la première des deux conversions CPS.

L'intérêt d'implanter les appels terminaux tel que spécifié par le standard n'est pas clair dans l'exemple. Toutefois, si on considère l'exemple suivant, on prend vite conscience de l'intérêt de cette méthode d'implantation:

```
(define destructive-reverse
  (lambda (original new)
    (if (null? original)
        new
        (let ((tail (cdr original)))
          (set-cdr! original new)
          (destructive-reverse tail original)))))
```

La fonction de cet exemple sert à renverser des listes par des mutations sur les paires. Essentiellement, cette fonction exprime un calcul itératif. Dans un langage comme C, elle serait implanté avec une boucle et trois variables. Or, ici, il s'agit d'une fonction récursive. Elle se rappelle récursivement pour chaque paire de la liste.

On remarque que l'appel récursif se trouve en position terminale. Si les appels terminaux ne sont pas implantés comme le standard de Scheme l'exige, cette fonction utilise un espace proportionnel à la longueur de la liste. S'ils le sont, alors l'exécution de la fonction demande seulement un espace constant. En termes de continuations, la différence vient du fait qu'une chaîne de continuations inutiles est créée dans une mauvaise implantation. Selon la représentation interne

des continuations, ceci représente soit un empilement d'informations déjà rendues inutiles, soit la création d'une chaîne de structures de données inutiles.

2.5.3 Impact sur l'interprète

Comme dans le cas des continuations comme objets de première classe, le traitement spécial des appels terminaux impose des contraintes sur l'architecture d'un interprète Scheme. Cette caractéristique ne se retrouve pas dans les langages plus conventionnels comme le C. Donc, en général, un appel en Scheme ne peut être simulé par le mécanisme d'appel de C. On le savait déjà à cause de la présence des continuations, mais la présence des appels terminaux constitue une raison supplémentaire.

Bien que ce traitement complique l'implantation d'un interprète, il est préférable de l'effectuer. Ce traitement spécial des appels terminaux est demandé par le standard mais, de plus, il constitue une façon d'économiser de l'espace. Un programme pour le micro-contrôleur a avantage à être écrit en utilisant au maximum des fonctions récursives terminales. En effet, le fait d'éviter de sauver les informations nécessitées par un retour inutile peut rendre possible des calculs demandant de nombreuses itérations. Sans le traitement spécial ou en n'utilisant que des appels non terminaux, la mémoire réduite du micro-contrôleur se remplit vite.

2.6 Gestion mémoire

Tout système Scheme doit être doté d'un GC afin de récupérer de l'espace mémoire. En effet, il n'y a aucune façon de libérer explicitement l'espace occupé par un objet ou une variable. Un de nos objectifs consiste à implanter un GC temps réel pour éviter que des pauses se produisent durant l'exécution des programmes.

La définition de GC temps réel change légèrement d'un auteur à l'autre. C'est pourquoi nous donnons la définition que nous avons adopté dans le cadre de la conception de notre GC. Ensuite, nous discutons brièvement de l'intérêt d'une telle définition.

2.6.1 Définition

GC bloquant

Les GC conventionnels sont des GC bloquants. C'est-à-dire que la plus grande partie du temps, un tel GC n'est pas en fonction. Le système de gestion mémoire alloue des objets à l'intention de l'application jusqu'à ce qu'il ne reste plus d'espace libre. C'est à ce moment que le GC est déclenché. Celui-ci s'accapare alors de tout le temps d'exécution pour effectuer son travail de récupération de mémoire. Lorsqu'il a terminé son cycle et qu'il a reconstitué une banque

d'espace libre, et seulement à ce moment, il redonne le contrôle à l'application. Celle-ci peut alors continuer à exécuter et, entre autres, à demander des allocations d'objets. Les moments où le GC fonctionne sont ceux où l'application est bloquée, d'où le nom de GC bloquant.

Les bloquages de l'application, dépendamment de la nature de celle-ci, ne sont pas toujours tolérables. C'est là où un GC temps réel prend tout son intérêt.

Fonctionnement général d'un GC temps réel

Un GC temps réel est un GC qui organise son travail de récupération de la mémoire de façon à minimiser les bloquages qu'il cause à l'application. Au lieu de n'être déclenché que lorsqu'il ne reste plus d'espace libre, il est déclenché à intervalles réguliers rapprochés mais il n'effectue à chaque fois qu'une toute petite partie du travail. Son cycle n'est donc pas fait de façon atomique mais plutôt est entremêlé avec le cours de l'exécution de l'application.

Bien qu'un des objectifs d'un GC temps réel soit de minimiser la durée des bloquages qu'il cause à l'application, il n'en reste pas moins qu'il se doit de remplir son rôle fondamental de récupération de mémoire. L'application ne doit pas manquer d'espace libre parce que le GC fonctionne à un rythme trop lent. Donc, le GC doit être invoqué assez souvent et effectuer à chaque fois une quantité suffisante de travail.

Critère à respecter

Selon nous, un GC temps réel doit respecter le critère que nous énonçons un peu plus bas. Ce critère procède par comparaison. Soit S* un interprète Scheme dont le système de gestion mémoire comporte un GC temps réel. Soit S un interprète dont on a changé le système de gestion mémoire par celui-ci: un allocateur (pas de récupérateur) opérant sur une mémoire infinie non-initialisée.

Le critère compare le temps nécessaire pour faire une opération donnée sur S et sur S*. Dénotons par T et T^* les fonctions mesurant la durée de l'exécution d'une opération primitive dans les interprètes S et S* respectivement. Le critère à respecter est le suivant: il existe une constante c telle que pour toute opération e sur des objets:

$$T^*(e) \leq cT(e)$$

Les opérations concernées sont celles ayant trait à la création et à la manipulation des objets. Ainsi, on se restreint aux effets observables du GC temps réel. Si on permettait que e soit n'importe quelle expression, on inclurait les effets de choix d'implantation qui n'ont aucun lien avec la gestion mémoire. Par exemple, la représentation des symboles, des continuations, etc.

La constante c dépend principalement de la quantité de mémoire utilisée par des objets "vivants" du programme. Lorsqu'il y a plus d'objets vivants conservés par le programme, le

tas est constamment plus rempli. Un tas plus rempli nécessite des cycles de GC plus fréquents. Dans le cas d'un GC temps réel, ceci représente une plus grande quantité de travail à effectuer à chaque période de travail. Donc une augmentation de la constante c. Nous discutons plus longuement de l'effet de l'occupation mémoire dans le chapitre sur le GC.

Le critère devient trivial si l'on permet à la fonction T d'avoir des images très grandes. Nous supposons que l'interprète S est un interprète "raisonnable". Les exemples suivants illustrent des temps raisonnables pour plusieurs opérations de Scheme.

L'accès dans une paire, dans un vecteur ou la création d'une paire sont des opérations faisables en temps constants.

$$T(\texttt{car}), T(\texttt{vector-ref}), T(\texttt{cons}) \in O(1)$$

Le calcul de la longueur d'une liste et la création d'une liste, d'un vecteur ou d'une chaîne de caractères sont faisables dans un temps proportionnel à la longueur de l'objet concerné.

$$T(\texttt{length}), T(\texttt{list}), T(\texttt{make-vector}), T(\texttt{make-string}) \in O(n)$$

Le critère fait donc en sorte que les opérations dans l'interprète S* ont la même complexité que dans l'interprète S.

2.6.2 Intérêt de la définition

Cette définition est beaucoup plus près du point de vue du programmeur Scheme que de notre point de vue d'implanteur de l'interprète. C'est ce qui fait son intérêt à notre avis. Si l'utilisateur désire développer une application qui ait un temps de réponse court à certains moments donnés, il n'a qu'à prêter attention aux opérations que son programme effectue à ces moments-là. Les opérations présumées rapides ne causent pas de bloquage à proprement parler. Les opérations supposant une recherche dans une structure de données ou une création importante d'objets causent des bloquages d'une longueur conséquente.

Il est important de comprendre que, dans un système Scheme doté d'un GC bloquant, la durée d'une opération primitive n'est pas prévisible de cette façon. En effet, lorsqu'une opération primitive déclenche le GC bloquant, celui-ci effectue sur-le-champ la récupération dans tout le tas. La quantité de travail à faire lors de cette récupération est généralement importante et est sans rapport avec la quantité de travail à faire par l'opération primitive. Lorsque le GC est déclenché durant une opération primitive, la durée de celle-ci est augmentée significativement. Il devient alors impossible de prévoir la durée de l'exécution d'un bout de programme.

2.7 Traitement d'erreurs

Notre interprète ne respecte pas une des demandes du standard ([R^{4}RS]) à propos des erreurs d'exécution. Nous n'avons pas l'intention de signaler les erreurs. Par conséquent, nous ne nous

occupons pas de la détection d'erreurs non plus.

De toute évidence, ceci va à l'encontre d'un des avantages d'utiliser Scheme que nous mentionnons dans l'introduction. L'interprète peut se mettre à se comporter bizarrement ou bloquer si le programme en exécution effectue une opération illégale.

La raison majeure de ce choix vient encore de notre objectif de faire une implantation pour micro-contrôleur. Le micro-contrôleur n'est normalement pas utilisé pour fonctionner avec une connexion à un terminal. Le programme (Scheme ou pas) ne doit plus comporter d'erreurs de conception lorsqu'il est utilisé pour diriger le micro-contrôleur dans sa tâche. Que le programme ait ou non la possibilité de déclarer qu'il ne fonctionne pas correctement ne change rien au fait qu'il ne remplit pas sa tâche et il est considéré comme un échec.

Une autre raison, qui est plus une heureuse conséquence qu'une raison, est le fait qu'ainsi, l'interprète n'a pas à comporter de code de détection d'erreurs et s'en trouve plus compact.

Nous faisons donc l'hypothèse que le programme fourni par le programmeur est parfaitement correct. C'est pourquoi nous considérons que notre implantation n'est pas orientée vers le développement d'applications. Un programme doit normalement être développé et testé sur une machine plus puissante et sur une implantation de Scheme plus conviviale.

2.8 Question d'efficacité

Etant donné que la limite sur la taille de l'interprète est une contrainte très forte imposée par le micro-contrôleur, nous avons visé une écriture très concise pour le noyau de l'interprète et la librairie standard. Sans être complètement incompatibles, la concision et l'efficacité en temps sont plus ou moins inconciliables. Nous ne visons donc pas une implantation particulièrement efficace en temps.

Toutefois, nous avons l'intention de faire en sorte que le coût de l'utilisation de chaque fonction de la librairie soit compatible avec ce que l'on peut espérer d'une "implantation raisonnable". Nous présentons plus haut quelques exemples de temps d'exécution attendus d'une implantation raisonnable (section 2.6.1). Nous ne les reprenons pas ici.

Chapitre 3

Représentation des objets

Dans ce chapitre, nous discutons de plusieurs aspects liés à la représentation des objets. La première section traite du typage des objets Scheme. Ceci est une question centrale puisque Scheme est un langage typé dynamiquement. Ensuite, une brève comparaison est faite entre les choix de représentation des symboles et des continuations. Finalement, diverses représentations pour les environnements sont comparées. Dans chaque comparaison, nous mentionnons quelle représentation a été choisie.

Bien que les détails d'un choix de représentation des objets soit très reliée au type de GC utilisé, nous essayons d'en rester le plus détachés possible.

3.1 Représentation des types

Il existe de nombreuses façons de représenter les types dynamiquement. De nombreuses techniques sont présentées dans [Gud93]. Les deux premières méthodes que nous présentons sont celles que nous utilisons dans différentes versions de l'interprète. Il s'agit de la représentation uniforme et de la représentation taggée. D'autres représentations, comme le typage par la position dans le tas et par le regroupement en pages d'objets du même type, ne semblent pas convenir à nos exigences. Nous ne discutons que brièvement de ces dernières.

3.1.1 Représentation uniforme

La représentation la plus simple est sans doute la représentation uniforme. Les objets de tous les types sont alloués dans le tas. Tous les objets ont un premier champ qui encode le type. Les champs suivants contiennent les informations qui composent l'objet. Leur nombre et leur signification dépendent du type.

Par exemple, une paire est représentée comme suit:

pair	car	cdr

,
un objet à trois champs contenant son type et les deux champs traditionnellement nommés `car` et `cdr`. En connaissant le type, on en déduit que deux champs suivent et que ceux-ci sont des références à d'autres objets. Une chaîne de caractères est représentée comme suit:

string	length	char0	char1	...

.
Le type indique que le second champ est la longueur de la chaîne et que les champs suivants contiennent des données brutes. Le nombre de ces champs dépend de la longueur de la chaîne.

Cette représentation a l'avantage de permettre un traitement simple des objets à cause de son uniformité. Toutefois, elle occasionne un certain gaspillage de mémoire.

Simplicité de traitement

Cette représentation permet de simplifier des tâches comme l'obtention du type ou de la longueur, lorsque c'est le cas. Mais c'est surtout le GC qui est simplifié.

Comme tous les objets sont alloués et qu'ils ont tous le type dans le premier champ, le GC n'a pas à reconnaître spécialement chaque type d'objet. Le format de chaque type peut être encodé dans une table. Le traitement de chaque objet se fait suivant les informations contenues dans la table. Cette approche est adoptée dans les premières versions de l'interprète et permet d'obtenir un algorithme de GC extrêmement compact.

Le champ de type peut être utilisé pour contenir des bits à l'usage du GC. En effet, il y a peu de types en Scheme, donc il reste normalement des bits libres dans ce champ.

Coût élevé en espace

Malgré le fait que cette représentation des types permette une manipulation simple des objets et permette d'implanter les fonctions associées de façon concise, la représentation, prise globalement, n'est pas particulièrement compacte. C'est dans la représentation des objets les plus courts que le gaspillage de mémoire est le plus élevé. Prenons quelques exemples afin de vérifier combien d'espace est pris.

La paire est un objet contenant deux champs. Sa représentation uniforme comporte donc 3 champs: le type et les deux champs de référence de la paire. Il y a un champ ajouté à cause de la représentation. Il n'y a donc pas de problème grave dans le cas de la paire.

Regardons ce qui se passe avec le type *number*. Nous rappelons que nous ne voulons que des entiers dans notre interprète. De plus, leur domaine est limité. L'espace requis pour le stockage d'un nombre est un champ. En représentation uniforme, le nombre prend la forme d'un objet alloué dans le tas et qui comporte deux champs: le type et le nombre. Dans le cas du nombre, il y a une dépense d'espace significative. Un objet de deux champs est créé pour contenir une information aussi peu volumineuse que le pointeur qui le référence! Toutefois, le fait de remplacer

un pointeur par un nombre directement ne nous permettrait pas d'avoir l'information de typage. Il existe tout de même de meilleures façons de représenter les nombres, comme nous le voyons plus loin.

Il est clair que dans le cas du type *char*, la consommation de mémoire est encore plus importante par rapport à la taille de la donnée à représenter.

3.1.2 Représentation taggée

La représentation taggée consiste à modifier certains bits de la référence à un objet pour y inscrire une information de typage. Cependant, il doit rester possible de reconstituer l'adresse exacte de l'objet référencé. L'idée consiste à utiliser des bits de la référence qui ont toujours la même valeur, quelle que soit la position de l'objet.

Habituellement, des bits constants existent pour deux raisons: l'alignement des objets; la taille du tas par rapport à la taille de l'espace d'adressage.

Alignement

Supposons que les champs ont une taille de 4 octets chacun et qu'ils se trouvent tous à des adresses multiples de 4. Il en découle que les objets commencent nécessairement à des adresses multiples de 4. Dans ce cas, l'adresse des objets, une fois écrite en binaire, comporte invariablement 2 zéros dans les positions les moins significatives. On peut donc adopter la convention que les deux derniers bits d'une référence à un objet sont utilisés pour indiquer le type de l'objet désigné. L'adresse de l'objet peut être retrouvée en masquant les deux bits de type.

Taille du tas

Supposons que le tas ait une grandeur de 32 Koctets et que les pointeurs (les champs) comportent 16 bits. Or, il suffit de 15 bits seulement pour identifier sans ambiguïté tous les emplacements dans le tas. Par exemple, si on décide de conserver la distance entre un objet et le début du tas plutôt que l'adresse propre de l'objet, on n'a besoin que de 15 bits. Comme seuls 15 bits sur 16 sont utilisés, le bit le plus significatif est toujours à zéro. Ce bit peut servir à contenir de l'information de typage. L'adresse originale peut être retrouvée en commençant par masquer le bit le plus significatif, etc.

Sous-typage

L'alignement et la taille du tas peuvent permettre de laisser certains des bits d'adressage constants. Ces bits peuvent alors contenir de l'information de typage. Lorsque ces bits sont

suffisamment nombreux, il est possible de trouver une combinaison distincte pour chaque type. Alors, aucun objet n'a besoin de sous-typage.

Lorsque les bits ne sont pas assez nombreux, une combinaison donnée peut servir à représenter plusieurs types distincts. Dans ce cas, un objet d'un de ces types doit comporter de l'information de sous-typage dans ses champs. La solution la plus simple consiste à ajouter un champ de sous-typage devant un tel objet. Cette méthode est similaire au typage de la représentation uniforme. D'autres solutions peuvent utiliser une représentation taggée à nouveau pour encoder le sous-typage. *A priori*, tous les moyens sont bons pour encoder l'information de typage avec la représentation taggée.

Objets non-alloués

La représentation taggée permet d'avoir des objets non-alloués. En effet, jusqu'à maintenant, nous n'avons parlé que de taggage de références. Mais il est possible de tagger des informations brutes. Il suffit d'adopter la convention que lorsque les bits de taggage ont une certaine valeur, cela signifie que les autres bits ne représentent pas une adresse mais plutôt une information brute.

Par exemple, si Scheme ne comportait que des nombres entiers à taille fixe et des paires, on pourrait décider qu'une référence dont le dernier bit est 0 contient l'adresse d'une paire dans les premiers bits et qu'une "référence" dont le dernier bit est 1 contient la valeur d'un entier dans les premiers bits. Dans ce cas, les entiers ne seraient pas des objets alloués. Toutefois, il y a un bit de moins pour encoder la valeur des entiers.

Nombreux cas particuliers

Contrairement à la représentation uniforme, la représentation taggée ne traite pas tous les objets de la même façon. Les objets de certains types sont alloués, d'autres pas. Certains ont un sous-type, d'autres pas.

Souvent, l'irrégularité inhérente à cette représentation fait en sorte qu'il faut des fonctions spécialisées pour chaque opération et pour chaque type. La partie du GC est celle qui se complique le plus. Chaque type peut être alloué ou non, avoir un champ de sous-type ou non, en plus du fait que le nombre de champs requis varie selon le type. Donc, l'ensemble des fonctions reliées à la manipulation des différents types de Scheme est *normalement* plus imposante avec la représentation taggée qu'avec la représentation uniforme.

Il est important de mentionner qu'il est *normalement* plus complexe de manipuler les types sous cette représentation. C'est habituellement vrai, mais, dans certaines implantations de systèmes de gestion de la mémoire, il peut arriver qu'il n'y ait pas beaucoup plus de travail à faire. Par exemple, si les objets alloués sont toujours alignés sur des adresses qui sont des multiples de 16, alors il y a 4 bits (les moins significatifs) qui peuvent être utilisés pour le typage. Ce qui

permet d'encoder 16 types différents et est donc amplement suffisant pour le langage Scheme. La vérification du type se fait toujours de la même façon. Par rapport à la représentation uniforme, la routine de GC n'a qu'à faire un test de plus pour vérifier si un type donné a ses objets alloués ou non.

Toutefois, sur une machine où il manque de mémoire, on ne peut se permettre d'imposer des alignements importants, ni d'utiliser des pointeurs (des champs) trop volumineux. Il est donc inévitable que le taggage des objets soit irrégulier. C'est justement le cas du taggage que nous employons dans notre interprète final. Il est décrit à la section 3.1.4.

Economie d'espace

La représentation taggée permet de réaliser des économies de mémoire, par rapport à la représentation uniforme. Même si l'ensemble des fonctions manipulant les objets devient plus volumineux, la représentation des objets dans le tas est moins volumineuse.

On peut constater une économie d'espace en regardant un exemple. Supposons que le passage à la représentation taggée permette d'éliminer dans les paires le champ de type. Pour chaque paire, on a donc gagné la valeur d'un champ. Supposons aussi que le passage à la représentation taggée cause une augmentation de 256 octets de la taille du code relié à la manipulation des paires. 256 octets de code compilé pour une machine comme un micro-contrôleur permet d'ajouter une quantité raisonnable d'instructions. Si on prend pour acquis que les champs ont 2 octets sur le micro-contrôleur, il suffit d'avoir alloué au moins 128 paires pour commencer à faire des économies de mémoire. Même en accordant 256 octets de plus pour le code faisant la manipulation de chacun des types, un millier de paires suffisent à compenser pour l'accroissement du code. Un millier de paires est un nombre relativement petit et l'accroissement de 256 octets de la taille du code pour le traitement de chaque type est une estimation vraiment pessimiste.

Nous ne *prétendons* pas démontrer avec cet exemple que la représentation taggée permet des économies de mémoire. De toute façon, les économies réalisées dépendent du type des objets utilisés. Or, la quantité d'objets des différents types varie d'un programme à l'autre et il faudrait alors fixer une distribution de probabilité sur les différents types pour avoir une "mesure" qui vaille. De façon concrète, le passage de la représentation uniforme à la représentation taggée a permis de réaliser une économie d'espace appréciable dans le cas de notre interprète, comme nous le montrons un peu plus loin.

Réduction du domaine

L'utilisation de bits de tag pour typer les références aux objets peut demander de réduire le domaine de certains types. Il s'agit des types pour lesquels les objets ne sont pas alloués. Les bits de tag laissent moins qu'un champ complet pour encoder la donnée brute.

Dans notre interprète, le type *number* voit son domaine restreint par le taggage. Comme

c'est un choix d'implantation de ne pas avoir de nombres de taille illimitée, ce n'est pas plus arbitraire de fixer la taille du domaine à une taille plus petite que celle qui serait disponible si un champ complet était utilisé. Malgré tout, il est important de minimiser cette réduction du domaine imposée aux nombres. Premièrement, parce que les mots sur le micro-contrôleur ne sont que de 16 bits. Deuxièmement, parce que les programmeurs habitués de programmer le micro-contrôleur en assembleur ou en C y verraient une perte de puissance trop importante. La section 3.1.4 montre le taggage utilisé sur les références et en particulier le taggage des nombres.

3.1.3 Les autres représentations

Outre les représentations uniforme et taggée, il existe de nombreuses autres façons de typer dynamiquement les objets. Nous en décrivons deux brièvement. Il s'agit du regroupement en zones selon le type et du regroupement des objets en pages d'objets du même type.

La première technique consiste à séparer l'espace du tas en autant de zones qu'il y a de types. L'allocation d'un objet d'un type donné doit se faire dans la zone correspondante et on reconnaît le type d'un objet par la zone dont il fait partie. De cette façon, il n'est pas nécessaire de tagger les références aux objets ni d'ajouter un champ de type à ces derniers.

Les séparations entre les zones peuvent être fixes ou être mobiles et adaptables aux besoins courants du programme.

Sur le micro-contrôleur, il y a peu d'espace et il n'est pas envisageable d'utiliser des séparations fixes entre les zones. En effet, un programme qui utilise principalement un des types de Scheme se retrouverait très rapidement à court d'espace pour ce type.

Il faudrait donc utiliser des séparations mobiles entre les zones. De cette façon, si le type le plus utilisé change avec le temps, la taille des zones pourra être adaptée en conséquence. Cette technique semble convenir. Du moins, jusqu'à ce que l'on tente de la faire fonctionner avec un GC temps réel. Nous n'avons pas de solution simple à ce problème et nous allons en discuter plus longuement dans le chapitre sur le GC.

La deuxième technique que nous mentionnons est celle où les objets sont regroupés par pages d'objets du même type. Les pages sont des blocs de mémoire d'une taille pré-définie et qui débutent toujours à des adresses qui sont des multiples de la taille des blocs. Une entête se trouve au début de chaque page afin d'indiquer entre autres le type des objets de la page. A l'allocation d'un objet d'un type donné, une des places libres dans une page dédiée à ce type est réservée. Une page dédiée au type en question doit être créée s'il n'y a pas de place disponible dans les pages existantes. Le type d'un objet s'obtient donc en retrouvant le début de la page qui le contient (par un simple arrondissement) et en lisant le type qui y est indiqué. Les objets eux-mêmes n'ont pas à avoir de champ de type et les références n'ont pas à être taggées. Des cas particuliers doivent être prévus pour gérer les chaînes de caractères et les vecteurs car ils peuvent atteindre des dimensions plus grandes que les pages.

Type	Encodage	Domaine
Entiers	`nnnnnnnnnnnnnnn1`	-16384 à 16383
Paires	`00rrrrrrrrrrrrr0`	max. 8192
Fermetures	`01rrrrrrrrrrrrr0`	max. 8192
Autres types alloués	`10rrrrrrrrrrrrr0`	max. 8192
Symboles	`11nnnnnnnnnnnn10`	max. 4096
Caractères	`11xxnnnnnnnn0000`	complet
Fonctions du noyau	`11nnnnnnnnnn0100`	complet
Booléens	`11xxxxxxxxxn1000`	complet
Liste vide	`11xxxxxxxxxx1100`	complet

Les bits contenant `0` ou `1` sont ceux qui servent à indiquer le type. Les bits représentés par un `n` sont des bits qui contiennent directement l'information d'un objet non-alloué. Les bits représentés par un `r` sont des bits qui contiennent l'adresse d'un objet alloué. Les bits représentés par `x` ne servent ni pour le typage ni comme information. Ils sont arbitrairement fixés à `1` dans notre implantation.

FIG. 3.1 - *Description du taggage*

A nouveau, nous voyons dans le chapitre sur le GC que nous n'avons pas de technique temps réel simple qui puisse travailler sous une telle représentation. La simplicité de l'algorithme est importante car le code du GC ne doit pas consommer tout l'espace sauvé grâce à l'utilisation de la représentation.

3.1.4 Nos choix d'implantation

Les différentes versions de notre interprète utilisent les représentations uniforme et taggée. Les premières versions utilisent la représentation uniforme. Nous ne prendrons pas la peine de décrire quel numéro était associé à chaque types. Par la suite, nous adoptons la représentation taggée. Le taggage change un peu dans les versions suivantes, suite à des optimisations, etc. Nous donnons ici un résumé de la représentation adoptée dans la dernière version (voir la figure 3.1).

Dans notre représentation finale, les objets devant être alloués dans le tas sont: les paires, les fermetures, les continuations, les vecteurs et les chaînes de caractères. Les objets de tous les autres types, que l'on appelle des types immédiats, ne sont pas alloués, ils sont codés à même la "référence". Il y a des remarques à faire à propos de certains types.

Premièrement, les nombres ne sont pas alloués et n'ont qu'un seul bit de tag. Il est important de ne pas allouer les nombres dans le tas pour permettre d'économiser l'espace. Néanmoins, nous avons fait un effort particulier afin d'utiliser le moins de bits possible pour tagger les nombres et de ne pas trop réduire le domaine des nombres. Les nombres peuvent donc être représentés sur le domaine -16384 à 16383, plutôt que sur le domaine -32768 à 32767 que les 16 bits d'un

Sous-type	Premier champ
Continuations	`rrrrrrrrrrrrrrr1`
Vecteurs	`1111111111111100`
Chaînes de caractères	`1111111111111110`

Le sous-typage est fait en taggant le premier champ. Dans le cas de la continuation, le sous-type est codé avec le point de retour dans le code-octets (`r`) (voir section 3.3). Dans les autres cas, c'est avec la longueur (`l`) qu'est codé le sous-type. Cet encodage limite les vecteurs et les chaînes à des longueurs inférieures à 16384. La taille du code-octets est limitée à 32767 octets à cause des 15 bits alloués à l'encodage de l'adresse de retour dans les continuations.

FIG. 3.2 - *Sous-typage des objets regroupés sous le type* "Autres types alloués"

champ complet (ou mot de la machine) permettent de représenter. La réduction du domaine est donc minime.

Deuxièmement, il y a deux sortes de fonctions Scheme dans notre interprète: les fonctions du noyau et les fermetures. Les premières sont peu nombreuses et correspondent aux fonctions qui sont implantées directement dans le noyau de l'interprète. Les fonctions de base pour la manipulation des objets des différents types sont de ce nombre. Les fonctions du noyau ne sont pas allouées, elles sont simplement représentées par un numéro de fonction taggé. Les autres fonctions de la librairie et les fonctions créées par le programme sont des fermetures. Ce sont des structures de données qui contiennent un point d'entrée et un environnement de définition, donc elles doivent être allouées dans le tas en tant qu'objet à deux champs.

Troisièmement, les continuations, les vecteurs et les chaînes de caractères n'ont pas leur propre tag. Ils sont regroupés sous le tag "autres types alloués" et doivent posséder un sous-type dans leur premier champ. La figure 3.2 montre les tags associés à ces sous-types. Les continuations ont leur tag dans le même champ que le point de retour (voir section 3.3). Les vecteurs et les chaînes ont leur tag dans le même champ que leur longueur. Il s'agit des longueurs en nombre de champs et en nombre de caractères respectivement. Les champs qui suivent contiennent soit des références à des objets dans le cas des vecteurs, soit des codes Ascii dans le cas des chaînes, en nombre variable.

Quatrièmement, les symboles sont des objets non alloués. Un symbole n'est en fait qu'un numéro taggé. Nous discutons de la représentation des symboles dans la prochaine section.

Le passage à la représentation taggée permet normalement d'économiser de l'espace mémoire. Ce fut le cas pour notre interprète. Toutefois, il est difficile de faire une évaluation juste à ce sujet car la représentation des objets est fortement reliée au GC. Certains GC peuvent avantager certaines représentations. Par exemple, même si un objet alloué ne nécessite pas de sous-typage, le GC peut faire en sorte qu'un champ supplémentaire lui soit ajouté tout de même. Nous nous contentons de donner tout simplement l'amélioration que nous avons observée lors du

changement.

Dans les versions où nous faisons le passage d'une représentation à l'autre, un GC bloquant à deux semi-espaces est utilisé. De plus, l'interprète charge au départ toute la librairie Scheme avant de pouvoir exécuter des commandes tapées au clavier. C'est la librairie qui constitue les données utilisées pour mesurer la compacité de la représentation. Le fait de tagger les références aux objets et de ne carrément plus allouer les types immédiats permet d'améliorer la représentation des types suivants: les nombres, les symboles, les caractères, les fonctions du noyau, les booléens et la liste vide. Les paires et les fermetures ne perdent pas de champs à cause d'exigences venant du GC. Le champ de typage de la représentation uniforme est conservé pour pouvoir stocker le bit de marquage du GC. Les chaînes de caractères perdent un champ à cette occasion. Toutefois, ceci n'est pas dû au taggage, mais plutôt à un changement de représentation indépendant du typage. L'espace occupé par la librairie passe d'environ 32000 champs à environ 29000. Les champs en question sont des mots machine de 16 bits. La plus grande partie de cet espace est pris par les paires utilisées pour représenter les expressions Scheme. Ceci explique que les amélioration apportées par le taggage ne soient pas plus importantes.

3.2 Les symboles

Deux représentations différentes des symboles sont utilisées dans les versions de l'interprète. De plus, dans les premières versions, les symboles sont utilisés pour contenir la valeur des variables globales du nom correspondant. Dans les dernières, seul le nom du symbole reste.

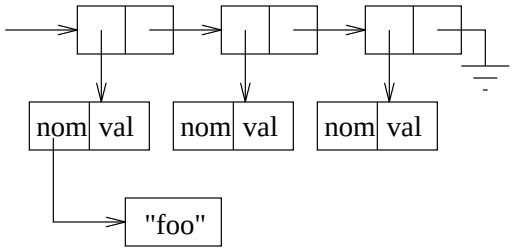
3.2.1 Contenant directement le nom

Cette façon de représenter les symboles est probablement la plus immédiate. Il s'agit simplement d'avoir un objet à deux champs: un pour le nom et un pour la valeur de l'éventuelle variable globale correspondante. Le champ du nom contient une référence à une chaîne de caractères. Lorsqu'il n'y a pas de variable globale correspondant au symbole, le champ de la valeur contient une valeur par défaut (nous utilisons `#f`). Les expressions `define` et `set!` modifient ce champ.

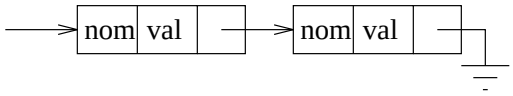
De plus, il faut maintenir une liste de tous les symboles existants dans l'interprète. Ceci est nécessaire car il ne peut y avoir qu'un seul symbole d'un nom donné. Si une fonction comme `read` ou `string->symbol` doit retourner un symbole ayant le même nom qu'un symbole existant, alors le symbole retourné est *le* symbole déjà existant [1]. A chaque fois qu'un symbole d'un nom

1. Ceci n'est pas parfaitement exact. Le standard ne requiert pas explicitement qu'il s'agisse du même objet. Ce qui est exigé par le standard, c'est que les symboles ayant le même nom soient équivalent au sens de `eq?`. Or, dans la plupart des implantations, `eq?` est implanté à l'aide d'une comparaison de pointeurs, donc très efficacement. Une comparaison des noms lors d'une comparaison entre symboles rendrait `eq?` beaucoup moins efficace. Les programmeurs Scheme s'attendent en général à une implantation de `eq?` très efficace. C'est pourquoi nous nous assurons de n'avoir qu'au plus un symbole de chaque nom.

encore inconnu est créé, il est ajouté à la liste.



Il est possible d'éviter l'allocation de la paire lors de chaque allocation de symbole. Il s'agit d'ajouter un champ à chaque symbole. Ce champ sert à chaîner les symboles en une liste. Un champ de plus par symbole est une dépense de mémoire moins importante qu'une paire par symbole.



Il est possible de ne pas avoir de chaîne de caractères pour chaque symbole en faisant en sorte que les symboles eux-mêmes soient des objets à taille variable contenant directement leur nom. Cette représentation est encore plus compacte que la précédente. Toutefois, le traitement des objets à taille variable a tendance à être compliqué, comme nous le voyons dans le chapitre sur le GC, et il n'est pas souhaitable d'ajouter un type à taille variable dans le système.

3.2.2 Numérotation des symboles

L'autre façon de représenter les symboles que nous utilisons consiste à assigner un numéro à chaque symbole. Un symbole devient alors un entier taggé. Le nom du symbole, sous forme de chaîne de caractères Scheme, est stockée dans un vecteur regroupant les noms de tous les symboles. Si la valeur des variables globales est stockée avec les symboles, il suffit d'avoir un deuxième vecteur qui regroupe toutes les valeurs. Lorsque la fonction `read` ou `string->symbol` s'apprête à retourner un symbole, elle peut vérifier l'existence du symbole en parcourant le vecteur des noms. Si les vecteurs sont pleins lors de l'ajout d'un nouveau symbole, de nouveaux vecteurs plus longs sont alloués et remplacent les anciens.

Cette représentation est compacte. Elle requiert le même nombre de champs par symbole que la représentation suggérée où un symbole est une structure à taille variable contenant le nom, la valeur et la référence au symbole suivant. En effet, ceci vient du fait que les vecteurs sont des structures de données compactes. Lorsqu'il y a suffisamment de symboles, les vecteurs deviennent longs et l'espace pris en moyenne pour contenir le nom et la valeur s'approche de 2 champs. On obtient donc une efficacité en espace presqu'aussi bonne que dans le cas de la structure contenant toutes les informations, mais sans introduire de nouveau type complexe à manipuler. Un numéro taggé est très simple à traiter par le GC.

Malgré tout, l'évaluation de l'espace pris par cette représentation est plutôt optimiste. En effet, l'évaluation suppose que les vecteurs contenant les noms et les valeurs sont pleins. Cette supposition implique que les vecteurs doivent être aggrandis à chaque allocation de symbole. Ceci n'est pas tellement efficace. Ce que nous faisons donc en pratique consiste à allouer de nouveaux vecteurs d'une longueur $\frac{4}{3}$ fois plus grande lorsque les vecteurs actuels sont pleins. Donc, en pire cas les vecteurs sont aux trois quarts pleins. L'espace pris en moyenne (sur l'ensemble des symboles) pour contenir le nom et la valeur s'approche de $2\frac{2}{3}$ champs, en pire cas (lorsque les vecteurs sont les plus vides). Il est possible de diminuer le facteur d'aggrandissement, ayant ainsi moins de perte d'espace, mais augmentant la fréquence des agrandissements des vecteurs.

Cette représentation des symboles est utilisée à partir de la version où le taggage est utilisé pour typer les objets. Le champ pour contenir la valeur de la variable globale est abondonné à partir de la version où l'interprète n'est plus interactif. La compilation s'occupe d'identifier les variables globales et leur espace est réservé de façon statique. Dans un système compilé, il n'est pas nécessaire de conserver le symbole nommant une variable, sauf lorsque le système fournit la fonction `eval`. Toutefois, cette fonction est non-standard et ne se retrouve pas dans notre système. Il n'y a donc plus de lien entre l'existence des symboles et des variables globales.

Cette représentation avec un ou des vecteurs a l'avantage d'être compacte. En fait, on peut rendre la taille moyenne de la représentation de chaque objet aussi proche que l'on veut du nombre de champs "utiles" de l'objet. Par "utiles", nous désignons les champs qui contiennent l'information que doit contenir un objet, par définition, et non ceux qui ne servent qu'à des tâches administratives. On pourrait être tenté d'utiliser cette représentation pour les objets d'autres types. Deux vecteurs pour représenter les paires, par exemple.

Malheureusement, les objets des autres types ne sont pas conservés indéfiniment comme les symboles. Un symbole, une fois créé, est conservé pour toute la durée de l'exécution du programme pour assurer son unicité. Avec les autres types, les vecteurs se rempliraient rapidement de champs d'objets abandonnés. Il faudrait effectuer une collecte d'objets abandonnés à l'intérieur même des vecteurs. Cette représentation finirait par avoir des défauts similaires à ceux de la représentation des types par regroupement en zones (voir section 3.1.3).

3.3 Les continuations

La représentation des continuations est fortement liée au modèle d'exécution utilisé dans un système Scheme. Elle peut donc changer radicalement d'un système à l'autre.

Nous présentons trois façons différentes de les représenter. La première consiste en une pile. La seconde, en des fermetures. La troisième, en des structures de données *ad hoc*. Enfin, nous décrivons la représentation employée dans notre interprète.

3.3.1 Pile

Plusieurs implantations de Scheme utilisent une pile. Celle-ci sert à faire le passage d'arguments, indiquer les points de retour, etc. Comme nous le mentionnons à la section 2.4.4, il n'y a habituellement pas de façon portable d'utiliser la pile du langage d'implantation. Une pile doit être gérée explicitement.

Lorsque la continuation est réifiée avec `call/cc`, une copie de la pile est créée dans le tas. L'activation de cette continuation cause un remplacement de la pile actuelle par la pile sauvée.

Cette représentation est très efficace lorsque les programmes n'utilisent pas `call/cc`. Toutefois, lorsqu'il y a fréquemment des réifications de la continuation courante, les copies demandent beaucoup de temps et d'espace car il n'y a pas de partage d'information entre les continuations.

Malgré tout, il existe des techniques plus sophistiquées qui effectuent les copies partiellement, au besoin. Un morceau de la pile n'est sauvé que lorsqu'il est sur le point d'être détruit par le dépilement. Ainsi, la pile est copiée aussi peu que possible. Ces techniques rendent la réification des continuations plus efficaces et moins gourmandes en espace mémoire.

3.3.2 Fermetures

Une façon de représenter les continuations, consiste à les rendre explicites par une conversion CPS. Après cette conversion, toutes les continuations sont devenues des fermetures à un argument.

L'avantage principal de cette méthode est d'éviter d'avoir à créer des objets du type "continuation" explicitement. Tout le mécanisme des continuations est pris en charge par des manipulations de fermetures. De plus, le problème de la réification de la continuation courante devient trivial. La fonction `call/cc` peut être impantée comme ceci (en style CPS, naturellement):

```
(define call/cc
  (lambda (proc k1)
    (proc (lambda (val k2) (k1 val))
          k1)))
```

Un inconvénient de cette méthode est la conversion CPS. Celle-ci a tendance à augmenter significativement la taille d'un programme. De nombreuses fonctions administratives sont introduites afin d'éliminer complètement l'utilisation du retour de valeur. Cette augmentation de la taille du programme peut être atténuée lors de la compilation, mais au prix d'analyses assez poussées.

3.3.3 Structures de données *ad hoc*

Au lieu d'effectuer une conversion CPS du programme et de ramener les continuations à des fermetures, on peut créer un type *continuation* dont les objets sont manipulés de façon interne et contiennent le même genre d'informations que les fermetures-continuations.

Le détail des champs d'une structure de type *continuation* dépend énormément du modèle d'exécution du système Scheme. Habituellement, le modèle d'exécution est celui d'une machine virtuelle. Cette machine virtuelle possède quelques registres d'états et sait faire un certain nombre d'opérations.

Un des registres contient toujours une référence à la continuation courante. Un objet de type *continuation* est créé chaque fois que la machine s'apprête à effectuer une série d'opérations qui risquent de modifier l'état de ses registres. Par exemple, un appel non-terminal demande généralement la sauvegarde de l'état de la machine dans une continuation.

La création d'une continuation cause entre autre la sauvegarde du point de retour et de la continuation courante. Le chaînage des structures de continuation permet ainsi de faire l'équivalent d'un empilement. A proprement parler, la continuation est la chaîne des structures.

Le retour du résultat d'un calcul entraîne le rétablissement des registres à l'état sauvé dans la première structure de la chaîne.

Cette représentation a l'avantage d'éviter une conversion CPS. Mais elle nécessite la création d'un type *continuation*. La réification demande de sauver l'état actuel dans une continuation et d'envelopper celle-ci dans une fonction. L'invocation de cette fonction cause le remplacement de l'état de la machine par l'état sauvé. La réification et l'invocation d'une continuation sont donc très peu coûteuses en temps et en espace.

3.3.4 Nos choix d'implantation

Nous utilisons une représentation avec des structures de données *ad hoc* dans toutes les versions de l'interprète. Premièrement, ce choix semble raisonnable *a priori*. Il n'est pas évident de décider laquelle des représentations est la plus compacte, bien que celle utilisant des fermetures semble moins bonne. La représentation à pile est la meilleure lorsqu'il n'y a pas de réifications, puisque toutes les données sauvées sont contiguës comme dans un vecteur. Toutefois, notre implantation permet l'utilisation de `call/cc` et il faut prévoir le cas où la réification est utilisée. La représentation à pile implantée de façon bête demande trop d'espace. Mais une implantation plus sophistiquée pourrait être plus avantageuse que la représentation par des structures *ad hoc*.

Deuxièmement, il est plus facile de gérer un tas qu'une pile et un tas. Dans le cas où une pile est utilisée, il y a deux régions de taille variable qui se disputent le peu de mémoire disponible. Une division fixe entre les deux régions ne permet pas de s'ajuster aux fluctuations d'allocation et d'empilement. La gestion d'une frontière mobile rend le travail du GC particulièrement complexe.

Troisièmement, comme la représentation des continuations est au coeur de l'interprète, un changement de représentation entraîne des changements majeurs dans l'interprète. C'est pourquoi nous n'avons pas essayé d'autres représentations.

Nous ne décrivons pas le contenu précis de la structure que nous utilisons. La machine virtuelle utilisée est décrite à la section 6.1. La liste des registres qui sont sauvés dans une continuation y est donnée.

3.4 Les environnements

Avant de commencer, nous tenons à préciser que nous parlons des environnements lexicaux dans cette section, et non de l'environnement global. Pour éviter une lourdeur inutile, nous utilisons le mot "environnement" pour désigner les environnements lexicaux.

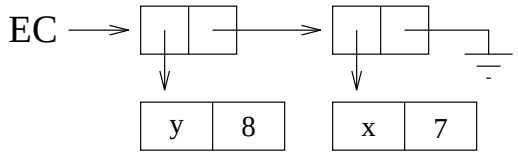
Il existe de nombreuses façons de représenter les environnements de Scheme. Dans certains systèmes, plusieurs représentations sont utilisées conjointement. Une représentation donnée est utilisée tout dépendant de l'usage prévu d'un environnement. Nous voyons ici plusieurs façons distinctes de représenter les environnements. Nous indiquons ensuite les choix que nous avons faits.

3.4.1 Listes d'association

La représentation par des listes d'association est probablement la plus intuitive. Il s'agit de listes dont chaque élément est une paire contenant un symbole (le nom) et un objet Scheme (la valeur). Commençons avec un exemple simple.

```
(let* ((x 7) (y 8)) (+ x y))
```

Au moment de faire la somme de `x` et `y`, l'environnement courant (EC) a l'aspect suivant:

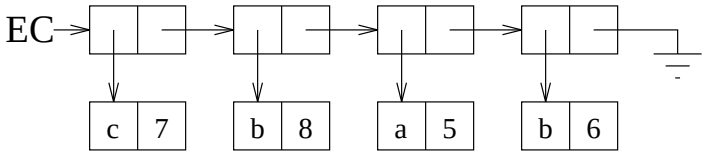


Lorsque l'interprète recherche la valeur de `x`, il parcourt la liste associative jusqu'à la paire contenant le symbole `x`. Il fait de même pour `y`. Similairement, lorsque l'interprète rencontre une expression `set!` opérant sur une variable lexicale, il recherche dans l'environnement courant la paire correspondant au nom de la variable affectée.

En Scheme, il est possible de masquer localement une variable d'un certain nom en ayant

une liaison à une nouvelle variable du même nom. Prenons cet exemple:

```
(let ((a 5) (b 6))
  (let ((c 7) (b 8))
    (+ a b c)))
```



La référence à la variable `b` dans la somme concerne la variable `b` qui est liée dans le `let` interne. Ceci est reflété dans la représentation par le fait que les variables les plus nouvellement liées sont ajoutées à la tête de la liste. La recherche des variables se fait toujours en commençant à la tête de la liste. De cette façon, l'accès aux variables respecte la sémantique de Scheme. Dans le cas d'un accès à une variable globale, le symbole nommant la variable ne se trouve tout simplement pas dans la liste d'association.

Lorsque l'évaluation ressort d'une forme de liaison, l'environnement courant revient à un état antérieur. Ce sont les continuations qui contiennent l'information nécessaire. Il n'y a donc pas d'élimination explicite des variables dans l'environnement courant, il y a simplement des retours aux environnements antérieurs.

Il est assez clair que cette représentation n'est pas la meilleure. Le fait de fouiller la liste associative à la recherche d'un symbole constitue un travail relativement important et fait perdre de l'efficacité. De plus, cette représentation demande deux paires par variable. Elle est donc peu compacte. Un programme utilisant beaucoup de fermetures et de formes de liaisons remplit plus rapidement son espace mémoire. Elle a toutefois l'avantage de ne pas requérir de pré-traitement sur les expressions à évaluer. Toutes les représentations qui suivent demandent un pré-traitement.

Les environnements ainsi représentés ont la propriété de pouvoir partager la queue de leur liste. C'est aussi vrai pour plusieurs des représentations qui suivent. Un exemple du partage des queues est présenté pour la prochaine représentation.

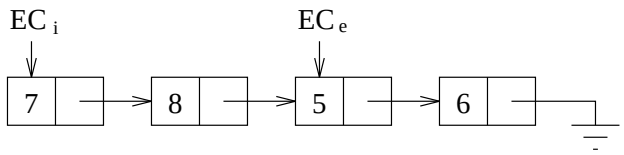
3.4.2 Listes

Une des caractéristiques de Scheme est la portée lexicale des variables. La portée lexicale permet de déterminer de façon statique (à la compilation) la variable concernée par une référence ou une affectation.

Cette représentation est presqu'identique aux listes d'association. La différence principale vient du fait que la liste représentant l'environnement ne contient pas des associations mais des valeurs uniquement. Dans ces listes, il n'y a pas le nom de chaque variable. Donc, une variable n'est pas recherchée par son nom, mais par sa position dans l'environnement. Reprenons cet

exemple:

```
(let ((a 5) (b 6))
  (let ((c 7) (b 8))
    (+ a b c)))
```



Lorsque l'on se trouve dans le `let` externe et hors du `let` interne, l'environnement est représenté par une liste de deux éléments (EC_e). Le premier élément est la valeur de `a` et le deuxième est la valeur de `b`. Lorsque l'on se trouve dans le `let` interne, l'environnement est une liste de quatre éléments (EC_i). Le premier: `c`; le deuxième: `b`; le troisième: `a`; le quatrième ne peut être accédé et est la valeur de la variable `b` masquée.

Cette représentation permet des accès plus rapides aux variables en éliminant la recherche des symboles nommant les variables. De plus, cette représentation ne peut être que plus compacte que celle des listes d'association. En effet, une seule paire est requise par variable.

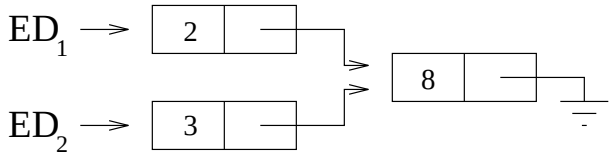
Les environnements représentés par des listes peuvent partager la queue de leur liste. Lors de l'interprétation d'un programme, il y a plusieurs environnements qui existent simultanément: l'environnement courant; les environnements de définition des fermetures; les environnements sauvés par les continuations. Des environnements partagent leur queue lorsqu'ils ont des instances de certaines variables en commun. Prenons par exemple le cas des deux fermetures stockées dans la paire:

```
(define make-thunk
  (let ((delta 8))
    (lambda (epsilon)
      (lambda () ...))))

(cons (make-thunk 2) (make-thunk 3))
```

Les deux fermetures ont un environnement contenant deux variables: `delta` et `epsilon`. Ces variables ne sont pas traitées de la même façon. La variable `delta` est la même pour les deux fermetures. Ce n'est pas le cas de la variable `epsilon`. En effet, si une des deux fermetures mute la variable `delta`, l'autre peut voir le changement en lisant `delta`. Toutefois, si une des fermetures mute *sa* variable `epsilon`, l'autre ne peut le constater. C'est que la variable `delta` est créée une fois pour toutes lors de la définition de `make-thunk`. Tandis que la variable `epsilon` est créée à chaque fois que `make-thunk` est appelée. Le graphique suivant illustre l'état des listes représentant l'environnement de chacune des fermetures. ED_1 et ED_2 sont les environnements

de définition des deux fermetures.



Le partage de la queue des listes se produit aussi avec les représentations par listes d'associations et par blocs chaînés.

3.4.3 Blocs de liaison chaînés

Cette représentation est une variante de la précédente. Plutôt que de systématiquement allouer chaque variable dans une paire séparément, on peut regrouper les variables liées simultanément dans un vecteur. Comme le vecteur est plus compact qu'une liste de même longueur à partir de 2 éléments et plus, on peut ainsi économiser de l'espace.

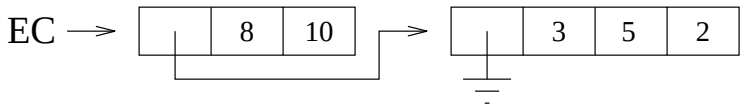
Les variables qui sont liées simultanément sont celles liées dans un `let` ou un `letrec` ou encore lors de l'invocation d'une fermeture à plusieurs paramètres. Evidemment, des formes `let` ou `letrec` ne liant qu'une seule variable ne peuvent prétendre lier des variables simultanément. La forme `let*` ne lie pas ses variables de façon simultanée. Comme il est décrit dans le R^4RS, une forme `let*` à plusieurs variables est sémantiquement équivalente à une série de formes `let` emboîtées les unes dans les autres.

Lorsqu'un bloc ne contient qu'une variable, c'est une paire qui sert à représenter la variable. Quand la variable est liée, la paire est ajoutée devant l'environnement courant. Lorsqu'un bloc contient plusieurs variables, c'est un vecteur qui représente les variables. Le vecteur contient une case par variable et une autre case qui sert à pointer sur le reste de l'environnement. C'est pourquoi nous disons que cette représentation consiste en des blocs de liaison chaînés.

La position d'une variable depuis un certain point du programme peut être décrite par le nombre de blocs à sauter et la position dans le bloc concerné. Une variable dans un bloc à une variable a systématiquement la première (et seule) position. Nous tenons à faire cette observation triviale parce qu'elle a une importance dans la section sur le code-octets (section 6.1.4).

Par exemple, la position de la variable `x` à partir de l'endroit où s'effectue l'addition est: deuxième bloc, première position.

```
(let ((x 3) (y 5) (z 2))
  (let ((a 8) (b 10))
    (+ a b x y z)))
```



Cette représentation a un double avantage sur celle utilisant les listes. Les variables peuvent être atteintes un peu plus rapidement lorsque l'environnement est imposant. Et surtout, il y a une économie d'espace dans la représentation des blocs de liaison d'au moins deux variables. Les blocs de liaison peuvent être partagés de façon analogue à ce qui se produit dans le cas de la représentation par des listes.

La représentation a un léger défaut telle que nous la présentons. Le dernier bloc d'un environnement contient lui aussi une case pour référencer un éventuel bloc suivant. Ce défaut peut être corrigé en faisant en sorte que le compilateur indique à la machine virtuelle qu'un accès est fait à un bloc final, donc que la représentation de celui-ci est légèrement différente.

3.4.4 Blocs de liaison avec *display*

Cette représentation ressemble beaucoup à la précédente. La différence vient du fait qu'un *display* est ajouté à chaque bloc. Un *display* contient un pointeur vers chacun des blocs d'activation qui suivent dans la chaîne, c'est-à-dire vers les blocs d'activation des formes de liaison englobantes. Le *display* est typiquement implanté avec des champs réservés au début du bloc. Plus il y a de blocs qui suivent dans la chaîne, plus il y a de champs qui doivent être réservés.

Cette représentation permet d'atteindre n'importe quelle variable en temps constant: à partir du bloc du départ, il faut emprunter au plus une fois le *display* pour rejoindre le bloc approprié; ensuite, il suffit de lire le bon champ dans le bloc rejoint. La représentation permet d'augmenter l'efficacité des accès aux variables. Toutefois, elle demande plus d'espace la plupart du temps. Du moins, elle ne peut être plus compacte que la représentation par des blocs chaînés. En fait, à chaque fois qu'un bloc est suivi par plus d'un autre dans la chaîne, il est plus volumineux que le bloc correspondant dans la représentation sans *display*. Comme la consommation d'espace est, pour nous, un critère beaucoup plus important que l'efficacité, cette représentation est moins intéressante.

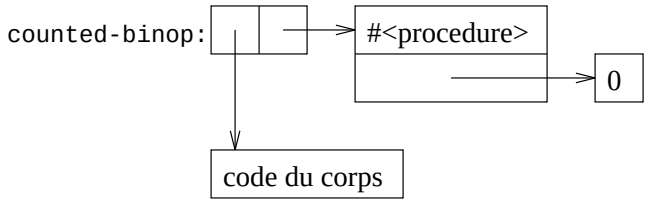
3.4.5 Représentation plate

Cette représentation concerne uniquement les environnements stockés dans les fermetures. Elle ne suffit pas à représenter l'environnement d'évaluation courant. On pourrait donc l'utiliser dans le cadre d'une représentation mixte.

Dans cette représentation, les environnements des fermetures sont constitués d'un tableau (pas nécessairement un vecteur Scheme) ayant une case pour chaque variable de l'environnement de définition. Au moment de la création de la fermeture, la valeur de chaque variable est copiée dans la case appropriée du tableau. Un soin particulier doit être apporté aux variables mutables.

Les variables mutables sont celles qui sont susceptibles d'être modifiées par une expression `set!`. En effet, si deux fermetures ont dans leur environnement de définition la même instance d'une certaine variable, les mutations apportées à cette variable par une fermeture doivent être visibles depuis l'autre fermeture. Dans un tel cas, ce n'est pas la valeur de la variable qui est directement copiée, mais une référence à une *boîte* contenant la valeur de la variable. La boîte en question est un objet à un champ, alloué dans le tas et servant à contenir la valeur d'une variable. Par exemple, la fonction `counted-binop` définie et représentée un peu plus bas capture les variables `binop` et `counter`. Cette dernière est mutable et est conservée dans une boîte.

```
(define counted-binop
  (let ((binop (get-binop))
        (counter 0))
    (lambda (x y)
      (set! counter (+ counter 1))
      (binop x y))))
```



A prime abord, cette représentation peut sembler causer un gaspillage de mémoire important. L'environnement courant est copié à chaque création de fermeture et des boîtes doivent être allouées dans le cas des variables mutables. Mais ce n'est pas exactement ce qui se passe. En effet, il est possible, grâce à des informations obtenues lors de la compilation, de ne copier dans l'environnement de la fermeture que les variables pouvant être utiles. Ces variables sont celles qui sont accédées depuis le corps de la fermeture. Toutes les autres ne peuvent influencer l'évaluation du corps. En quelque sorte, la fermeture, à sa création, sélectionne les variables qui la concerne.

La représentation est décrite dans [FL92].

Comparaison avec la représentation par des blocs chaînés

Même si cette représentation a l'avantage de permettre des accès plus rapide aux variables fermées (capturées lors de la création de la fermeture), le critère primordial reste l'espace. Sur ce point, il est plus difficile de comparer les deux techniques.

Les deux exemples qui suivent montrent que les deux représentations ont leurs avantages. Voyons le premier exemple.

```
(define make-thunk1
  (lambda (a)
    (let* ((b (f1 a))
```

```
        (c (f2 b))
        (d (f3 c)))
    (lambda () (g d)))))
```

La fonction `make-thunk1` sert à retourner des fermetures à aucun paramètre dont le comportement est déterminé par la variable fermée `d`. Les variables `a`, `b`, `c` et `d` sont les variables de l'environnement lors de la création d'une fermeture. Dans la représentation par des blocs de liaison chaînés, l'environnement de définition sauvé consiste en une chaîne de 4 blocs de 1 variable. Dans la représentation plate, seul l'espace pour contenir la variable `d` est requis, puisque les autres variables présentes dans l'environnement ne peuvent influencer l'exécution du corps de la fermeture. Dans cet exemple, la représentation plate est clairement gagnante. C'est grâce à la sélection des variables à conserver. Voyons le deuxième exemple.

```
(define make-thunk2
  (let ((a (f1 1))
        (b (f2 2))
        (c (f3 3)))
    (lambda (d)
      (lambda () (list a b c d)))))
```

La fonction `make-thunk2` sert aussi à retourner des fermetures à aucun paramètre. Mais cette fois-ci, les fermetures ont un comportement qui dépend des 4 variables présentes dans l'environnement de définition. Dans la représentation avec les blocs chaînés, l'environnement sauvé dans les fermetures consiste en un bloc contenant `d` suivi d'un bloc contenant `a`, `b` et `c`. Le second bloc n'est pas alloué à la création de chaque fermeture, il est commun à toutes les fermetures pouvant être créées. Dans la représentation plate, un tableau contenant les 4 variables doit être alloué à la création de chaque fermeture. La représentation par des blocs est gagnante cette fois-ci. C'est grâce au partage des blocs d'activation communs.

Aucune des deux représentations ne peut prétendre être toujours meilleure que l'autre. Il faudrait faire de nombreux tests pour montrer qu'en moyenne, sur de nombreux programmes *représentatifs*, il y a une représentation qui se distingue par rapport à l'autre. Dans la plupart des articles où est développée une représentation, les auteurs porte leur attention presqu'exclusivement sur l'efficacité. Nous n'avons pas entrepris de faire d'étude poussée pour déterminer la meilleure des deux représentations.

3.4.6 Nos choix d'implantation

Les versions interactives de l'interprète utilisent la représentation par des listes d'association, étant donné que cette représentation ne requiert pas de pré-traitement.

Les versions suivantes utilisent la représentation par des blocs de liaison chaînés. Cette représentation peut être utilisée dans tous les contextes, et non pas seulement pour les environnements de définition des fermetures. Il n'y a donc qu'une seule méthode d'accès aux variables à inclure

dans le noyau de l'interprète. De plus, il n'est pas nécessaire d'inclure quelqu'information que ce soit pour la création des fermetures car il n'y a pas de sélection de variables à faire.

Chapitre 4

Recherche d'un GC adéquat

La recherche d'un GC pour notre interprète est une des parties importantes du travail. En effet, nous préférons avoir un GC temps réel pour permettre un déroulement régulier de l'exécution. De plus, il est important que la méthode retenue soit relativement simple puisque nous ne pouvons accepter une implantation du GC trop volumineuse. Aussi, elle doit laisser la chance d'adopter une représentation des objets qui soit compacte.

La prochaine section explique la nécessité d'avoir un GC compactant. Après, nous mentionnons les principaux problèmes liés à la réalisation des GC temps réel. Ensuite, différentes possibilités sont mentionnées et classées. Enfin, nous décrivons en détail la technique que nous avons adoptée en bout de ligne.

4.1 L'importance d'éviter la fragmentation

Nous décrivons tout d'abord ce qu'est la fragmentation. La fragmentation dans un espace mémoire est une répartition des objets qui fait en sorte que l'espace peut contenir moins d'objets que si la répartition était optimale. Elle se produit lorsque l'espace libre est reparti en plusieurs petits morceaux entre des objets alloués et que les morceaux ne sont pas assez gros pour recevoir certains objets. Par exemple, un espace mémoire de 1000 champs, contenant seulement 10 paires placées à intervalles réguliers, peut être incapable d'accepter des vecteurs d'une longueur d'une centaine de champs; alors même qu'il reste plus de 900 champs libres au total. Il est clair que nous ne pouvons tolérer la fragmentation dans le peu d'espace mémoire dont dispose notre interprète.

Il est possible d'éviter les méfaits de la fragmentation en contraignant tous les objets à avoir la même longueur. Dans ce cas, l'espace libre se brise quand même en multiples morceaux avec le temps, mais ce découpage ne nuit pas à la capacité de l'espace mémoire. C'est parce que tous les bouts d'espace libre ont nécessairement une longueur qui est un multiple de la longueur

des objets. Ils peuvent tous éventuellement servir à loger des objets. De cette façon, on n'évite pas un éparpillement des objets dans l'espace mémoire, mais il n'a aucun effet nuisible sur la capacité de ce dernier.

Malheureusement, en Scheme, il n'est pas possible de se conformer à une contrainte aussi forte. Les objets n'ont pas tous la même taille. Et surtout, les objets à taille indéterminée comme les vecteurs et les chaînes de caractères n'ont pas de limite de taille. Il n'est donc pas possible de fixer une taille unique pour tous les objets. Une autre approche pourrait consister à diviser l'espace mémoire en deux sections: une pour les objets à taille fixe et une pour les vecteurs et les chaînes. Il faudrait décréter que tous les objets à taille fixe ont la même taille: celle du plus grand objet. Ce décret entraînerait un gaspillage important d'espace.

Une autre solution à la fragmentation est d'utiliser un GC qui ait une phase de compactage des objets. A chaque cycle, le GC déplace certains ou tous les objets afin de les regrouper en un seul bloc, laissant tout l'espace libre en un autre bloc. Nous nous intéressons donc aux GC compactants.

4.2 Les difficultés liées aux GC temps réel

Il y a plusieurs difficultés liées à l'utilisation d'un GC temps réel. Les difficultés que nous mentionnons ici sont les incohérences créées par le GC durant son cycle, les mutations effectuées par l'application, le traitement des grands objets et la vitesse de travail nécessaire pour récupérer de l'espace libre avant qu'il n'en manque.

Dans ce chapitre, lorsque nous parlons de l'application, nous excluons normalement le GC. L'application et le GC peuvent être vus comme deux programmes en exécution entrelacée.

4.2.1 L'incohérence des données

Le GC effectue plusieurs opérations sur les données présentes lorsqu'il effectue son cycle de collecte. Une application dotée d'un GC bloquant ne peut tomber sur des données dans un état incohérent. Mais lorsque le GC est temps réel, l'application continue à fonctionner pendant que le GC remanie les objets dans l'espace mémoire. Les données peuvent être dans un état incohérent pour plusieurs raisons. Les objets sont normalement marqués d'une façon ou d'une autre. Ils sont déplacés lorsque le GC utilisé déplace les objets, ce qui est toujours le cas pour un GC compactant. La nouvelle position suite à leur déplacement doit être indiquée d'une façon ou d'une autre.

Il est normalement simple de faire en sorte que les opérations de lecture et d'écriture fonctionnent bien malgré que des bits de marquage soient allumés. Un problème plus difficile est de s'assurer que tout objet soit rapidement accessible même s'il vient d'être déplacé. En effet, il est normal d'avoir plusieurs références à un même objet. Lorsque cet objet se fait déplacer, il

n'est généralement pas possible de mettre à jour toutes les références à l'objet immédiatement. Ceci vient du fait que les périodes durant lesquelles le GC peut travailler sont habituellement courtes. Celui-ci ne peut se permettre de traverser tout le tas pour mettre à jour les références à chaque fois qu'un objet se fait déplacer. Il est donc vital de prévoir un mécanisme qui permette au GC d'avertir l'application qu'un objet vient de changer de position.

4.2.2 Les mutations faites par l'application

Les GC temps réel ne ramassent pas toujours les objets qui meurent durant le cycle même. En effet, un objet peut être rejoint très tôt durant le cycle du GC. Il est alors marqué et évite d'être collecté durant ce cycle. Mais il peut devenir inaccessible avant même la fin du cycle. Ce type de situation fait en sorte que des objets morts survivent au cycle du GC. Toutefois, il sont assurés d'être ramassés au cycle suivant. Ce n'est donc pas un problème sérieux. Les GC temps réel ont normalement tendance à conserver une partie des objets morts un cycle de plus que ne l'aurait fait un GC bloquant. Ce comportement est appellé le conservatisme.

Le problème inverse quant à lui doit être évité à tout prix: un GC ramassant un objet qui était encore vivant. Ce problème semble tellement grossier qu'aucun concepteur de GC ne devrait commettre cette erreur. Pourtant, dans le cas d'un GC temps réel, il n'est pas trivial de s'assurer de ne pas commettre l'erreur. Le problème vient du fait que l'application continue à exécuter durant le cycle du GC. L'application peut faire des mutations dans les objets et *tenter* de cacher un objet vivant aux yeux du GC.

Voici comment cela peut se produire. Supposons que le GC est en train de rechercher les objets vivants, que la paire **A** n'a pas encore été rejointe et que la paire **B** a été rejointe, fouillée et ne sera plus fouillée à nouveau. Nous appelons "fouiller" l'action d'examiner des champs à la recherche d'objets non marqués. Ensuite, l'application mute la paire **B** afin qu'elle contienne la paire **A** et efface *toute autre* référence à cette dernière. Il en résulte que la paire **A** est vivante mais elle ne sera pas marquée et sera donc ramassée d'ici la fin du cycle.

Il existe plusieurs moyens de prévenir ce problème. On utilise ce qu'on appelle des barrières. Il existe des barrières en lecture et des barrières en écriture. Il est possible d'utiliser des barrières d'une ou des deux sortes dans une application. Les barrières en lecture ont comme effet d'empêcher l'application de manipuler directement des objets qui n'ont pas été rejoints. Les objets manipulés sont rejoints sur-le-champ ou prochainement s'ils ne l'étaient pas encore. Les barrières en écriture ont comme effet de déclarer au GC toute mutation effectuée afin que celui-ci puisse en tenir compte. Il existe de nombreuses barrières en écriture. Nous n'entrons pas dans les détails ici.

4.2.3 Les grands objets

En Scheme, les vecteurs et les chaînes de caractères sont des objets qui peuvent être arbitrairement longs. Lorsque le GC fouille ou déplace un long objet, il doit pouvoir suspendre temporairement ses activités pour permettre à l'application de continuer. Le déplacement, surtout, est une tâche délicate. Un objet à moitié déplacé doit être manipulable par l'application malgré tout. Le GC doit être en mesure d'avertir l'application qu'un objet est présentement coupé en deux et il doit laisser des informations suffisantes pour permettre de retrouver les deux bouts.

4.2.4 Réglage de la vitesse du GC

Ce problème est crucial. Il faut trouver un équilibre dans le réglage de la vitesse de travail du GC. Un GC trop rapide consomme du temps de processeur inutilement. Un GC trop lent risque de ne pas fournir d'espace libre à temps pour permettre la poursuite de l'exécution.

4.3 Les différentes approches

La plupart des techniques de GC temps réel que nous avons trouvées dans la littérature sont soit des GC à deux semi-espaces, soit des GC qui nécessitent que les objets aient tous la même taille, et parfois même elles imposent ces deux contraintes à la fois.

Nous abordons différentes approches en les classant par catégories et nous discutons de leur intérêt pour notre interprète. En ce qui concerne les approches sur lesquelles il n'y a rien ou presque, nous tentons tout de même d'évaluer leur intérêt. Nous ne prétendons pas faire la preuve que nous évaluons parfaitement les différentes possibilités, mais nous essayons d'apporter des arguments raisonnables pour ou contre ces possibilités.

Nous commençons par les approches où les objets sont regroupés en zones ou en pages d'objets en fonction du type. Ensuite, les autres approches décrites sont celles où les objets de tous les types sont ensemble dans une ou deux grandes zones. Les GC à deux semi-espaces sont décrits avant les GC à un seul espace.

4.3.1 Regroupement en zones ou en pages

Lors de notre recherche dans la littérature, nous n'avons pas trouvé de description de système utilisant une de ces organisations mémoire tout en étant doté d'un GC temps réel. Nous commentons tout de même quelque peu sur ces approches.

Nous abordons brièvement ces organisations de la mémoire dans la section sur la représentation des types (section 3.1.3). Il importe d'en donner un bref rappel. Le regroupement en

zones consiste à séparer le tas en autant de zones qu'il y a de types d'objets. La zone contenant un objet détermine son type. On peut soit séparer le tas de façon définitive ou bien permettre que les frontières entre les zones glissent afin de s'ajuster aux fluctuations des taux d'utilisation des différents types. Le regroupement en pages consiste à séparer le tas en pages de taille prédéfinie. Chaque page peut être libre ou bien comporter des objets d'un seul type. C'est la page qui contient un objet qui détermine son type.

Nous avons vu que ces organisations ont un certain avantage du point de vue de la compacité. Dans le cas du regroupement en zones, aucune mention du type n'est nécessaire sur les objets. Dans le cas du regroupement par pages, le type ne doit être indiqué que sur chaque page. Comme les pages ont toutes la même taille, on retrouve facilement le début de la page contenant un objet et le type qui lui est associé.

Malheureusement, il ne semble pas évident d'utiliser ces organisations mémoire en conjonction avec un éventuel GC temps réel tout en gardant les choses raisonnablement simples.

Même sans utiliser de GC temps réel, la technique de regroupement en pages ne peut gérer adéquatement les objets à longueur variable de Scheme. En effet, il est tout à fait possible que l'on veuille allouer un vecteur ou une chaîne plus long que la taille des pages. Il faudrait donc prévoir une zone à part pour ces objets. Dans les moments où les longs objets seraient intensément utilisés, il faudrait faire disparaître des pages afin de libérer de l'espace. Dans les situations inverses, il faudrait s'assurer que les longs objets soient périodiquement compactés à un bout du tas pour permettre la création de nouvelles pages. Il faut noter que la partie du GC traitant ce cas est déjà aussi complexe qu'un GC compactant à un espace! Une fois que l'on veut un GC temps réel, les choses se compliquent encore plus. Nous abordons plus loin les difficultés qui émergent lorsqu'il faut effectuer un compactage en temps réel à l'intérieur même d'un seul espace.

L'organisation mémoire par pages demande, dans un petit tas comme celui dont dispose l'interprète, que les objets soient regroupés dans le plus petit nombre de pages possible, ou presque. En effet, s'il n'y a pas de regroupement de ce genre qui est effectué, un type abondant auparavant risque de monopoliser toutes les pages. Par exemple, s'il y a beaucoup de paires à un moment donné et que la plupart sont ensuite libérées, il peut en rester une ou deux par pages et ceci demande la conservation de toutes les pages pour ces quelques paires. Le fait de déplacer les objets d'une page à l'autre pose le problème de la mise à jour des références. De plus, il faut pouvoir déplacer les pages elles-mêmes afin de libérer de l'espace pour la zone des longs objets.

L'organisation par zones semble plus simple à aborder. Son problème principal est lié à la difficulté à s'adapter aux changements des taux d'utilisation des différents types. On peut envisager qu'à la fin de chaque cycle, le GC laisse chaque zone avec une taille proportionnelle à l'utilisation de son type. Ainsi, chaque zone aurait le même pourcentage d'espace libre. Mais si un type resté peu utilisé jusque là devient pendant une période le seul type alloué, le GC doit se mettre à travailler à un rythme terriblement élevé. En effet, un type peu utilisé prend peu d'espace, donc sa zone est minuscule, et son espace libre l'est aussi.

Si, autrement, on décide que le GC laisse chaque zone avec la même quantité d'espace libre, celui-ci n'a pas à se mettre à travailler à un rythme fou à la suite d'un malheur. Le problème est qu'il y a normalement des types plus utilisés que d'autres (comme les paires) et que le GC doit travailler rapidement parce les zones associées à ces types se remplissent vite.

Ou encore, on pourrait guider le comportement du GC avec des observations savantes utilisant les probabilités et visant à réduire l'espérance de sa vitesse de travail. Or, il reste que le GC doit s'assurer de travailler assez vite en prévision des *pires* situations possibles. Dans ce cas, le GC doit finir chaque cycle avant qu'*aucune* des zones n'ait pu manquer d'espace libre. Un tel GC doit donc travailler beaucoup plus vite que si l'espace libre était mis en commun. Même si l'efficacité n'est pas le facteur important dans nos travaux, elle n'est à négliger complètement.

Outre le problème de son inefficacité, un GC dans une organisation par zones se devrait de déplacer les objets dans le tas. Ceci reste un problème ardu, comme nous en discutons dans la section sur les GC à un seul espace. Cet aspect combiné à la recherche d'une façon acceptable de guider la vitesse du GC nous laisse supposer qu'un éventuel GC temps réel pour cette organisation mémoire est probablement fort complexe.

Nous n'avons donc pas trouvé par nous-mêmes de technique raisonnablement simple pouvant être utilisée avec une de ces deux organisations par regroupement.

4.3.2 GC à deux semi-espaces

Lorsqu'un GC à deux semi-espaces est utilisé, le tas est divisé en deux moitiés. C'est ce qui explique le nom du GC. Il existe de nombreuses variantes de ce GC, autant comme GC bloquants que comme GC temps réel.

Nous commençons par la description sommaire d'une version bloquante. Nous continuons avec une version temps réel. Nous nous intéressons particulièrement au GC de Brooks ([Bro84]).

Version bloquante

Lorsque le GC ne travaille pas, un seul des semi-espaces est en utilisation. Celui qui est utilisé contient les objets du système et une certaine quantité d'espace libre. L'allocation des nouveaux objets se fait aussi dans ce semi-espace. Lorsqu'il ne reste plus assez d'espace libre pour permettre une allocation, le GC est déclenché.

Le GC copie les objets vivants (ou accessibles) du semi-espace utilisé à l'autre. Nous appellons **FROM** le semi-espace en cours d'utilisation et **TO** le semi-espace qui s'apprête à recevoir les objets vivants. Les objets copiés sont placés à la queue-leu-leu dans **TO**. Ainsi, lorsque le GC a terminé son travail, le rôle des semi-espaces est interchangé et le nouveau semi-espace utilisé contient un bloc compact d'objet suivi d'espace libre. Les GC à deux semi-espaces effectuent un compactage et préviennent la formation de fragmentation.

La recherche des objets vivants se fait à partir des racines de l'application. Les racines d'une application sont les références toujours présentes. Par exemple, dans le cas de notre interprète, les racines comprennent les quelques registres de la machine virtuelle et les variables globales du programme. Tout objet référencé depuis les racines est vivant et est copié dans **TO**. Tout objet référencé par un objet vivant est copié aussi.

Lorsqu'un objet est copié, les étapes suivantes sont faites: ses champs sont copiés dans **TO**; il est marqué; un de ses champs est utilisé pour indiquer la position de la nouvelle version de l'objet. Il est important de pouvoir marquer un objet, afin d'éviter de le copier plus d'une fois. De plus, lorsque les objets copiés sont fouillés à la recherche d'autres objets, leurs champs sont mis à jour afin que les références reflètent la nouvelle position des objets. Les racines aussi voient leurs références mises à jour lorsqu'elles sont fouillées.

Il existe de nombreuses façons de parcourir les racines, de fouiller les champs des objets, de marquer un objet et d'indiquer la position de la nouvelle version d'un objet. Nous n'entrons pas dans les détails de chaque méthode existante. Nous nous contentons de mentionner que l'algorithme de Cheney ([Che70]) est un algorithme simple, représentatif et qu'il est utilisé couramment.

Version temps réel: GC de Brooks

Pour obtenir une version temps réel d'un GC à deux semi-espaces, il ne suffit pas que le GC et l'application s'échangent le contrôle régulièrement. En effet, durant un cycle, le GC modifie considérablement l'apparence des semi-espaces et l'application continue à faire des mutations sur les objets. Ces difficultés sont déjà exposées plus haut.

Afin qu'il puisse être temps réel, le GC ne doit pas être déclenché seulement lorsqu'il ne reste plus d'espace libre dans le semi-espace actif. Il doit être déclenché longtemps à l'avance, ou même être continuellement en activité.

Plus un GC est déclenché à l'avance, plus il risque d'être conservateur. Un GC conservateur est un GC qui conserve des objets qui sont morts avant la fin du cycle. Ces objets auraient été ramassés par un GC bloquant. Le fait d'être conservateur est normalement une conséquence du comportement temps réel. Le conservatisme a un impact sur la quantité d'objets (vivants et morts) présents dans le tas et donc sur la quantité de travail qu'il faut faire à chaque cycle.

Toutefois, un GC déclenché plus longtemps à l'avance peut découper le travail à faire en plus petits morceaux, assurant des bornes plus serrées sur la durée des opérations de l'application. La façon optimale de régler le rythme de travail du GC dépend donc de ce que l'on privilégie entre la recherche d'efficacité et la minimisation de la durée des bloquages dûs au GC. La discussion des derniers paragraphes tient en fait pour tous les GC temps réel.

Etant donné que le GC déplace les objets de **FROM** à **TO** pendant que l'application s'exécute, il faut s'assurer que cette dernière soit capable de les retrouver rapidement. La méthode que nous décrivons ici fait partie de la technique de Brooks ([Bro84]). Il s'agit d'ajouter à

tout objet un champ supplémentaire servant à pointer sur l'objet lui-même. Plus exactement, le champ supplémentaire est juste devant l'objet et pointe sur le premier de ses champs. Un objet ne détient pas des références directes aux autres objets, mais des références à leur champ supplémentaire. Les accès à un objet doivent toujours passer par son champ supplémentaire.

L'intérêt de ce champ est de permettre au GC de copier les objets de **FROM** à **TO** sans que l'application ne puisse voir la différence. Ce que le GC a à faire est: créer la nouvelle version de l'objet dans **TO**; faire pointer le champ supplémentaire de la vieille version vers le premier champ de la nouvelle version. Ainsi, que l'application accède à l'objet via une référence mise à jour ou via une vieille référence, elle aboutit au même objet grâce à la double indirection. Eventuellement, le GC parvient à copier tous les objets vivants et mettre à jour toutes les références. Le semi-espace **TO** devient le semi-espace actif et ce, jusqu'à ce qu'un nouveau cycle commence.

Le champ supplémentaire permet du même coup d'indiquer le marquage. Un objet marqué est un objet dont le champ supplémentaire ne pointe pas sur le champ suivant. Lorsqu'un objet est marqué, c'est son champ supplémentaire qui indique la position de sa nouvelle version.

Nous arrêtons là les explications sur les GC temps réel à deux semi-espaces. Nous avons insisté sur l'idée du champ supplémentaire car il nous a servi d'inspiration pour la création de la technique que nous décrivons plus loin.

Intérêt des GC à deux semi-espaces

Parmi les avantages des GC à deux semi-espaces, nous pouvons mentionner le compactage, leur relative simplicité et leur efficacité. Ils sont probablement parmi les plus efficaces, du moins, parmi les GC qui déplacent les objets. Il existe des GC passablement plus efficaces, mais ils laissent les objets sur place, permettant à la fragmentation de s'installer.

Les GC à deux semi-espaces ont le désavantage de demander deux semi-espaces et de ne pouvoir stocker des objets que dans un seul à la fois (ou dans l'équivalent d'un seul, dans le cas des GC temps réel). D'une certaine façon, chaque objet prend deux fois l'espace qu'il a l'air d'occuper.

Ce problème n'existe pas avec les GC travaillant sur un seul espace. Une technique à un espace qui demande d'augmenter légèrement la taille des objets par rapport à une technique à deux semi-espaces permet malgré tout d'atteindre une plus grande compacité.

Un GC temps réel à deux semi-espaces est implanté dans une des versions de notre interprète. Il ne s'agit pas exactement de la technique de Brooks. Le marquage et le lien à la nouvelle version d'un objet sont faits différemment. Dans cette version, le typage est fait de façon uniforme, donc les objets disposent tous d'un champ de type et nous utilisons ce champ pour faire le marquage. L'application doit vérifier à chaque accès si l'objet est une vieille version. Si c'est le cas, le lien à la nouvelle version se trouve dans le premier champ "ordinaire".

Cette version de l'interprète nous a permis de nous familiariser avec le contrôle du rythme de travail du GC. Cette tâche étant plus simple dans un GC à deux semi-espaces que dans le GC que nous avons adopté par la suite. Nous n'avons malheureusement pas de comparaisons empiriques sur la consommation de mémoire des deux techniques. La représentation des types par taggage est adoptée entre ces versions.

4.3.3 GC à un seul espace

Parmi les GC à un seul espace, il y a deux grandes familles: les GC *mark & sweep* et les GC compactants (*mark & compact*).

Les GC *mark & sweep* fonctionnent en identifiant tous les objets vivants et en convertissant les objets morts en espace libre. Les objets ayant survécus ne sont pas déplacés. Ce type de GC n'élimine donc pas la fragmentation. Ils sont peu intéressants dans le cadre de l'interprète.

Les GC compactants, quant à eux, identifient les objets vivants et les regroupent à un bout du tas. Il existe diverses stratégies pour effectuer le regroupement. Des GC déplacent certains des objets vers les trous dans les adresses plus basses. Dans ce cas, le regroupement n'est pas nécessairement parfaitement compact et peut avoir un comportement en pire cas qui est assez grave. Pour être acceptable, une telle technique devrait pouvoir garantir une perte d'espace bornée et assez réduite.

D'autres GC s'assurent que les adresses basses sont complètement occupées. La méthode qui est souvent utilisée pour parvenir à cette fin consiste à faire glisser les objets vivants vers les adresses basses. Il s'agit de rechercher linéairement les objets marqués et de les déplacer à la suite du précédent objet vivant.

Les GC compactants pour lesquels nous avons une description sont tous des GC bloquants. Ils consistent en une phase de marquage où les objets accessibles sont tous marqués, suivie par une phase où le déplacement des objets et la mise à jour des références sont effectués de façon plus ou moins conjointe.

La mise à jour des références peut se faire de différentes façons. Par exemple, l'algorithme de Morris ([Mor78]) fait deux passes dans les objets. Dans la première passe, les références pointant vers des adresses plus basses sont mises à jour. Dans la seconde, les références pointant vers des adresses plus hautes sont mises à jour et les objets sont déplacés vers leur nouvelle position. L'idée consiste à chaîner ensemble les objets pointant vers chaque objet. Une fois parvenu à un objet, sa position finale peut être calculée et tous les objets le pointant sont mis à jour.

La réalisation d'un GC temps réel compactant pose des problèmes plus sérieux quant à la mise à jour des pointeurs. Lorsqu'un objet est déplacé, on sait qu'il n'est pas possible de mettre à jour toutes les références le désignant dans un temps raisonnablement court. Il est donc inévitable que l'application reprenne le contrôle de l'exécution avant que toutes les références à l'objet soient mises à jour. Il devient nécessaire de laisser une information quelconque permettant à

l'application de retrouver la nouvelle version de l'objet.

Une des premières idées qui peuvent venir à l'esprit consiste à dire qu'un des champs de la vieille version de l'objet devrait servir à pointer vers la nouvelle version. De cette façon, lors d'un accès à un objet, l'application vérifie s'il a été déplacé et, si oui, suit le pointeur à la nouvelle version. Comme il peut rester des liens à la vieille version jusqu'à la fin du cycle du GC, celle-ci doit rester en place jusqu'à la fin. Il y a alors un problème: la vieille version risque de se trouver sur un emplacement où la nouvelle version d'un autre objet doit être installée.

Une des façons de régler ce problème consiste à faire en sorte que les nouvelles versions ne puissent être installées aux mêmes emplacements que les vieilles versions. On peut donc programmer le GC pour qu'il compacte vers les adresses basses et hautes alternativement. Il faut alors s'assurer que les objets déplacés n'empiètent pas sur la partie des objets possiblement à déplacer. Finalement, cette opération revient à faire une division du tas en deux semi-espaces.

Nous avons développé une autre technique pour parvenir à réaliser un GC temps réel compactant à un espace. Elle est inspirée de la technique de Brooks en ce qui concerne l'utilisation systématique d'un champ supplémentaire pour accéder à un objet. Une modification de ce champ fait implicitement suivre toutes les références à cet objet. Dans notre technique, ces champs sont tous regroupés à un bout du tas. De plus, un tel champ est assigné à un objet pour toute sa durée de vie.

La technique que nous présentons ici a été publiée dans une conférence (voir [DFS96]). Nous reprenons les explications presque complètement dans le reste du chapitre.

Nous commençons par décrire une version non temps réel de la technique. Ensuite, nous décrivons les ajouts à y faire pour la rendre temps réel. Les aspects du traitement des longs objets et du réglage de la vitesse du GC, que nous n'avons qu'effleurés jusqu'ici, y sont abordés en profondeur. Cette technique est plus compacte que toutes celles que nous avons trouvées dans la littérature et elle est utilisée dans les dernières versions de l'interprète.

4.4 La version bloquante de notre technique de GC

Avant de décrire de quelle façon le GC récupère de l'espace dans le tas, il importe de présenter l'organisation du tas. Ensuite, les conditions nécessaires au déclenchement du GC ainsi que la technique de récupération sont décrites.

4.4.1 Disposition du tas

Nous supposons au départ que le tas est divisé en deux régions: celle des identificateurs et celle des objets et de la pile de marquage. La taille des régions ne doit pas être choisie arbitrairement, comme on le verra un peu plus loin.

Un identificateur sert de pointeur vers un objet dans la deuxième région. Généralement, les identificateurs ne sont pas tous utilisés. Les identificateurs inutilisés sont chaînés en une liste d'identificateurs libres. L'identificateur assigné à un objet demeure le même durant toute la vie de l'objet. La région des identificateurs n'est pas compactée. Mais comme tous les identificateurs ont la même taille, ceci ne cause pas de fragmentation, mais simplement une dispersion des identificateurs vivants.

La région des objets et de la pile de marquage ne comporte pas de séparation fixe entre ses deux parties. Les objets sont placés un à la suite de l'autre à partir des adresses basses. L'espace réservé pour la pile grossit à partir des adresses hautes. Chaque objet a un champ supplémentaire qui sert à contenir l'adresse de son identificateur. Nous appelons ce champ le pointeur arrière. On verra à la section 4.4.3 l'utilité de la pile de marquage.

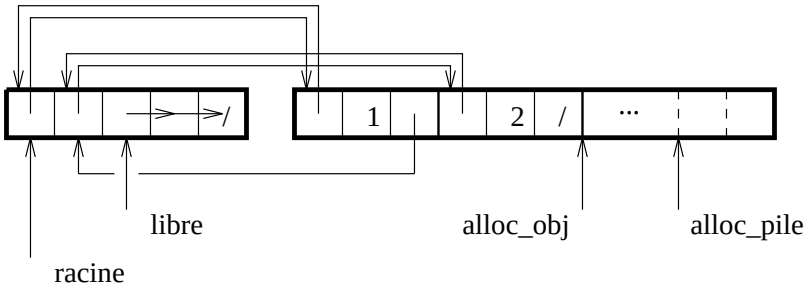
Lors de l'allocation d'un objet, un identificateur est réservé dans la première région, l'espace nécessaire est réservé dans la deuxième région, les champs de l'objet sont initialisés, l'identificateur reçoit l'adresse du nouvel objet et l'adresse de l'identificateur est retournée à l'application. L'espace réservé dans la deuxième région est constitué des champs de l'objet, incluant son pointeur arrière, et d'une case mémoire pour la pile. L'application doit toujours accéder aux champs de l'objet via l'identificateur.

La pile n'est utilisée que durant la phase de marquage, mais un espace à son effet est réservé en permanence. Nous jugeons qu'il est préférable que le GC gère lui-même une pile pour le marquage. Premièrement, la pile d'exécution de l'application risque de ne pas être assez grande pour permettre de marquer récursivement tous les objets d'une structure de données très profonde. Deuxièmement, les informations stockées par le GC dans sa propre pile sont beaucoup plus compactes que celles stockées dans la pile d'exécution.

La figure 4.1 montre un exemple où le tas ne contient que deux paires. Celles-ci forment la liste `(1 2)`. On peut remarquer que le premier champ de chaque paire est le pointeur arrière vers leur identificateur respectif. A l'extrème droite du tas se trouvent les deux cases réservées pour la pile. On peut voir aussi que les identificateurs qui ne sont pas utilisés forment une liste d'identificateurs libres. A chaque allocation, les pointeurs d'allocation et de pile se rapprochent l'un de l'autre.

4.4.2 Déclenchement du GC

Le GC doit être déclenché lorsque l'application demande l'allocation d'un objet pour lequel il ne reste plus assez d'espace. Il n'est pas nécessaire de vérifier s'il reste un identificateur libre si l'on fixe une taille suffisamment grande pour la région des identificateurs. En effet, supposons que le plus petit objet alloué n'ait qu'un champ. Alors, lors de l'allocation d'un tel objet, on réserve un identificateur, un pointeur arrière, le champ et une case pour la pile. Ce type d'objet étant le plus petit, on sait qu'au maximum, le quart du tas sera occupé par des identificateurs. Donc, en prenant la région des identificateurs trois fois plus petite que celle des objets, on élimine ce test. Il ne reste que le test pour vérifier s'il y a assez d'espace pour l'objet dans la deuxième



Légende: racine: une racine de l'application
libre: la tête de la liste des identificateurs libres
alloc_obj: le pointeur d'allocation des objets
alloc_pile: début de la zone réservée pour la pile

FIG. 4.1 - *Illustration du tas contenant la liste* `(1 2)`

région.

Dans la version finale de l'interprète, nous nous sommes assurés que tous les objets qui sont alloués ont au moins deux champs. Pour parvenir à ce minimum de champs, nous ajoutons un champ artificiel aux vecteurs et aux chaînes vides. Ces objets ne sont pas nécessairement très utilisés, donc l'espace perdu est minime en moyenne. Toutefois, ceci nous permet de réduire la région des identificateurs à un cinquième du tas.

4.4.3 Description de l'algorithme

Le marquage

La première phase du GC est le marquage. L'algorithme commence par marquer tous les objets accessibles directement des racines. Le fait de marquer un objet cause son ajout dans la pile de marquage. Les objets qui sont sur la pile sont des objets dont on dira qu'ils n'ont pas encore été fouillés à la recherche de références à des objets non encore marqués. Par la suite, l'algorithme extrait un objet de la pile, le fouille et marque tous les objets nouvellement atteints. Le processus de marquage se poursuit tant que la pile n'est pas vide.

Connaissant la façon de marquer, on peut comprendre pourquoi nous avons choisi de réserver une case de pile par objet alloué. En effet, il est possible que toute la pile sauf une case soit utilisée. Par exemple, un vecteur qui contient des références sur tous les autres objets du tas est une structure de données qui fait en sorte que la pile est utilisée presque complètement.

Nous utilisons une astuce simple et relativement portable pour éviter de réserver de l'espace supplémentaire pour le bit de marquage. Il est raisonnable de supposer que les mots de la machine sont à des adresses paires, en particulier les identificateurs. On peut donc utiliser le bit le moins significatif du pointeur arrière comme bit de marquage. Il serait également possible d'utiliser le bit le moins significatif de l'identificateur mais cela serait moins efficace pour les deux raisons

suivantes: lors du compactage, le test du bit de marque demanderait une référence de plus à la mémoire; les accès ordinaires aux champs de l'objet demanderaient de masquer le dernier bit de l'identificateur.

Le compactage

La deuxième phase du GC est le compactage. Les objets marqués sont regroupés au début du tas comme s'ils avaient tous glissé vers les adresses basses. Leur ordre est donc préservé. Deux pointeurs partent du début de la région des objets. Le premier est le pointeur de source et sert à rechercher les objets marqués. Le second est le pointeur de destination et indique la nouvelle position de chaque objet marqué. Si l'objet pointé par le pointeur source est marqué, il est déplacé à l'endroit pointé par le pointeur de destination. De plus, son identificateur est mis à jour pour qu'il pointe vers la nouvelle position de l'objet. Bien sûr, l'identificateur de l'objet est retrouvé rapidement grâce au pointeur arrière. Si l'objet n'est pas marqué, son identificateur est libéré et ajouté à la liste des identificateurs libres. Durant le compactage, les objets vivants sont comptés afin de pouvoir réserver la bonne taille de pile pour le prochain cycle de GC et de mesurer correctement la quantité de mémoire libre.

4.5 La technique temps réel

Nous décrivons tout d'abord les barrières en lecture et en écriture qui doivent être ajoutées à la technique de base; ensuite, nous montrons de quelle façon l'algorithme du GC est découpé; nous parlons de l'aspect synchronisation entre le GC et l'application; enfin, nous décrivons trois améliorations pouvant être apportées au GC.

Il y a certaines contraintes qui doivent être imposées à l'application en plus des barrières pour que cette technique, telle nous allons la décrire, puisse fonctionner. Premièrement, le nombre de racines de l'application doit être assez petit. Deuxièmement, l'application ne doit pas tenter de conserver plus qu'une certaine fraction du tas en objets vivants. Cette dernière contrainte est commune à toutes les techniques de GC temps réel, à l'exception du comptage de références.

4.5.1 Les barrières en lecture et en écriture

L'indirection via l'identificateur

La barrière la plus importante est déjà incluse dans la technique de base, c'est l'utilisation des identificateurs. En effet, ces identificateurs permettent au GC de déplacer un objet et de faire suivre "toutes" les références à cet objet en une seule affectation, donc de façon atomique.

Barrière en écriture

Nous utilisons une barrière en écriture durant le marquage pour éviter que le GC ne collecte des objets accessibles. A l'instant où un objet A non marqué est stocké dans un objet B déjà marqué, A est lui-même marqué. Cette barrière en écriture est similaire à celle proposée dans [DLM+78], et est appelée barrière en écriture à mise à jour incrémentielle (*incremental update write barrier*). Cette barrière est moins conservatrice qu'une barrière en lecture ([Bak78, Bro84, Nil88, Bak92, NS92]), qui ne laisse accéder l'application à aucun objet non marqué, ou qu'une barrière en écriture de type *snapshot-at-beginning* ([Yua90]), qui conserve tout objet ayant été accessible au début du cycle de GC. Il y a une autre barrière en écriture de type *incremental update* qui est décrite dans [Ste75]. L'action de cette barrière consiste à noter que l'objet déjà fouillé qui vient de recevoir l'objet non marqué devra être fouillé à nouveau ultérieurement. Cette barrière est encore moins conservatrice que celle que nous avons adoptée ([Wil92]), mais elle est plus difficile à implanter.

Barrière pour les longs objets

Il y a une dernière barrière en lecture et en écriture qui doit être en place pour les "longs objets". La définition de "long objet" peut varier d'une implantation à l'autre. Dans notre interprète, ce sont les vecteurs et les chaînes de caractères qui sont considérés comme de longs objets. Lorsqu'un long objet est en cours de copie, le début de l'objet se trouve à la nouvelle position et le reste se trouve à la position originale. Durant un tel copiage, le GC "déclare" l'identificateur du long objet, c'est-à-dire que l'identificateur est mis dans une variable globale. Ainsi, l'application peut vérifier quel objet le GC est en train de déplacer. La barrière consiste simplement à comparer l'identificateur de l'objet accédé à celui qui est déclaré. En cas d'égalité, il faut vérifier dans laquelle des deux parties se trouve le champ qui doit être accédé.

4.5.2 L'algorithme du GC

La banque de temps

Avant de décrire l'algorithme du GC, il est important de parler de la *banque de temps* du GC. L'idée consiste à maintenir un compteur qui indique la quantité de travail que peut effectuer le GC. A chaque allocation, une certaine quantité de temps est ajoutée à la banque, le GC travaille tant que sa banque est positive et ensuite l'objet demandé est réellement alloué. Nous verrons dans la prochaine sous-section comment le temps est géré lors des allocations et lors du travail du GC.

Les opérations avec un astérisque sont des opérations qui coûtent du temps au GC, c'est-à-dire qui enlèvent du temps de sa banque.

L'algorithme proprement dit

L'algorithme est décrit à la façon d'un automate fini. C'est à chaque "transition" qu'un test est fait pour vérifier que la banque de temps n'est pas épuisée. L'état initial est l'état 1.

1. Préparer le cycle du GC. Déclarer à l'application que le GC est en phase de marquage. Aller à 4.
2. S'il n'y a pas d'objet sur la pile de marquage, aller à 4. Si l'objet à fouiller est long, aller à 3. Sinon, fouiller* l'objet et aller à 2.
3. Fouiller* autant de champs que possible dans l'objet. S'il a été fouillé au complet, aller à 2. Sinon, aller à 3.
4. Marquer les objets directement accessibles depuis les racines. Si la pile est vide, aller à 5. Sinon, aller à 2.
5. Fin du marquage. Début du compactage. Aller à 6.
6. S'il n'y a plus d'objets à déplacer, aller à 9. Sinon, aller à 7.
7. Si l'objet n'est pas marqué, libérer* son identificateur, sauter par-dessus l'objet et aller à 6. Si l'objet est petit, le déplacer*, changer la valeur de son identificateur et aller à 6. Sinon, changer* la valeur de son identificateur, déclarer que l'objet est en déplacement et aller à 8.
8. Déplacer* autant de cases de l'objet que possible. Mettre à jour le point de coupure déclaré à l'application. Si l'objet est complètement déplacé, cesser de le déclarer coupé et aller à 6. Sinon, aller à 8.
9. Clore le cycle du GC. Aller à 1.

Observations

L'algorithme nécessite en plus un test spécial lorsque le tas est vide. En effet, il ne coûte aucun temps au GC pour faire un cycle sur un tas vide. Un moyen simple d'éviter le test consiste à introduire un objet toujours vivant dans le tas dès son initialisation.

L'état qui fouille les racines peut être atteint plus d'une fois par cycle. Ceci est nécessaire à cause du fait que des allocations et des changements dans les racines se font lors des pauses du marquage. De plus, il est nécessaire de fouiller toutes les racines d'un coup. Sinon, il n'est pas possible de savoir s'il n'y a vraiment plus d'objets à marquer. Ceci explique pourquoi il faut que l'application n'ait qu'un petit nombre de racines. Malgré les allocations faites par l'application durant le marquage, le rythme de marquage du GC est réglé de façon à assurer la terminaison du marquage.

Il est possible d'éliminer la limite sur le nombre de racines. Nous en parlerons dans les améliorations qui peuvent être apportées à la technique.

Nous avons préféré que cette fouille des racines soit atomique et ne coûte aucun temps au GC. Nous aurions pu considérer que cette opération avait un coût. Auquel cas, nous aurions un contrôle plus serré sur les bornes de temps d'exécution, mais aussi l'analyse du rythme de travail du GC serait plus complexe (pour l'analyse, voir plus loin).

Pour ce qui est du compactage, les modifications dans la version temps réel sont assez simples. Là encore, le rythme de compactage est suffisamment élevé pour permettre au pointeur de source de rejoindre le pointeur d'allocation. A la fin du compactage, le pointeur d'allocation est modifié pour indiquer la même position que le pointeur de destination.

Afin que le GC ne conserve pas trop d'objets inaccessibles, nous avons choisi d'allouer les objets non marqués pendant le marquage. Ceci laisse une chance aux objets de courte durée de devenir inaccessibles *avant* que le GC ne les marque. Par exemple, un groupe d'objets fraîchement alloués peut être détenu temporairement par une racine et abandonné ensuite. Si les racines ne sont pas fouillées durant ce temps, ces objets risquent de ne jamais être atteints par le marquage. Durant le compactage, toutefois, les objets doivent être alloués marqués, étant donné que le GC ne peut avoir aucune preuve qu'il sont devenus inaccessibles avant d'avoir fait un nouveau marquage.

4.5.3 Synchronisation du GC avec l'application

Contrainte sur la quantité d'objets vivants

A tout moment, l'application ne doit pas tenter de conserver plus qu'une certaine fraction du tas en objets accessibles. En fait, plus le tas contient d'objets accessibles, plus le temps en pire cas pris par les opérations primitives est grand. Si on calcule pour l'application le pourcentage d'occupation maximale du tas en objets alloués accessibles, on peut alors calculer les bornes de temps sur chaque opération primitive.

Comptabilité du temps du GC

La façon de compter le temps lors des allocations et du travail du GC a une grande importance pour assurer un bon fonctionnement du GC. La vitesse du GC est réglée par un paramètre que nous appelons le *ratio*. A chaque allocation, nous ajoutons à la banque de temps du GC le produit entre le ratio et l'espace demandé pour le stockage de l'objet. Nous avons décidé que l'espace qui compterait serait celui requis par les champs de l'objet, le pointeur arrière et la case réservée pour la pile. Bref, c'est l'espace qui est consommé par un objet dans la région des objets et de la pile de marquage. De façon analogue, le temps qui est requis pour *fouiller* un objet (et non pas pour le marquer) est l'espace pris par cet objet dans la région en question. Pareillement pour le traitement d'un objet durant la phase de compactage. Que l'objet soit accessible ou non,

son traitement consomme une quantité de temps égale à l'espace qu'il occupe dans la région des objets. Nous ne comptons pas l'espace pris par l'identificateur étant donné que nous supposons que le nombre d'identificateurs a été choisi de façon à ce qu'ils ne puissent être épuisés (du moins, sans épuisement de l'espace dans la région des objets).

Cette *comptabilité* du temps a été adoptée parce qu'elle simplifie beaucoup le calcul du ratio. En effet, le ratio indique, d'une certaine façon, combien de cases mémoire doivent être traitées pour chaque case allouée. Plus un objet à allouer est long, plus il consomme de l'espace libre et plus il fournit de temps au GC lors de son allocation. Cette comptabilité concorde avec le critère sur les bornes de temps des opérations élémentaires imposé à un GC temps réel. Le critère est énoncé à la section 2.6.1.

Ainsi, il est simple de compter combien d'unités de temps il faut pour la phase de compactage. Chaque objet doit être traité une et une seule fois. L'ensemble des objets à traiter est constitué de ceux existant au début du cycle du GC, ceux alloués durant le marquage et ceux alloués durant le compactage. Le temps requis pour cette phase est donc l'espace qu'occuperont ces objets à la fin de la phase. On peut calculer aussi combien d'unités de temps sont requises par le marquage. Chaque objet doit être traité (fouillé) *au plus* une fois. L'ensemble des objets qui sont possiblement à considérer est constitué de ceux qui existaient au début du cycle, plus ceux qui ont été alloués durant le marquage.

Calcul du ratio

Nous allons maintenant expliquer la façon de calculer le ratio. Il est important de noter que le temps requis pour faire le marquage ne peut être plus long que celui requis pour faire le compactage. Aussi, il faut rappeler que l'application ne doit jamais conserver plus qu'une certaine fraction du tas en objets vivants. Soit $\alpha, 0 < \alpha < 1$, la fraction maximale occupée par les objets vivants. Nous allons montrer que si nous choisissons un ratio R égal à $\frac{5+3\alpha}{2-2\alpha}$, l'occupation du tas au début de chaque cycle sera au maximum $\frac{1+\alpha}{2}$.

Preuve par induction. *Base.* Au départ, le tas est complètement vide, donc l'occupation du tas est au plus $\frac{1+\alpha}{2}$. *Pas.* Il faut d'abord montrer que le cycle du GC aura eu le temps de se terminer avant que l'espace libre ne se soit épuisé. Par hypothèse d'induction, au maximum $\frac{1+\alpha}{2}$ du tas est occupé, au début du cycle. Soit v, $0 \leq v \leq \alpha$, la fraction du tas occupée par les objets vivants et m, $0 \leq v + m \leq \frac{1+\alpha}{2}$, la fraction du tas occupée par les objets morts. (Par abus de langage, nous parlons de l'occupation du tas, mais, en fait, c'est de l'occupation de la région des objets dont nous parlons réellement.) Clairement, au moins $\frac{1-\alpha}{2}$ du tas est libre. Après l'allocation de $\frac{1-\alpha}{2}$ du tas, nous prétendons que le GC a reçu assez de temps pour compléter son cycle. Après l'allocation de $\frac{1-\alpha}{2}$ du tas, grâce au ratio, le GC a reçu du temps pour traiter $\frac{5+3\alpha}{4}$ du tas. Le temps sera distribué en $\frac{1+3\alpha}{4}$ pour le marquage et 1 pour le compactage.

Le temps requis en pire cas pour le marquage est celui demandé si tous les objets vivants au début sont marqués et tous les objets alloués durant le marquage sont marqués. Comme le marquage n'est jamais plus long que le compactage, au plus $\frac{1-\alpha}{4}$ du tas est alloué durant le

marquage. Le temps en pire cas requis par le marquage est donc $v + \frac{1-\alpha}{4} \leq \frac{1+3\alpha}{4}$.

Le temps requis en pire cas pour le compactage est celui demandé pour traiter tous les objets existants au début et tous les objets alloués durant le cycle. Ce qui représente un temps en pire cas de $v + m + \frac{1-\alpha}{2} \leq 1$. Le cycle du GC peut se terminer avant que l'espace libre ne soit épuisé. Du moins, pour ce cycle.

Il reste à montrer qu'à la fin de ce cycle, il y aura au maximum $\frac{1+\alpha}{2}$ du tas qui sera occupé. En pire cas, les objets qui peuvent survivre au cycle sont ceux qui étaient vivants au début et ceux alloués durant le cycle. Il y en avait v au début. Au plus $\frac{1-\alpha}{2}$ ont été alloués durant le cycle. Donc, l'occupation au début du prochain cycle sera au plus $v + \frac{1-\alpha}{2} \leq \frac{1+\alpha}{2}$. Ce qui termine la preuve.

Bien entendu, dans une implantation concrète, il est préférable de choisir un ratio entier. Il suffit de prendre le plus petit entier supérieur ou égal à $\frac{5+3\alpha}{2-2\alpha}$.

4.5.4 Trois améliorations

Il est certain qu'un GC temps réel impose une pénalité sur la performance globale d'un système. Nous commençons par proposer deux améliorations qui permettent d'accroître l'efficacité du GC.

La première est une amélioration pour réduire le temps pris par le compactage. Normalement, un objet alloué durant le compactage l'est à la position du pointeur d'allocation, à la suite du dernier objet alloué. De plus, il est alloué marqué. Et, éventuellement, le pointeur de source du compactage parvient à cet objet et le déplace à sa nouvelle position. Là où on peut faire une économie, c'est lorsqu'il y a assez d'espace entre le pointeur de source et le pointeur de destination pour loger l'objet. En l'allouant immédiatement parmi les objets déplacés, on évite de le déplacer inutilement. La figure 4.2 illustre cette amélioration. Au lieu d'allouer l'objet marqué (en noir) à l'endroit pointé par *alloc*, il est alloué non marqué (en blanc) à l'endroit pointé par *dest*.

L'effet de cette première amélioration est de faire terminer les cycles de GC plus tôt. Ce qui permet, en général, d'avoir une occupation du tas moins grande à la fin de chaque cycle. Pour que cette amélioration soit profitable, il faut implanter aussi la suivante: retarder le lancement du cycle si le tas n'est pas encore assez plein. Comme on a vu à la sous-section précédente, le ratio R est calculé de façon à ce qu'il assure un bon fonctionnement du GC étant donnée une occupation initiale pouvant aller jusqu'à $\frac{1+\alpha}{2}$. Donc, il est inutile de déclencher le nouveau cycle tant que l'occupation est sous ce seuil. La technique pour régler le départ du GC consiste simplement à enlever de la banque de temps du GC la bonne quantité. Par exemple, s'il reste Δ en espace libre avant d'atteindre le seuil où le GC doit obligatoirement se déclencher, on enlève $R\Delta$ de sa banque de temps. Cette dernière amélioration ne nécessite pas que la première amélioration soit implantée.

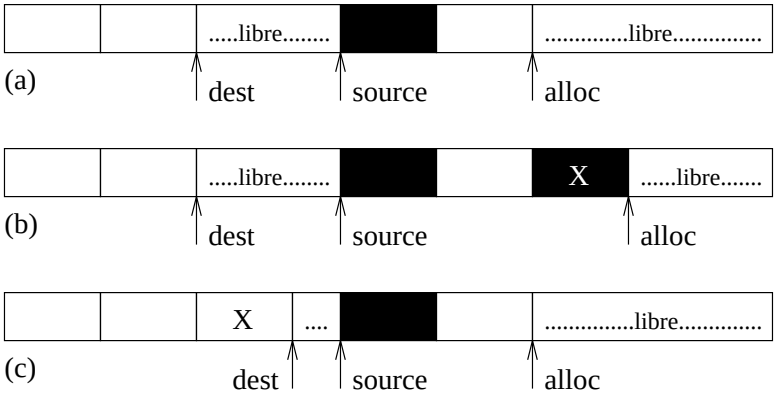


FIG. 4.2 - *Objet alloué immédiatement parmi les objets déplacés durant le compactage*
(a) Etat original du tas
(b) Allocation de X de la façon habituelle
(c) Allocation de X directement parmi les objets déplacés

La troisième amélioration à apporter consiste à enlever la contrainte exigeant qu'il y ait un petit nombre de racines dans l'application. On peut avoir un nombre non borné de racines et les fouiller de façon incrémentielle. Toutefois, ceci nécessite la mise en place d'une barrière en écriture sur les racines. La barrière consiste à marquer, si nécessaire, l'objet qui est stocké lors d'une écriture dans une racine. C'est-à-dire qu'elle fonctionne comme si les racines étaient les champs d'un objet déjà marqué et fouillé. Toute application qui utilise une pile ou un grand nombre de variables globales risque d'avoir à prendre une telle mesure.

Dans le cas des dernières versions de notre interprète, nous devons implanter cette amélioration à cause du fait que toutes les variables globales du programme Scheme sont rassemblées dans un tableau et elles peuvent ainsi introduire une quantité non bornée de racines. Ce ne sont toutefois que les variables globales du programme Scheme qui sont fouillées de façon incrémentielle. Les quelques variables de travail du noyau sont trop peu nombreuses et trop souvent modifiées pour être traitées de cette façon. Elles continuent d'être traitées de la façon décrite à la section 4.5.2.

Afin que la fouille des racines soit incrémentielle, il importe d'associer un coût à leur fouille. Comme cette opération est faite au début du marquage, c'est à cette phase que le travail est ajouté. Afin de ne pas briser l'invariant que le compactage n'est jamais plus court que le marquage, on doit "ajouter" artificiellement un temps équivalent à la phase de compactage. C'est-à-dire que dans le calcul du ratio, on doit considérer que le marquage est plus long (ce qui est naturel) et que le compactage est allongé de façon équivalente (ce qui est artificiel) même s'il n'y a pas plus de travail à faire dans cette phase.

Bien que les deux premières améliorations ne soient pas strictement nécessaires, nous les implantons ainsi que la troisième, qui, elle, est nécessaire.

4.5.5 Le calcul du ratio utilisé par l'interprète

Afin de *garantir* que le GC ne cause pas de pauses indûment longues, il faut calculer le ratio de la façon que nous avons décrite. Toutefois, pour parvenir à calculer ce ratio, il faut déterminer la fraction maximale du tas qui peut être occupée par des objets vivants. Ceci demande de faire une estimation de cette fraction à chaque fois qu'un programme Scheme est pré-traité pour être envoyé dans le micro-contrôleur. Bien sûr, l'estimation ne peut être faite automatiquement. Il faudrait le concours du programmeur afin qu'il détermine des bornes supérieures sur l'utilisation des objets de chaque type, du nombre de variables globales présentes, la quantité de chacune des fermetures pouvant être créées, etc. En bref, il s'agit de nombreuses statistiques à obtenir.

Bien qu'il soit possible de demander ces statistiques au programmeur, nous optons pour une approche plus simple. Le GC calcule un ratio à chaque cycle. Etant donnée l'occupation du tas au début du cycle, en objets vivants et morts, il calcule le ratio qui lui permet de terminer son cycle avant l'épuisement de l'espace libre. Ainsi, aucune collaboration n'est exigée pour le contrôle de la vitesse du GC. En pratique, le GC choisit un ratio relativement stable, comme nous avons pu le constater en faisant plusieurs tests. Ceci à condition que le programme Scheme ne tente pas de conserver plus qu'une certaine quantité d'objets vivants, bien entendu. En théorie, nous croyons qu'il est possible que le ratio ait à monter arbitrairement haut, si les circonstances sont parfaitement défavorables. En revanche, nous croyons aussi que de telles circonstances sont très improbables.

Une façon de trouver la fraction du tas occupée en objets vivants sans demander d'aide trop importante du programmeur serait de créer un simulateur. Le programmeur ferait fonctionner son programme en le soumettant aux conditions qui devraient maximiser la conservation d'objets par son programme. Le simulateur fournirait une mesure de l'occupation mémoire à la fin de la simulation. Toutefois, la qualité des chiffres déprendrait de la qualité des conditions imposées lors de l'interprétation du programme. Les résultats du simulateur ne pourrait pas servir à prouver qu'un ratio donné est adéquat. Nous n'avons pas créé de tel simulateur, étant donné que ce n'est pas un outil infaillible et que le calcul du ratio à chaque cycle est une technique qui fonctionne déjà bien en pratique.

Chapitre 5

Compilation vers du code-octets

Les dernières versions de l'interprète ne sont pas interactives. Ceci vient du fait que la programmation du micro-contrôleur ne change pas pour une application donnée. On peut donc utiliser un compilateur pour transformer le programme Scheme afin de produire un exécutable à installer dans la mémoire du micro-contrôleur.

La première section présente un aperçu de l'organisation du système en insistant sur le compilateur. Les autres sections abordent les différents aspects du processus de compilation.

5.1 Aperçu du système

5.1.1 Le système entier

La figure 5.1 résume les étapes de la construction d'un exécutable pour le micro-contrôleur à partir d'un programme Scheme. Le langage d'implantation de l'interprète de code-octets est le C. L'assembleur serait préférable au C si l'on voulait faire un produit fini avec le système. Pour notre travail, une implantation en C est suffisante. En effet, nous disposons d'un compilateur C qui génère des exécutables pour le micro-contrôleur que nous utilisons. De plus, une implantation en C peut être testée facilement sur une station de travail, contrairement à une implantation en assembleur du 68HC11.

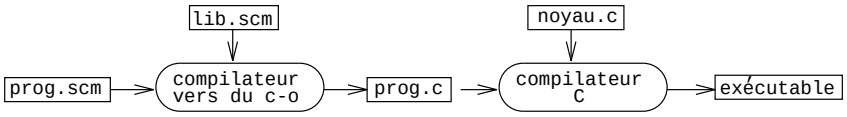


FIG. 5.1 - *Production d'un exécutable pour micro-contrôleur à partir du programme* `prog.scm`

Dans ce système compilé, la compilation n'est pas obligatoirement faite sur le micro-contrôleur. Il est même beaucoup plus simple de l'effectuer sur un ordinateur de plus grande puissance.

La compilation s'effectue en deux étapes: traduire le source Scheme en un fichier C; compiler ce fichier avec l'interprète de code-octets (le noyau) pour produire l'exécutable pour le micro-contrôleur. Un fichier contenant une partie des fonctions de la librairie (`lib.scm`) est utilisé lors de la compilation de Scheme vers C. Nous avons écrit nous-mêmes le compilateur vers du code-octets en Scheme. Scheme est un choix intéressant pour l'écriture de ce compilateur car il est facile de manipuler un programme Scheme sous forme de données Scheme. Le compilateur C est une version disponible publiquement. Nous parlons plus longuement de ce compilateur dans le chapitre sur les résultats expérimentaux (section 7.1.2).

Le fichier C produit par le compilateur Scheme contient des déclarations décrivant complètement le programme source et les fonctions de la librairie qui sont nécessaires. Un exemple de fichier de sortie est donné en appendice (voir section A.4). Les déclarations sont les suivantes. Il y a un tableau d'octets et sa longueur: celui-ci constitue le code-octets à exécuter. Il y a le nombre de variables globales et une description de leur valeur initiale. Il ne s'agit que d'une description et non de la représentation des valeurs initiales. Les descriptions sont inscrites directement dans le tableau C qui doit contenir la valeur des variables globales. Enfin, un second tableau d'octets et sa longueur donnent une description des constantes littérales du source.

Le noyau contient tout ce qui est nécessaire à l'interprétation des informations produites pas le compilateur. Premièrement, il y a bien sûr la machine virtuelle interprétant le code-octets. Deuxièmement, plusieurs fonctions Scheme de base y sont implantées, comme `cons`, `vector-ref`, etc. Troisièmement, un système de gestion mémoire s'occupe de l'allocation et de la récupération des objets. Le GC utilisé est basé sur la technique décrite au chapitre précédent. Enfin, une phase d'initialisation s'occupe de préparer les registres, le tas, les variables et les données avant le début de l'interprétation.

5.1.2 Le compilateur

Pour produire le fichier C en sortie, le compilateur effectue les étapes décrites ci-bas. Les étapes sont décrites en détail dans les sections qui suivent.

- Le programme source est lu.
- Les formes de syntaxe dérivées sont ramenées aux formes de base. Le programme résultant a le même comportement que l'original.
- Les expressions sont transformées en arbres de syntaxe. Chaque noeud d'un arbre a un type qui correspond à une des formes de syntaxe primitives. Plusieurs types de noeuds ont des champs supplémentaires servant à contenir des attributs calculés par la suite.
- Le programme est traversé afin de connaître l'ensemble des fonctions de la librairie dont il a besoin pour fonctionner.

- Les tâches suivantes sont effectuées lors d'une traversée du programme et des fonctions de la librairie:
  - Extraire les constantes littérales.
  - Trouver la position de chaque variable, que ce soit dans l'environnement global ou dans l'environnement lexical.
  - Détecter les variables globales mutables.
  - Dans chaque fonction, compter les paramètres formels et indiquer s'il y a un paramètre reste.
- La valeur initiale des variables globales de la librairie est déterminée.
- Les deux tâches suivantes sont effectuées lors d'une nouvelle traversée du programme et des fonctions de la librairie:
  - Découvrir statiquement le résultat d'une référence de variable, lorsque c'est possible.
  - Remplacer l'appel d'une fonction complexe par l'appel d'une fonction plus simple, lorsque c'est possible. Par exemple, un appel à la fonction `append` peut être remplacé par un appel à la fonction interne `append2`, qui est spécialisé pour deux arguments.
- La position de chaque variable globale est fixée.
- Le code-octets est généré.
- La description des constantes littérales est générée.
- La description de la valeur initiale des variables globales est générée.

Les différentes parties du compilateur sont décrites dans les sections suivantes, mais pas nécessairement dans le même ordre que les étapes énumérées ici. Les parties sont plutôt regroupées selon leur fonction. La génération du code-octets est décrite dans le chapitre suivant, qui traite en profondeur de la machine virtuelle.

5.2 Réduction des formes syntaxiques dérivées

La réduction des expressions du programme source en formes syntaxiques de base permet de diminuer le nombre de formes syntaxiques différentes dans le programme. De cette façon, il ne reste que quelques formes fondamentales à traiter par la suite. Ainsi, le reste du compilateur est plus simple et donc moins susceptible de contenir une erreur. De plus, la machine virtuelle qui doit interpréter le code-octets est, elle aussi, plus simple.

5.2.1 Les formes de base

Avant de décrire les différentes réductions que nous effectuons sur les expressions, il importe de préciser quelles sont les formes de base.

Liste des formes syntaxiques de base

Les formes de base que nous avons choisies sont presque les mêmes que celles décrites dans le R^4RS:

- Les références aux variables.
- Les constantes.
- Les appels de fonctions.
- Les lambda-expressions ayant une seule expression pour corps.
- Les conditions (expressions `if`) avec ou sans alternative.
- Les affectations (expressions `set!`).
- Les blocs d'expressions (expressions `begin`). Ceci n'est pas une expression primitive dans le standard.
- Les définitions de variables globales (expressions `define`). Ceci n'inclut pas les définitions locales qui peuvent être faites au début du corps de certaines formes.

Les expressions exprimées sous une des ces formes ne sont pas transformées en d'autres formes plus simples. Toutefois, certaines sont uniformisées afin de simplifier leur manipulation par la suite.

Uniformisation des formes de base

Bien que la lambda-expression soit une de nos formes de base, elle se doit de ne contenir qu'une seule expression dans son corps. Lorsqu'une lambda-expression en contient plus qu'une, elles sont placées dans une forme `begin`.

Les formes `begin` imbriquées sont aplaties. Cette mesure est utile principalement à cause que la récriture des expressions peut introduire des formes `begin` superflues. De plus, une forme `begin` ne contenant qu'une seule expression est remplacée par cette expression.

Les définitions aussi subissent une uniformisation. Le R^4RS permet de définir directement une variable contenant une fonction avec la notation suivante: `(define (`⟨variable⟩ ⟨formals⟩`)` ⟨body⟩`)`. Dans un tel cas, la création de la variable et de la fonction sont réexprimées séparément. On obtient alors une expression comme celle-ci: `(define` ⟨variable⟩ `(lambda` ⟨formals⟩ ⟨body⟩`)))`. L'apparence finale des expressions `define` est toujours semblable à celle des expressions `set!`.

En Scheme, il est possible d'effectuer des définitions internes au début du corps d'une fonction ou d'une forme de liaison. La sémantique de ces définitions correspond à celle du `letrec`. Une transformation comme celle montrée dans l'exemple est effectuée lorsque des définitions internes

se trouvent au début d'un corps. Or, comme nous le voyons plus loin, la réduction des formes dérivées fait en sorte qu'un corps finit toujours par être le corps d'une lambda-expression. Ainsi, il suffit de faire la détection des définitions internes lors de l'uniformisation des lambda-expressions.

```
(lambda (a)                  (lambda (a)
  (define (f x)                (letrec ((f (lambda (x)
    ...)                                     ...))
  (define (g x)        ⟹                 (g (lambda (x)
    ...)                                     ...)))
  (+ (f a) (g a)))             (+ (f a) (g a))))
```

Un appel à une lambda-expression sans paramètre est remplacé par le corps de la lambda-expression. Une telle expression est habituellement introduite par la réduction des formes dérivées.

((lambda () ⟨body⟩)) ⟹ ⟨body⟩

5.2.2 Les réductions proprement dites

Nous pouvons maintenant décrire les réductions qui sont faites sur les formes dérivées. Afin de ne pas allourdir inutilement le texte avec trop de descriptions et d'exemples de transformations, nous nous limitons aux transformations qui ne sont pas décrites dans le standard ou que nous effectuons différemment.

La réduction des expressions `cond` se fait selon les règles décrites dans la section 7.3 du R^4RS sauf dans le cas d'une clause "`=>`". ⟨tmp⟩ représente une variable dont le nom est unique. Elle ne peut masquer aucune autre variable.

```
(cond (⟨test⟩ => ⟨recipient⟩)
      ⟨clause₂⟩ ...)
⟹
(let ((⟨tmp⟩ ⟨test⟩))
  (if ⟨tmp⟩
      (⟨recipient⟩ ⟨tmp⟩)
      (cond ⟨clause₂⟩ ...)))
```

La réduction des expressions `case` n'est pas faite comme dans le R^4RS. A nouveau, nous utilisons une variable au nom unique afin d'obtenir une expression transformée moins grosse.

```
(case ⟨key⟩
  ((d1 ...) ⟨sequence⟩)
  ...)
⟹
(let ((⟨tmp⟩ ⟨key⟩))
  (cond ((⟨memv⟩ ⟨tmp⟩ (d1 ...))
```

```
    ⟨sequence⟩)
  ...))
```

Dans notre règle de transformation, et dans celle du standard d'ailleurs, il est indiqué à l'aide de "⟨memv⟩" qu'une expression s'évaluant à la fonction `memv` doit être insérée.

Il est important de noter que, dans un tel cas, il n'est pas suffisant d'y mettre la référence de variable "`memv`". Il y a deux bonnes raisons pour cela. Premièrement, lorsque la réduction de l'expression `case` est faite, on ne sait pas dans quelle autre expression elle se trouve. Si elle se trouve dans une forme de liaison qui a lié la variable `memv` à une valeur quelconque, le résultat de la réduction ne sera clairement pas équivalent à l'expression originale. Deuxièmement, même si l'environnement lexical baignant l'expression `case` ne masque pas la variable `memv`, il n'est pas certain que la variable globale ainsi nommée contienne toujours la fonction `memv`. Les mutations sont permises sur toutes les variables, donc la variable `memv` peut avoir été mutée ou redéfinie avant le moment où l'expression `case` est exécutée.

Notre façon de régler ce problème consiste à ajouter quelques expressions au début du programme. Ce sont 6 définitions qui copient sous un autre nom les fonctions comme `memv`. Les 6 fonctions ainsi protégées sont: `memv`, `make-promise`, `list->vector`, `list`, `append2` et `cons`. Comme on le voit un peu plus loin, `make-promise` sert dans la réduction des formes `delay` et les 4 dernières fonctions de la liste sont utilisées dans la réduction des formes `quasiquote`. Les 6 définitions copient les fonctions dans des variables dont le nom est unique.

La forme `and` est réduite d'une façon un peu plus simple que celle proposée par le R^{4}RS. La différence réside dans les cas où il y a plus d'une expression interne. La réduction que nous utilisons est la suivante:

```
(and ⟨test1⟩ ⟨test2⟩ ...)
⟹
(if ⟨test1⟩ (and ⟨test2⟩ ...) #f)
```

La forme `or` est réduite d'une façon plus simple que celle donnée par le standard grâce à l'utilisation d'une variable au nom unique. Il s'agit du cas où il y a plus d'une sous-expressions dans la forme:

```
(or ⟨test1⟩ ⟨test2⟩ ...)
⟹
(let ((⟨tmp⟩ ⟨test1⟩))
  (if ⟨tmp⟩
      ⟨tmp⟩
      (or ⟨test2⟩ ...)))
```

La réduction des formes `let` et `let*` est faite selon les règles de transformation du standard. La réduction des formes `letrec` est faite de façon différente. Elle ne respecte pas la sémantique de Scheme, mais seulement au niveau de la détection d'erreurs, donc elle est acceptable pour l'interprète. La réduction est la suivante. S'il n'y a aucune variable liée dans une forme `letrec`,

elle est transformée en une forme `let`. S'il y a des variables liées, la transformation est la suivante:

```
(letrec ((⟨variable1⟩ ⟨init1⟩)
         ...)
  ⟨body⟩)
⟹
(let ((⟨variable1⟩ #f)
      ...)
  (set! ⟨variable1⟩ ⟨init1⟩)
  ...
  (let ()
    ⟨body⟩))
```

La forme `let` sans liaison qui entoure le corps du `letrec` sert à assurer que d'éventuelles définitions au début de ce corps sont encore au début d'un corps après la réduction. Ce `let` artificiel est éliminé par la suite par le processus d'uniformisation des appels de fonctions.

Les formes `do` et "`let` nommé" sont réduites de façon similaire à la méthode décrite dans le standard. Il en va de même pour les formes `delay`. La méthode de réduction des formes `delay` explique la présence de la fonction `make-promise` dont nous avons parlé plus tôt. Le standard ne donne pas de méthode précise pour effectuer la transformation des formes `quasiquote`. Nous montrons quelques exemple représentatifs des réductions que nous effectuons sur ces formes.

```
`#(1 ,a 5)
⟹
(⟨list->vector⟩ (⟨cons⟩ '1 (⟨cons⟩ a '(5))))

``,,a
⟹
(⟨list⟩ 'quasiquote (⟨list⟩ 'unquote a))

`(1 ,@(list a b) 7)
⟹
(⟨cons⟩ '1 (⟨append2⟩ (list a b) '(7)))
```

Il est important de noter que les parties qui ne contiennent pas suffisamment de "`unquote`"s pour être évaluables ne sont pas décomposées et sont incluses dans une forme `quote`.

Ceci conclut la description des méthodes que nous utilisons pour réduire les expressions du programme source à quelques formes simples.

5.3 Transformation en arbres de syntaxe

La transformation du programme en arbres de syntaxe est une opération fort simple. Il s'agit de transformer chaque expression de base en un noeud qui la symbolise. Par exemple, une expression `if` est transformée de la façon suivante:

`(if` $\langle \text{exp}_1 \rangle$ $\langle \text{exp}_2 \rangle$ $\langle \text{exp}_3 \rangle$`)` $\Longrightarrow$

if	N_1	N_2	N_3

Chaque expression interne à l'expression `if` est aussi tranformée en noeud. L'exemple de cette expression est le plus simple. La plupart des autres expressions sont transformées en noeuds qui ont quelques champs supplémentaires. Ces champs supplémentaires sont utilisés plus tard comme champs d'attributs. Le compilateur y met les résultats de calculs faits lors de traversées de l'arbre de syntaxe. Par exemple, les noeuds symbolisant les références de variables contiennent entre autre un champ indiquant si la variable en question est globale ou lexicale et un champ pour indiquer la position de cette variable.

Les noeuds sont implantés à l'aide de vecteurs et le type du noeud est tout simplement un petit entier. La discrimination entre les différents types de noeuds se fait donc très rapidement. Elle s'effectue grâce à un accès à un vecteur contenant les actions requises pour le traitement des différents noeuds.

5.4 Sélection des fonctions de la librairie

Un programme Scheme typique n'utilise pas toutes les fonctions de la librairie. Or, même si cette librairie est relativement réduite, il est avantageux de n'inclure que les fonctions requises par le programme.

Dans l'aperçu de notre système, nous mentionnons que certaines fonctions de la librairie sont implantées directement dans le noyau et que les autres sont implantées en Scheme. Les fonctions implantées dans le noyau font toujours partie de l'interprète résultant de la compilation. Mais celles qui sont implantées dans le fichier de librairie peuvent être incluses au besoin seulement.

La façon de procéder consiste à trouver toutes les références aux variables globales; charger la librairie; identifier les fonctions qui sont requises, ceci inclut les fonctions dont dépendent celles qui sont directement utilisées par le programme.

5.4.1 Références globales

Cette phase consiste à trouver toutes les références aux variables globales. Une référence à une variable lexicale ne compte pas. La liste des variables de l'environnement lexical est donc maintenue lors de cette recherche. Si une variable nommant une fonction de la librairie n'est

jamais référencée alors on peut en conclure que le programme n'utilise pas cette fonction. Du moins, il ne l'utilise pas directement.

Par la suite, lorsque la librairie est chargée il est possible de faire l'intersection entre la liste des variables globales référencées et les noms des fonctions de la librairie. On obtient ainsi la liste des fonctions potentiellement utilisées par le programme. Il faut noter que c'est un problème indécidable de vérifier si une fonction est véritablement utilisée. Nous nous contentons donc de cette approximation.

5.4.2 La librairie

Le fichier source de librairie Scheme contient plusieurs déclarations et définitions de fonctions. Il est utilisé à chaque compilation de programme.

Tout d'abord, nous montrons que les fonctions de la librairie et les expressions du programme voient des espaces de noms différents. Ensuite, nous décrivons l'organisation du fichier de librairie. Enfin, la méthode d'extraction est décrite.

2 espaces de noms

Nous avons divisé l'espace de noms en deux. Cette division sert à s'assurer que les fonctions de la librairie ont toujours accès aux autres fonctions quelles que soient les affectations faites par le programme. Nous illustrons ce fait avec un exemple. Notre implantation de la fonction standard `+` utilise la fonction interne `math+2`. Celle-ci additionne exactement deux nombres. Elle n'est pas standard et, par conséquent, la variable `math+2` n'est pas visible depuis le programme. Nous disons que cette dernière est cachée. Même si le programme définit une variable globale s'appelant `math+2`, il ne s'agit pas de la même variable. Un accès à cette variable est interprété différemment suivant qu'il est effectué dans le programme ou dans une fonction de la librairie.

Les fonctions de la librairie voient toutes les variables définies ou déclarées dans la librairie. Les expressions du programme voient leurs propres variables et les variables visibles de la librairie. De plus, même si le programme modifie une des variables visibles de la librairie, les fonctions de la librairie continuent à voir le contenu original de cette variable.

Organisation du fichier de la librairie

Le fichier de la librairie est séparé en quatre parties. Les parties sont délimitées par l'expression `#f`.

La première partie sert à déclarer les fonctions implantées dans le noyau ainsi que leur numéro. Chaque fonction est déclarée grâce à une donnée de la forme: `(car . 6)`. Cette partie ne sert pas à dire si les fonctions en question sont visibles ou cachées.

```
(quit . 2)
(car  . 6)
#f
(define max2 (lambda (x y) (if (> x y) x y)))
#f
quit
#f
car
(define caar (lambda (p) (car (car p))))
(define max (lambda l (foldl1 max2 l)))
#f
```

On trouve les exemples suivants:
— La fonction `quit` du noyau est visible mais non-standard.
— La fonction `max2` est cachée et sert à l'implantation de la fonction standard `max`.
— La fonction standard `car` du noyau sert entre autres à l'implantation de la fonction standard `caar`.

FIG. 5.2 - *Extraits du fichier de librairie*

Les trois autres parties contiennent les déclarations et définitions de toutes les fonctions visibles et cachées. La deuxième partie regroupe les fonctions cachées. La troisième regroupe les fonctions visibles non-standard. La quatrième regroupe les fonctions standard (donc visibles).

Dans ces trois dernières parties, les déclarations et définitions peuvent prendre une de trois formes.

- Une définition ordinaire a la forme suivante:
  `(define` ⟨variable⟩ `(lambda` ⟨formals⟩ ⟨body⟩`)))`.
  La section dans laquelle se trouve une définition détermine sa visibilité.
- Une définition d'alias a la forme suivante:
  `(define` ⟨variable$_1$⟩ ⟨variable$_2$⟩`)`.
  Cette forme permet de définir une variable qui doit contenir exactement la même fonction qu'une autre variable.
- Une déclaration de fonction du noyau a la forme suivante:
  ⟨variable⟩.
  Elle sert à donner l'information de visibilité qui n'est pas donnée dans la première partie.

La distinction entre la troisième et la quatrième partie (standard versus non-standard) est seulement conceptuelle. Le compilateur traite de la même façon les fonctions qui en font partie.

La figure 5.2 illustre l'organisation du fichier.

Factorisation du code

Afin de réduire la quantité de code nécessaire à l'implantation de la librairie, les fonctions de la librairie sont écrites de façon à réutiliser le code le plus possible. Le travail qui est commun à plusieurs fonctions est regroupé en fonctions générales. Voici les principaux exemples:

- Les fonctions comme `append`, `+` et `-` consistant à accepter un nombre variable d'arguments et à appliquer un opérateur binaire sur les arguments en associant par la gauche ou par la droite sont toutes implantées à l'aide de fonctions élévées de traitement de listes. Ces fonctions sont habituellement appelées `fold-...`.
- Les fonctions `memq`, `memv` et `member` utilisent toutes la fonction cachée `generic-member` qui est paramétrable avec une fonction de comparaison.
- `Assq`, `assv` et `assoc` utilisent la fonction `generic-assoc`, qui prend en paramètre la fonction de comparaison appropriée.
- Les comparaisons entre nombres sont implantées en utilisant la fonction `generic-compare` pouvant tester une relation entre les éléments d'une liste en les prenant deux à deux.
- Les comparaisons entre caractères qui ne distinguent pas les majuscules des minuscules sont implantées comme une comparaison entre caractères précédée d'une réduction en minuscules.
- Les comparaisons entre chaînes de caractères sont toutes implantées en utilisant un comparateur de chaînes générique qui retourne -1, 0 ou 1 comme résultat. Les fonctions de comparaison de caractères `char<?` et `char=?` ou `char-ci<?` et `char-ci=?` lui sont passées et déterminent du même coup si la comparaison est sensible à la différence entre majuscules et minuscules.

5.4.3 Extraction des fonctions requises

Lorsque le compilateur lit le fichier de librairie, il classe les fonctions selon leur visibilité et selon leur nature (du noyau, en Scheme ou simple alias). Par une intersection entre la liste des variables globales référencées par le programme et la liste des fonctions visibles de la librairie, il trouve celles qui sont directement requises par le programme. Ensuite, celles-ci sont réduites, transformées en arbres et parcourues afin d'ajouter les fonctions dont elles ont elles-mêmes besoin. Le tout se poursuit jusqu'à ce que toutes les fonctions requises aient été incluses.

Le programme et les fonctions requises forment l'ensemble du code Scheme à compiler. Toutefois, le compilateur ne mélange pas ces deux parties car elles accèdent à des espaces de noms différents.

5.5 Traitement des constantes littérales

Les constantes littérales sont présentes dans un programme sous forme de nombres, de chaînes de caractères, de caractères, de booléens et d'expressions `quote`. Le compilateur accumule certaines des constantes lors de sa première traversée. Les constantes sont traitées différemment selon qu'elles s'évaluent à des données allouées ou non-allouées.

Les constantes non-allouées comme le `5` dans `(+ x 5)` ne subissent pas de traitement particulier. Plus tard, elles sont compilées en des instructions. Les constantes allouées comme la liste `(1 3 5)` dans `(car '(1 3 5))` sont traitées spécialement. C'est le traitement de ces constantes qui est décrit dans cette section.

Les constantes allouées contiennent nécessairement au moins une paire, un vecteur ou une chaîne de caractères. Il n'y a pas de représentation externe pour les fermetures et les continuations. Aussi, les constantes allouées ont comme propriété d'être des structures arborescentes. En effet, toute autre structure ne peut s'écrire avec les représentations externes permises en Scheme.

Dans cette section, nous utilisons le mot "morceau" pour désigner un sous-arbre de l'arbre qu'est une constante.

Les constantes allouées subissent les étapes suivantes: elles sont découpés en morceaux; l'ensemble des morceaux est codé en une description; la description est écrite sous forme de tableau d'octets en C; le noyau rebâtit les constantes lors de l'initialisation. Chacun de ces points est maintenant traité plus en détail.

5.5.1 Découpage des constantes

Les morceaux

Durant la première traversée de l'arbre de syntaxe, si le compilateur rencontre un noeud représentant une constante allouée, il effectue le découpage de la constante et inscrit un numéro de constante dans le noeud.

Le découpage d'une constante consiste à numéroter chaque sous-morceau de cette constante et à tous les décrire. Par définition, la constante elle-même est un de ses sous-morceaux. La description d'un morceau consiste à décrire l'objet qui est sa racine. La description inclut le type, une éventuelle valeur et les numéros des éventuels "fils" de la racine.

Par exemple, prenons la liste constante `(1)`. Il s'agit d'un paire contenant dans l'ordre le nombre `1` et la liste vide. Il y a donc 3 morceaux: la paire elle-même, le nombre et la liste vide. Chacun se voit attribuer un numéro. La description de la paire est le type *pair* et les numéros des morceaux qu'elle contient. La description du nombre est le type *number* et la valeur 1. La description de la liste vide est le type "liste vide". La constante reçoit aussi un numéro de constante, qui est généralement différent de son numéro de morceau.

L'algorithme de découpage accumule l'ensemble des constantes et morceaux qu'il traite. Il les conserve dans deux vecteurs. Dans le premier vecteur se trouvent les descriptions des morceaux. Par une attribution en postfixe des numéros de morceaux, on fait en sorte que les numéros des sous-morceaux d'un morceau ont des numéros inférieurs au numéro de ce dernier. Le deuxième vecteur contient le numéro de morceau de chaque constante. Autrement dit, il contient le numéro de l'objet à la racine de cette constante.

Un exemple plus complet de découpage est donné plus loin.

La factorisation des constantes et des morceaux

Le numéro donné à une constante n'est pas nécessairement unique. Deux constantes identiques (au sens de `equal?`) se voient attribuer le même numéro. Ceci implique que durant l'interprétation du programme, les deux expressions distinctes dans le source retournent la même constante (au sens de `eq?`). Cette factorisation est permise par le R^4RS car il est interdit d'effectuer des mutations sur les données provenant de constantes littérales.

La factorisation s'étend aussi aux morceaux. C'est-à-dire que tous les morceaux semblables sont regroupés en un seul. Ainsi, après reconstruction des constantes, l'interprète possède des morceaux qui sont inclus dans plusieurs constantes différentes.

Exemple

Nous illustrons le découpage à l'aide des deux constantes suivantes: `(#f "s")` et `#(2 #\a s)`. Les constantes reçoivent respectivement les numéros 0 et 1. Les figures qui suivent montrent l'effet du découpage.

Numéro	Morceau	Description	
0	`#f`	*Boolean*:	`#f`
1	`"s"`	*String*:	`"s"`
2	`()`	"Empty list"	
3	`("s")`	*Pair*:	`(1 2)`
4	`(#f "s")`	*Pair*:	`(0 3)`
5	`2`	*Number*:	`2`
6	`#\a`	*Char*:	`#\a`
7	`s`	*Symbol*:	`(1)`
8	`#(2 #\a s)`	*Vector*:	`(5 6 7)`

Numéro de constante	Numéro de morceau
0	4
1	8

On peut remarquer plusieurs choses dans ces figures. Premièrement, la représentation de chaque morceau se réfère, si c'est le cas, seulement à des morceaux ayant un plus petit numéro. Deuxièmement, il y a partage de la chaîne entre les deux constantes. Troisièmement, les morceaux de tête des deux constantes n'ont pas les mêmes numéros que les constantes elles-mêmes.

5.5.2 Codage des constantes

Une fois que toutes les constantes sont découpées, leur codage peut être effectué. La description codée consiste en une suite d'octets. La description complète des constantes comporte les informations suivantes: le nombre de morceaux; la description de chaque morceau, dans l'ordre de leur numérotation; le nombre de constantes; le numéro du morceau de tête de chaque constante.

Les nombres et les longueurs sont codés par deux octets. Le premier octet est le plus significatif.

A l'exclusion du codage de la description de chaque morceau, le reste du codage ne demande pas vraiment d'autres explications. Il reste donc à montrer le codage que nous employons pour décrire les morceaux.

Liste vide Elle est représentée par le caractère "`0`".

Paire La représentation comprend le caractère "`1`" suivi des numéros des deux champs de la paire.

Booléen Pour le booléen faux, la représentation est "`2f`". Pour le booléen vrai, elle est "`2t`".

Caractère La représentation est le caractère "`3`" suivi d'un octet décrivant le caractère.

Chaîne de caractères Le caractère "`4`" est suivi de la longueur et de tous les caractères de la chaîne.

Symbole Un symbole est représenté par le caractère "`5`" suivi du numéro de la chaîne contenant le nom.

Nombre La représentation est le caractère "`6`", suivi du caractère "`+`" ou "`-`" selon le signe du nombre et ensuite vient le nombre en valeur absolue. Ce codage permet de représenter les nombres indépendamment du taggage utilisé par l'interprète.

Vecteur Il est représenté par le caractère "`7`", suivi de la longueur et du numéro de chacun de ses sous-morceaux.

Cette représentation n'est pas la plus compacte que l'on puisse imaginer. Mais nous montrons dans la prochaine section que ça n'a pas beaucoup de conséquences.

5.5.3 Reconstruction des constantes

Cette étape du traitement des constantes ne fait pas partie du compilateur à strictement parler, mais nous préférons la décrire ici.

Méthode de reconstruction

Lors de l'initialisation, le noyau convertit la description codée en un vecteur de constantes. Chaque constante est placée dans la case qui porte son numéro. Les étapes de la reconstruction sont les suivantes:

- Lire le nombre de morceaux et créer un vecteur de la longueur indiquée.
- Pour chaque entrée du vecteur, lire la description d'un morceau, bâtir celui-ci et le stocker dans le vecteur. A ce point, tous les morceaux sont reconstruits. En particulier, les constantes aussi.
- Lire le nombre de constantes et créer un vecteur approprié.
- Remplir dans l'ordre chaque entrée de ce vecteur avec le morceau dont le numéro est indiquée dans la description codée. Ce vecteur est le vecteur des constantes désiré.
- Abandonner le vecteur des morceaux.

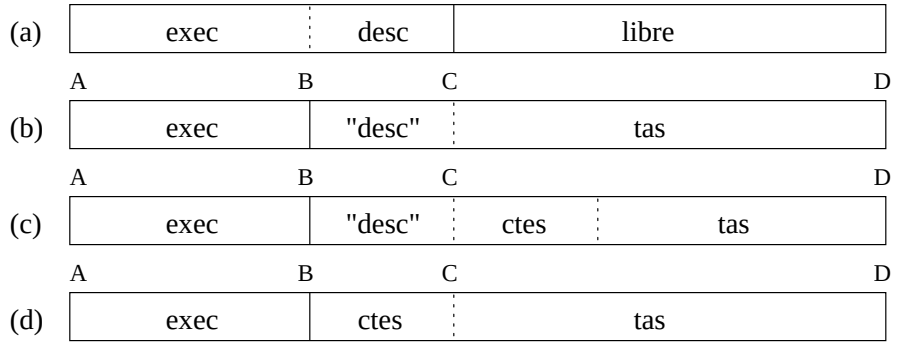
La reconstruction est très simple. Tout est placé dans le bon ordre par le compilateur. Par la suite, l'évaluation d'une expression devant retourner une constante allouée demande simplement de faire une référence dans le vecteur des constantes.

Abandon de la description codée

Un des avantages de cette technique de transmission des constantes consiste en la possibilité d'éliminer la description codée une fois que le vecteur des constantes a été construit.

Avant l'initialisation du noyau, la description occupe un espace vaguement semblable à la taille des constantes à rebâtir. Après l'initialisation, seules restent les constantes. Cette technique est donc économe en mémoire, en plus d'être simple.

Malheureusement, notre implantation en C ne comporte pas cette possibilité d'abandonner la description codée. Toutefois, une implantation en assembleur pourrait effectuer cette tâche facilement. En assembleur, il est relativement aisé de faire en sorte que le tableau contenant la description se retrouve à la toute fin de l'exécutable. Quand le programme est chargé, il n'occupe pas toute la mémoire du micro-contrôleur et l'excédent devient le tas. En ayant la description juste à côté de l'excédent de mémoire, le tas peut s'étendre du début de la description à la fin de la mémoire.



(a) L'exécutable est chargé au début de la mémoire du micro-contrôleur (A-C). La description des constantes se trouve à sa fin (B-C). (b) A l'initialisation du tas, la description est transformée en une chaîne de caractères Scheme. Notons que le tas (B-D) commence au début de l'espace occupé précédemment par la description. (c) Les constantes sont reconstruites. (d) La chaîne contenant la description est abondonnée, c'est-à-dire que la référence à la chaîne est écrasée et le GC se charge éventuellement de la ramasser.

FIG. 5.3 - *Traitement idéal de la description des constantes*

La figure 5.3 montre les différentes étapes de la reconstruction des constantes telle qu'elle pourrait être faite en assembleur. Elle montre de quelle façon on se débarrasse de la description.

Autres techniques de transmission

Nous avons préféré notre approche aux deux suivantes.

La première consiste à produire une description du tas tel qu'il devrait avoir l'air si les constantes y avaient été bâties. Cette approche demande moins de travail de la part du noyau. Il n'a même pas à interpréter quelque code que ce soit pour obtenir les constantes. Toutefois, elle augmente beaucoup la complexité du travail à faire par le compilateur. Celui-ci devient complètement dépendant de la représentation des objets dans l'interprète et il devient plus dur à modifier.

La deuxième consiste à ajouter au début du programme des expressions supplémentaires visant à construire les constantes et à les conserver dans un vecteur ou dans des variables à nom unique. Une expression constante consisterait en une référence à une de ces variables. Cette approche n'est pas trop complexe, mais elle ne permet pas de se débarrasser de la "description" des constantes, c'est-à-dire des expressions ajoutées. Elle n'est donc pas très compacte.

Malgré que notre approche soit compacte, elle demande tout de même l'ajout d'expressions au programme. Il s'agit du traducteur de la représentation vers les constantes. Ce traducteur n'est toutefois pas très complexe, étant donné la simplicité du codage. De plus, cet espace est fixe

et ne dépend pas de la quantité de constantes utilisées dans le programme source, contrairement à l'approche précédente.

5.6 Localisation des variables

Cette partie comprend les deux tâches suivantes. Premièrement, trouver la position dans l'environnement global ou lexical d'une variable impliquée dans une référence, une affectation ou une définition. La position ainsi trouvée permet au générateur de code-octets de produire la bonne instruction. Deuxièmement, recueillir la liste des variables globales.

Un accès à une variable globale n'est pas traité de la même façon qu'un accès à une variable lexicale.

Lorsqu'il s'agit d'une variable lexicale, il faut trouver les coordonnées de la variable dans l'environnement d'évaluation en vigueur au moment de l'accès à la variable. La représentation employée dans les dernières versions de l'interprète est celle des blocs d'activation chaînés (voir la section 3.4.3). Les coordonnées d'une variable lexicale sont donc le nombre de sauts de blocs et la position de la variable dans le bloc atteint. En plus de trouver les coordonnées d'une variable, le fait que cette variable soit seule ou non dans son bloc est noté. Cette dernière information sert ensuite lors de la génération du code-octets: la position de la variable dans le bloc n'a pas à être spécifiée lorsque celle-ci est seule dans son bloc (voir section 6.1.4).

Lorsqu'il s'agit d'une variable globale, il faut trouver son numéro. Toutes les variables globales introduites par le programme ou la librairie reçoivent un numéro. Une particularité de l'attribution des numéros vient du fait qu'il y a deux espaces de noms: un pour la librairie, un pour le programme (section 5.4.2). Un accès à une variable du même nom ne concerne pas toujours la même variable. Cette numérotation permet en même temps de compter les variables globales.

5.7 Valeur initiale des variables globales

Le compilateur trouve la valeur initiale des variables globales introduites par la librairie. Ce calcul est simple mais il permet d'effectuer plusieurs optimisations intéressantes par la suite.

Nous commençons par décrire le type de valeurs initiales que l'on calcule pour ces variables. Nous voyons ensuite les optimisations que l'on peut effectuer à l'aide de ces valeurs. Enfin, nous décrivons très rapidement la méthode utilisée pour transmettre les valeurs initiales à l'interprète.

5.7.1 Valeur initiale

Variables de la librairie

La fonction correspondant à chacune de ces variables est recherchée. Ainsi, on sait si la fonction présente au départ dans une variable est une fonction du noyau ou une fermeture. Dans le premier cas, le numéro de la fonction est noté. Dans le deuxième cas, c'est la lambda-expression générant la fermeture qui est notée. Plus tard, lors de la production du code-octets, l'adresse du corps de la fermeture peut être calculée.

Il est important de noter que toutes les fonctions sous forme de fermetures qui sont introduites par la librairie ont un environnement de définition vide. C'est pourquoi il est suffisant de noter seulement la lambda-expression. La syntaxe des définitions admissibles dans le fichier de librairie (voir section 5.4.2) force les fermetures qui y sont introduites à avoir un environnement vide.

La valeur initiale des alias, c'est-à-dire des variables qui sont définies comme ayant la même valeur qu'une autre variable, est calculée aussi. Pour trouver leur valeur, il suffit de suivre les alias jusqu'à ce que l'on tombe sur une variable définie avec une valeur qui est une fonction du noyau ou une fermeture.

Variables du source

Nous ne cherchons pas à trouver les valeurs des variables introduites par le programme. Premièrement, les programmes ne sont pas organisés aussi simplement que le fichier de librairie et le calcul de la valeur initiale peut être passablement complexe. De plus, à strictement parler, la valeur des variables au début de l'exécution du programme est arbitraire. Deuxièmement, la valeur initiale des variables du programme peut être de n'importe quel type. En particulier, les fermetures peuvent posséder un environnement de définition non-vide. Ce type de valeur initiale n'est pas vraiment utile pour nos optimisations.

Nous attribuons donc la valeur initiale `#f` aux variables du programme.

5.7.2 Utilité des valeurs initiales

Variables pré-définies

La première utilisation de la valeur initiale est tout simplement de connaître à l'avance la valeur qui doit être mise dans certaines variables globales. Il est préférable que les variables globales du programme contiennent déjà leur valeur initiale plutôt que de générer du code-octets représentant des définitions complètes. En effet, lors de la génération du code-octets pour les fonctions de la librairie, seul le corps des fonctions est compilé dans le cas des fermetures, et rien n'est généré dans le cas des fonctions du noyau. Ainsi, pour chaque fonction de la librairie,

les instructions pour la création d'une fermeture, le cas échéant, et pour le stockage de la valeur sont éliminées.

Connaissance *a priori* de la fonction appelée

Le fait de connaître la valeur initiale des variables de la librairie permet parfois de connaître statiquement le résultat d'une référence à une variable. Le résultat peut être connu à deux conditions: la variable référencée est globale et a une valeur initiale connue; cette variable ne risque pas d'être mutée.

Nous voyons dans la section 5.8 que ceci peut permettre de remplacer l'appel d'une certaine fonction par l'appel d'une fonction plus simple. Le fait de connaître la fonction appelée permet aussi de générer du code-octets plus compact, principalement dans le cas d'un appel de fonction du noyau.

5.7.3 Codage des valeurs initiales

Une description de la valeur initiale de chaque variable est transmise au noyau via le fichier généré par la compilateur. On se souvient que ce fichier contient entre autres un tableau de variables globales. Or, en C, on peut spécifier la valeur initiale des entrées d'un tableau. Dans chaque entrée du tableau, une description de la valeur initiale Scheme est inscrite comme valeur initiale C. A l'initialisation, le noyau traduit les valeurs initiales C en des valeurs initiales Scheme. Ainsi, l'ensemble des descriptions ne prend pas plus d'espace que les variables elles-mêmes et ces descriptions sont écrasées par la traduction.

La description de la valeur initiale d'une variable globale est codée de façon simple. Si la valeur initiale est le booléen `#f`, elle est codée par -1. Si la valeur initiale est la fonction du noyau numéro n, elle est codée par $-2-n$. Si la valeur initiale est une fermeture dont le corps se trouve à l'adresse n dans le code-octets, elle est codée par n.

On sait que les mots machine du micro-contrôleur sont de 16 bits seulement. On peut donc se demander si l'adresse d'une fermeture pourrait être tellement élévée qu'elle causerait un débordement. Ce n'est toutefois pas possible car le code-octets ne peut dépasser 32 kilo-octets. Le compilateur s'assure que cette restriction est respectée. La restriction est imposée par la représentation des continuations (voir à la section 3.3). Le point de retour de la continuation est stocké dans le premier champ, lequel est partagé avec le bit de tag indiquant le sous-type propre aux continuations. Ainsi, il n'y a que 15 bits pour stocker le point de retour dans les continuations, ce qui contraint la taille du code-octets à ne pas dépasser 32 Koctets.

5.8 Substitution des fonctions appelées

Nous parlons ici d'une opération faite lors de la deuxième traversée des arbres de syntaxe du programme. Cette opération vise à remplacer un appel à une fonction connue par un appel à une fonction plus simple. Cette opération permet principalement d'augmenter l'efficacité de l'interprétation du programme. Des économies d'espace peuvent aussi être réalisées, car l'appel d'une fonction du noyau connue est plus concis que l'appel d'une fermeture connue.

En Scheme, il y a plusieurs fonctions qui acceptent un nombre variable d'arguments. Par exemple, la fonction `+` additionne aucun nombre ou plus. La fonction `<` compare deux nombres ou plus. La fonction `map` applique une fonction à une liste ou plus. La plupart de ces fonctions sont souvent utilisées avec le même nombre d'arguments. Par exemple, on additionne ou on compare habituellement deux nombres. On applique souvent une fonction à une seule liste.

Tel que demandé par le standard, ces fonctions sont implantées avec tout le protocole nécessaire à la gestion du nombre variable d'arguments. Le besoin de compacité exige d'adopter un traitement régulier plutôt que d'écrire plusieurs spécialisations de ces fonctions. Reprenons avec les mêmes exemples: la fonction `+` est implantée avec la fonction binaire `math+2` et la fonction générale `foldl`; la fonction `<` est implantée avec l'opérateur `math<2` et la fonction générale `generic-compare`; la fonction `map` est implantée à l'aide de la fonction `map1` qui applique une fonction aux éléments d'une seule liste. Pour voir l'implantation complète de ces fonctions, voir la section A.2.

Un exemple de substitution consisterait à remplacer un appel à la fonction `+` par un appel à la fonction `math+2`.

Pour permettre une substitution de fonctions, il y a plusieurs conditions à remplir. Premièrement, c'est seulement dans les appels de fonctions que le compilateur effectue des substitutions. Deuxièmement, on doit savoir statiquement quelle est la fonction qui est appelée. Troisièmement, cette fonction doit posséder une fonction substituante plus simple. Quatrièmement, le nombre d'arguments passés doit correspondre au nombre de paramètres de la fonction substituante. Si toutes ces conditions sont remplies, la substitution peut avoir lieu.

Voici la liste des substitutions que notre compilateur peut faire.

- Remplacer la fonction `append` par `append2` quand elle est appliquée sur deux arguments.
- Remplacer les fonctions `=`, `<`, `>`, `<=` et `>=` par les fonctions de comparaison analogues à deux arguments.
- Remplacer les fonctions `max`, `min`, `+`, `*`, `-`, `/`, `gcd` et `lcm` par les fonctions binaires correspondantes.
- Remplacer les fonctions `make-string` et `make-vector` par des fonctions ne prenant que la longueur en paramètre.
- Remplacer la fonction `apply` par une fonction analogue à deux paramètres.

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

FIG. 5.4 - *Implantation de la fonction de Fibonacci utilisée pour un test de performance*

– Remplacer la fonction `map` par la fonction `map1` qui applique une fonction sur les éléments d'une seule liste.

A l'exception du cas de la fonction `map`, les fonctions substituantes sont justement employées de façon interne pour l'implantation des fonctions substituées.

Ces substitutions de fonctions permettent réellement d'améliorer les performances. Nous avons pu le constater sur un programme calculant le 20ème nombre de Fibonacci (voir la figure 5.4). Ce programme utilise constamment le genre de fonctions qui peuvent être substituées. Ses performances en temps sont améliorées par un facteur 7 grâce à la substitution.

5.9 Production du code-octets

Nous décrivons la méthode que nous employons pour produire le code-octets. Toutefois, ce n'est pas ici que nous présentons le jeu d'instructions de la machine virtuelle, pas plus que nous ne montrons les instructions générées pour chaque forme syntaxique. Ces points sont passablement vastes et sont traités dans le prochain chapitre.

Nous présentons les items suivants: l'organisation globale du code-octets; l'organisation arborescente du code directement produit par la compilation des arbres de syntaxe; la linéarisation de l'arborescence; la résolution des références aux étiquettes.

5.9.1 Organisation globale du code-octets

Comme les variables introduites par la librairie sont initialisées implicitement, il n'est pas nécessaire de les initialiser explicitement avant l'exécution du programme et c'est le code des expressions de ce dernier qui occupe le début du code-octets. Les quelques expressions ajoutées au début du programme par le compilateur s'occupent de faire les initialisations nécessaires. Parmi ces expressions figurent celles qui conservent les fonctions `memv`, `make-promise`, ..., dans des variables à nom unique.

L'exécution débute donc à l'adresse 0 du code-octets. Les expressions du premier niveau ont leur code disposé dans l'ordre. Une instruction d'arrêt du programme les suit. Ensuite, se trouve le code du corps de chacune des fermetures de la librairie.

5.9.2 Compilation des formes syntaxiques

Le code-octets n'est pas généré immédiatement sous sa forme finale. Nous nous permettons de le générer sous une forme arborescente calquée sur la structure des arbres de syntaxe. Nous utilisons de plus un système d'étiquettage pour les références à des adresses qui ne sont pas encore connues.

Le code brut produit par la compilation d'un noeud consiste en une liste pouvant contenir des "octets", des définitions d'étiquettes, des références à des étiquettes et d'autres codes bruts. Les "octets" sont des nombres entre 0 et 255 inclusivement, qui sont des instructions complètes ou des morceaux, et qui sont conservés jusqu'au code-octets final. Les définitions d'étiquettes sont composées du symbole "`def`" suivi d'un numéro d'étiquette. Les références aux étiquettes sont le symbole "`ref`" suivi d'un numéro d'étiquette.

A titre d'exemple, le résultat de la compilation d'un noeud `if` est une liste contenant (pas dans cet ordre-ci): des sous-listes représentant le test, le conséquent et l'alternative; des octets représentant un saut conditionnel; l'étiquette et la référence nécessaires au saut conditionnel; etc.

5.9.3 Linéarisation et résolution des références

Cette dernière étape permet de transformer le code qui est sous forme arborescente et qui est pourvu d'étiquettes en un code linéaire où toutes les références sont résolues.

La linéarisation de l'arborescence consiste simplement à ramener l'arborescence en une seule liste contenant octets, symboles et numéros.

La résolution des références se fait en deux passes. La première détermine la position de chaque étiquette. La seconde remplace les références aux étiquettes par l'adresse des étiquettes et élimine les définitions d'étiquettes. Les adresses ainsi obtenues sont codées en deux octets.

A ce point, le code-octets est représenté par une liste d'octets. Il ne reste qu'à l'écrire sous forme de tableau C.

Chapitre 6

La machine virtuelle et le code-octets

La seule partie de l'interprète de code-octets qu'il reste à décrire est la machine virtuelle. En fait, il y a deux machines virtuelles. La première est celle qui est utilisée dans la première version compilée. La seconde est dérivée de la première et permet de coder les programmes de façon beaucoup plus compacte. Mais elle comporte plus d'instructions et plusieurs d'entre elles sont des amalgames de plusieurs instructions. Ce qui fait en sorte que la seconde machine virtuelle est passablement plus complexe que la première.

Nous commençons par décrire la première machine. Nous montrons comment la compilation s'effectue à l'aide du jeu d'instructions de cette machine. Une description sommaire de la seconde machine suit. Nous nous intéressons surtout aux transformations qui sont apportées à la première machine afin d'obtenir du code compact. Enfin, nous illustrons les gains d'espace obtenus avec le passage de la première à la seconde machine. Nous préférons illustrer la compilation à l'aide de la première machine car elle est plus simple que la seconde.

6.1 La première machine virtuelle

La première machine virtuelle vise la simplicité. Elle dispose de quelques registres. Ses quelques instructions permettent la compilation de toutes les formes syntaxiques de base.

6.1.1 Les registres

Les registres de la machine sont les suivants. Le registre PC contient l'adresse d'exécution courante dans le code-octets. Le registre VAL est l'accumulateur de la machine. Le registre ENV

contient l'environnement d'évaluation courant. Le registre ARGS contient la liste d'arguments en construction. Le registre PREV_ARGS contient une pile des listes d'arguments en construction. Le registre CONT contient la continuation courante.

La machine a accès à d'autres registres qui sont utilisés moins régulièrement. Ceux-ci servent à contenir quelques constantes couramment utilisées, le vecteur des noms des symboles et le vecteur des constantes allouées.

6.1.2 Comportement général

Nous faisons un survol de la façon dont les expressions sont exécutées dans la machine virtuelle. La machine virtuelle n'exécute pas vraiment les expressions directement, elle suit simplement les étapes décrites par les instructions générées par le compilateur.

L'évaluation d'une expression est effectuée différemment selon qu'elle se trouve en position terminale ou non-terminale. Dans tous les cas, l'évaluation commence en ayant le registre PC qui pointe au début du code et le registre ENV qui contient l'environnement courant. Les différences se trouvent dans la façon de terminer l'évaluation de l'expression.

Si l'expression est non-terminale, elle retourne son résultat et préserve l'état en vigueur avant son évaluation. Plus précisément, l'évaluation de l'expression a les effets suivants sur les registres. Le registre VAL contient le résultat de l'évaluation. Le registre PC pointe sur l'instruction qui suit le code de l'expression. Les autres registres sont inchangés et, ce, peu importe les modifications qu'ils ont pu subir entre temps.

Si l'expression est terminale, elle retourne son résultat à la continuation courante. Donc, le registre VAL contient le résultat de l'évaluation. Tous les autres registres sauf PREV_ARGS sont extraits de la continuation courante. Le registre PREV_ARGS doit rester inchangé. Une expression en position terminale peut donc modifier la plupart des registres comme elle le souhaite avant de retourner son résultat.

Une sauvegarde de l'état est habituellement faite en créant une continuation. La structure de données que nous utilisons sert à sauver la valeur des registres PC, ENV, ARGS et CONT. C'est le fait de sauvegarder le registre CONT dans les nouvelles continuations qui crée une chaîne de structures.

Dans certains cas, une sauvegarde doit être faite, mais seulement du registre ARGS. Dans les exemples de compilation, nous voyons que l'appel d'une fonction du noyau connue ne cause pas d'autre modification de l'état que l'écrasement du registre ARGS. La sauvegarde qui est effectuée dans de tels cas consiste à ajouter le contenu de ARGS devant le contenu de PREV_ARGS. L'ajout se fait à l'aide d'une paire. Une sauvegarde avec une paire dépense moins d'espace qu'une sauvegarde avec une continuation. Lorsque c'est possible, il est préférable de faire une sauvegarde dans PREV_ARGS.

En quelque sorte, ces deux formes de sauvegarde se partagent l'information contenue dans la

continuation conceptuelle. Aussi, lorsque le programme capture la continuation courante, il ne suffit pas de faire une sauvegarde du premier genre et de retourner la structure résultante. Il faut plutôt sauver la continuation résultante dans une paire avec le contenu du registre PREV_ARGS. De cette façon, la paire contient toute l'information qu'une continuation conceptuelle doit contenir. Il est à noter que le registre VAL n'est jamais sauvé: les continuations conceptuelles représentent le reste du calcul, elle ne connaissent pas le résultat du calcul courant.

Nous mentionnons à la section 2.4.2 qu'un interprète ne crée pas nécessairement toutes les continuations qui entrent en jeu dans une évaluation. C'est le cas de notre interprète. De façon générale, il ne crée une continuation que lorsqu'il doit sauver son état.

6.1.3 Opérandes courts et longs

Dans le jeu d'intructions de la machine virtuelle, certaines instructions sont en deux versions. Une version prend un opérande d'un octet et l'autre version prend un opérande de deux octets.

Par exemple, l'instruction qui sert à lire une variable globale est en deux versions. Il est clair qu'un programme peut comporter plus que 256 variables globales. Mais, dans bien des cas, leur nombre est inférieur ou dépasse de peu 256. C'est pourquoi, par généralité et par souci de compacité, il est utile d'avoir des versions longue et courte de cette instruction.

Un tel choix rend l'implantation de la machine virtuelle plus volumineuse, mais nous en décidons ainsi seulement lorsque la version courte est utilisée fréquemment.

6.1.4 Opérande facultatif

Les instructions de lecture et d'écriture aux variables lexicales ont deux opérandes, mais le second peut (et doit) être omis dans certaines situations. Le premier opérande indique le nombre de blocs de liaison à sauter pour parvenir au bloc adéquat. Le second indique la position de la variable dans le bloc. (La description des environnements est donnée à la section 3.4.3.)

Or, on sait que certains blocs ne contiennent qu'une seule variable. Ces blocs sont représentés par des paires. La machine, en exécutant une de ces instructions, commence par se rendre au bloc approprié, vérifie s'il s'agit d'un bloc à plusieurs variables (un vecteur) et, le cas échéant, lit le prochain opérande. S'il s'agit d'un bloc à une variable, l'opérande n'est pas requis et le compilateur ne l'a pas généré.

6.1.5 Le jeu d'instructions

Afin de ne pas diluer les remarques importantes dans trop de répétitions, nous ne donnons pas nécessairement une description de chaque instruction séparément.

0 `GetImmCte` ⟨description⟩: Obtention d'une constante non allouée. L'opérande est une "description" de la constante en question. La description indique le type et, si nécessaire, la valeur de la donnée. Le résultat est mis dans VAL.

1 `GetAllocCte` ⟨number⟩: Obtention d'une constante allouée. L'opérande est le numéro (sur 16 bits) de la constante. La constante est tirée du vecteur des constantes. Le résultat est mis dans VAL.

2-5 `ReadVarXX` ⟨operand_1⟩ [⟨operand_2⟩]: Lecture d'une variable lexicale ou globale. Ces instructions sont en versions courtes et longues. Le(s) opérande(s) indique(nt) la position de la variable. Le résultat est mis dans VAL. Le suffixe "`XX`" de la mnémonique indique seulement que cette mnémonique regroupe toutes les variantes de l'instruction.

6-9 `WriteVarXX` ⟨operand_1⟩ [⟨operand_2⟩]: Ecriture d'une variable lexicale ou globale. Ce sont les instructions d'écriture analogues à celles de lecture. La valeur à écrire est prise dans VAL.

10 `MakeClos`: Création d'une fermeture. Une fermeture est créée d'après les valeurs de PC et ENV et est mise dans VAL. Le point d'entrée indiqué dans la fermeture est 3+PC. Nous expliquons ce calcul dans l'exemple de la compilation des lambda-expressions.

11 `CJump` ⟨address⟩: Saut conditionnel. Si VAL contient la valeur `#f`, un saut est effectué à l'adresse indiquée par l'opérande.

12 `Jump` ⟨address⟩: Saut inconditionnel. Un saut est effectué à l'adresse indiquée par l'opérande.

13 `SaveCont` ⟨address⟩: Sauvegarde de l'état de la machine dans une continuation. Le point de retour est indiqué par l'opérande. La continuation est mise dans le registre CONT.

14 `RestoreCont`: Rétablissement de l'état de la machine à partir de la continuation contenue dans CONT.

15 `InitArgs`: Initialisation de la liste d'arguments. La liste vide est mise dans ARGS.

16 `PushArg`: Empilement d'un argument. La valeur contenue dans VAL est ajoutée à liste contenue dans ARGS.

17 `Apply`: Application d'une fonction. La liste dans ARGS contient une fonction en tête et des arguments par la suite. Cette fonction est appliquée sur les arguments. L'exécution reprend plus tard au point désigné par la continuation courante et avec le résultat contenu dans VAL.

18 `ApplyKernel` ⟨numéro⟩: Application d'une fonction du noyau. La fonction du noyau dont le numéro est indiqué par un opérande d'un octet est appliquée sur les arguments stockés dans ARGS. L'exécution reprend immédiatement avec le résultat dans VAL.

19 `FlushEnv`: Vider l'environnement. La liste vide est mise dans ENV.

20-23 `MakeBlockXX` ⟨size⟩: Créer un bloc de liaison avec ou sans paramètre *reste*. Ces instructions sont en versions courtes et longues. En Scheme, un paramètre *reste* est un paramètre qui reçoit une liste contenant tous les arguments suivant les arguments requis. L'opérande indique la taille du bloc d'activation à créer. Les valeurs sont prises dans ARGS et le nouveau bloc est ajouté devant l'environnement contenu dans ENV.

24 `Stop`: Terminer l'exécution du programme. L'interprète cesse ses opérations.

25 `SaveArgs`: Sauvegarde de la liste d'arguments en construction. La liste d'arguments contenue dans ARGS est ajoutée au début de la liste contenue dans PREV_ARGS.

26 `RestoreArgs`: Rétablissement de la liste d'arguments précédente. La première liste sauvée dans la liste de PREV_ARGS est retirée et placée dans ARGS.

Les exemples de compilation illustrent l'utilité de chacune de ces instructions.

6.2 Compilation des différentes formes

Nous montrons la façon de compiler plusieurs des formes syntaxiques de base. Afin de condenser l'écriture du code compilé, nous utilisons une notation mathématique symbolisant la compilation.

$\mathcal{C}[\![$ ⟨exp⟩ $]\!]$ représente le code généré pour l'expression ⟨exp⟩ si celle-ci se trouve en position non-terminale.

$\mathcal{C}^*[\![$ ⟨exp⟩ $]\!]$ représente le code généré pour l'expression ⟨exp⟩ si celle-ci se trouve en position terminale.

$\mathcal{C}^?[\![$ ⟨exp⟩ $]\!]$ représente le code généré pour l'expression ⟨exp⟩. Le point d'interrogation indique indifféremment une des deux positions. Celles de ses sous-expressions qui sont compilées avec un point d'interrogation sont compilées dans la même position.

Nous utilisons aussi une pseudo-mnémonique `DefLabel` qui ne correspond pas à une instruction mais sert plutôt à définir une étiquette.

6.2.1 Constantes

$\mathcal{C}[\![$ ⟨Constant⟩ $]\!] =$

- `GetXXCte` ⟨operand⟩

$\mathcal{C}^*[\![$ ⟨Constant⟩ $]\!] =$

- `GetXXCte` ⟨operand⟩
- `RestoreCont`

Pour simplifier l'exemple, nous ne distinguons pas expressément entre les constantes allouées et non-allouées.

6.2.2 Références de variables

$\mathcal{C}[\![$ ⟨variable⟩ $]\!] =$

- `ReadVarXX` ⟨operand$_1$⟩ [⟨operand$_2$⟩]

$\mathcal{C}^*[\![$ ⟨variable⟩ $]\!] =$

- `ReadVarXX` ⟨operand$_1$⟩ [⟨operand$_2$⟩]
- `RestoreCont`

On utilise une des instructions de lecture de variable. Dans le cas d'une référence à une variable globale, l'opérande est son numéro. Dans le cas d'une référence à une variable lexicale, les deux opérandes désignent le nombre de blocs à sauter et la position dans le bloc atteint. Le dernier opérande doit être omis si le compilateur détecte qu'il s'agit d'un bloc à une variable.

6.2.3 Affectations et définitions

Ces deux formes syntaxiques sont compilées presque de la même façon. La seule différence vient du fait qu'un noeud de définition ne peut concerner qu'une variable globale. Sans perte de généralité, nous faisons l'exemple sur une affectation.

$\mathcal{C}[\![$ `(set!` ⟨variable⟩ ⟨exp⟩`)` $]\!] =$

- $\mathcal{C}[\![$ ⟨exp⟩ $]\!]$

– `WriteVarXX` $\langle \text{operand}_1 \rangle$ [$\langle \text{operand}_2 \rangle$]

$\mathcal{C}^*[\![$ `(set!` $\langle \text{variable} \rangle$ $\langle \text{exp} \rangle$`)` $]\!] =$

– $\mathcal{C}[\![\langle \text{exp} \rangle]\!]$

– `WriteVarXX` $\langle \text{operand}_1 \rangle$ [$\langle \text{operand}_2 \rangle$]

– `RestoreCont`

6.2.4 Blocs d'expressions

$\mathcal{C}^?[\![$ `(begin` $\langle \text{exp}_1 \rangle \ldots \langle \text{exp}_{n-1} \rangle$ $\langle \text{exp}_n \rangle$`)` $]\!] =$

– $\mathcal{C}[\![\langle \text{exp}_1 \rangle]\!]$

– ...

– $\mathcal{C}[\![\langle \text{exp}_{n-1} \rangle]\!]$

– $\mathcal{C}^?[\![\langle \text{exp}_n \rangle]\!]$

La très grande simplicité de la compilation des blocs est une des raisons justifiant leur adoption comme forme de base dans notre compilateur.

6.2.5 Conditions

$\mathcal{C}[\![$ `(if` $\langle \text{exp}_1 \rangle$ $\langle \text{exp}_2 \rangle$ $\langle \text{exp}_3 \rangle$`)` $]\!] =$

– $\mathcal{C}[\![\langle \text{exp}_1 \rangle]\!]$

– `CJump` $\langle \text{label}_1 \rangle$

– $\mathcal{C}[\![\langle \text{exp}_2 \rangle]\!]$

– `Jump` $\langle \text{label}_2 \rangle$

– `DefLabel` $\langle \text{label}_1 \rangle$

– $\mathcal{C}[\![\langle \text{exp}_3 \rangle]\!]$

– `DefLabel` $\langle \text{label}_2 \rangle$

$\mathcal{C}^*[\![$ `(if` $\langle \text{exp}_1 \rangle$ $\langle \text{exp}_2 \rangle$ $\langle \text{exp}_3 \rangle$`)` $]\!] =$

- $\mathcal{C}[\![$ $\langle \text{exp}_1 \rangle$ $]\!]$
- `CJump` $\langle$label$\rangle$
- $\mathcal{C}^*[\![$ $\langle \text{exp}_2 \rangle$ $]\!]$
- `DefLabel` $\langle$label$\rangle$
- $\mathcal{C}^*[\![$ $\langle \text{exp}_3 \rangle$ $]\!]$

Quand l'expression est en position terminale, ses deux dernières sous-expressions le sont aussi et le déroulement de l'exécution ne dépasse pas la fin du code de celles-ci. C'est pourquoi il y a un saut en moins que dans le cas où l'expression est en position non-terminale.

6.2.6 Lambda-expressions

La compilation d'une lambda-expression produit du code destiné à deux usages distincts. Premièrement, il y a le code lié à l'évaluation de la lambda-expression, c'est-à-dire à la création de la fermeture. Deuxièmement, il y a le code lié au corps de la fermeture. Voici ce qui est généré:

$\mathcal{C}[\![$ `(lambda` $\langle$formals$\rangle$ $\langle$body$\rangle$`)` $]\!] =$

- `MakeClos`
- `Jump` $\langle$label$\rangle$
- `MakeBlockXX` $\langle$size$\rangle$
- $\mathcal{C}^*[\![$ $\langle$body$\rangle$ $]\!]$
- `DefLabel` $\langle$label$\rangle$

Il y a quelques détails qu'il faut expliquer.

Premièrement, l'instruction `MakeBlockXX` n'est pas générée s'il n'y a pas de paramètres formels.

Deuxièmement, grâce à l'exemple, on peut comprendre pourquoi une instruction de création de fermeture fixe le point d'entrée à 3+PC. C'est que *durant* l'exécution de cette instruction, PC pointe sur l'instruction de saut, qui occupe 3 octets.

Troisièmement, lors de l'invocation d'une fermeture, l'environnement de définition est mis dans ENV et l'adresse du corps est mise dans PC. De plus, la liste des arguments est dans ARGS.

Ainsi, l'instruction de création de bloc prend les arguments et en fait un bloc de liaison qu'elle ajoute devant l'environnement de définition.

Une lambda-expression en position terminale génère un code presqu'identique à celui du cas précédent. Seule une instruction de rétablissement d'état est ajoutée:

$\mathcal{C}^*[\![$ `(lambda` $\langle$formals$\rangle$ $\langle$body$\rangle$`)` $]\!] =$

- $\mathcal{C}[\![$ `(lambda` $\langle$formals$\rangle$ $\langle$body$\rangle$`)` $]\!]$
- `RestoreCont`

6.2.7 Appels de fonctions

La compilation de ces formes est nettement la plus difficile. C'est principalement parce que nous traitons plusieurs sortes d'appels, en fonction des informations statiques que le compilateur arrive à calculer.

Avant de montrer comment compiler chaque type d'appel, nous montrons comment compiler la liste des expressions fournissant les arguments de l'appel. Cette liste peut contenir ou non l'expression qui retourne la fonction à invoquer. Ensuite, nous voyons les différentes formes d'appel. La fonction à invoquer dans un appel de fonction peut être soit inconnue, soit une fonction du noyau, soit une fermeture de la librairie ou, encore, une lambda-expression. De plus, l'appel est compilé différemment suivant qu'il est en position terminale ou non.

Compilation de la liste d'arguments

La liste des expressions dans un appel est compilée dans l'ordre inversé. C'est-à-dire que la dernière expression est la première à être évaluée. Cet ordre permet de séparer plus facilement les arguments qui sont requis de ceux qui vont dans un éventuel paramètre reste.

La fonction $\mathcal{C}_A[\![\ .\ ]\!]$ compile les listes d'arguments. La fonction $\mathcal{C}_{A'}[\![\ .\ ]\!]$ est une fonction auxilliaire.

$\mathcal{C}_A[\![\ \langle \mathrm{exp}_1 \rangle, \ldots, \langle \mathrm{exp}_n \rangle\ ]\!] =$

- `InitArgs`
- $\mathcal{C}_{A'}[\![\ \langle \mathrm{exp}_1 \rangle, \ldots, \langle \mathrm{exp}_n \rangle\ ]\!]$

$\mathcal{C}_{A'}[\![\ \varepsilon\ ]\!] =$

- *(aucune instruction)*

$\mathcal{C}_{A'}[\![\ \langle \mathrm{exp}_1 \rangle, \ldots, \langle \mathrm{exp}_{n-1} \rangle, \langle \mathrm{exp}_n \rangle\ ]\!] =$

- $\mathcal{C}[\![\ \langle \mathrm{exp}_n \rangle\ ]\!]$
- `PushArg`
- $\mathcal{C}_{A'}[\![\ \langle \mathrm{exp}_1 \rangle, \ldots, \langle \mathrm{exp}_{n-1} \rangle\ ]\!]$

On peut remarquer que toutes les expressions d'une liste d'arguments sont en position non-terminale et sont compilées comme tel.

La fonction à invoquer est inconnue

$\mathcal{C}[\![\ (\langle \mathrm{exp}_1 \rangle \ldots \langle \mathrm{exp}_n \rangle)\ ]\!] =$

- `SaveCont` $\langle \mathrm{label} \rangle$
- $\mathcal{C}_A[\![\ \langle \mathrm{exp}_1 \rangle, \ldots, \langle \mathrm{exp}_n \rangle\ ]\!]$
- `Apply`
- `DefLabel` $\langle \mathrm{label} \rangle$

$\mathcal{C}^*[\![\ (\langle \mathrm{exp}_1 \rangle \ldots \langle \mathrm{exp}_n \rangle)\ ]\!] =$

- $\mathcal{C}_A[\![\ \langle \mathrm{exp}_1 \rangle, \ldots, \langle \mathrm{exp}_n \rangle\ ]\!]$
- `Apply`

Lorsque l'appel est en position terminale, son évaluation écrase le contenu de ARGS et de ENV. Ca ne pose pas de problème puisque l'état est rétabli à la fin de l'appel.

La fonction à invoquer est une fonction du noyau

Dans un tel cas, le protocole d'appel est normalement moins lourd puisqu'il n'est pas nécessaire de calculer l'expression en position appelante.

$\mathcal{C}[\![\ (\langle \mathrm{kernelfunction} \rangle\ \langle \mathrm{exp}_1 \rangle \ldots \langle \mathrm{exp}_n \rangle)\ ]\!] =$

- `SaveArgs`
- $\mathcal{C}_A[\![\ \langle \mathrm{exp}_1 \rangle, \ldots, \langle \mathrm{exp}_n \rangle\ ]\!]$

- `ApplyKernel` ⟨kernelfunction⟩
- `RestoreArgs`

$\mathcal{C}^*[\![$ `(`⟨kernelfunction⟩ ⟨$\exp_1$⟩ ... ⟨$\exp_n$⟩`)` $]\!] =$

- $\mathcal{C}_A[\![$ ⟨$\exp_1$⟩, ..., ⟨$\exp_n$⟩ $]\!]$
- `ApplyKernel` ⟨kernelfunction⟩
- `RestoreCont`

Il y a toutefois une exception: lorsque la fonction à invoquer est `apply1`. Cette fonction du noyau à deux arguments sert à implanter `apply`. `Apply1` est la seule fonction du noyau qui risque de modifier l'état de la machine virtuelle. En effet, elle lance l'interprète sur un calcul inconnu *a priori*.

Lorsque la fonction est `apply1`, la compilation de l'appel se fait comme si la fonction à invoquer était inconnue.

La fonction à invoquer est une fermeture de la librairie

Rappellons nous qu'une fermeture de la librairie a un environnement de définition vide. L'adresse du corps de la fermeture est connue.

$\mathcal{C}[\![$ `(`⟨bodyaddress⟩ ⟨$\exp_1$⟩ ... ⟨$\exp_n$⟩`)` $]\!] =$

- `SaveCont` ⟨label⟩
- $\mathcal{C}_A[\![$ ⟨$\exp_1$⟩, ..., ⟨$\exp_n$⟩ $]\!]$
- `FlushEnv`
- `Jump` ⟨bodyaddress⟩
- `DefLabel` ⟨label⟩

$\mathcal{C}^*[\![$ `(`⟨bodyaddress⟩ ⟨$\exp_1$⟩ ... ⟨$\exp_n$⟩`)` $]\!] =$

- $\mathcal{C}_A[\![$ ⟨$\exp_1$⟩, ..., ⟨$\exp_n$⟩ $]\!]$
- `FlushEnv`
- `Jump` ⟨bodyaddress⟩

Même si l'application en tant que telle est plus longue (en octets) que l'application générale, le fait de ne pas calculer l'expression en position appelante cause une économie d'espace.

La fonction à invoquer est produite par une lambda-expression

Une telle construction est rare dans un source Scheme écrit par un programmeur. Mais elle est très fréquente parmi les expressions réduites. Toutes les formes de liaison produisent ce type de construction.

Nous avons décidé de traiter ce cas explicitement plutôt que de laisser générer du code qui fait créer une fermeture, qui l'applique et qui l'abandonne tout de suite après. De plus, le code généré est plus compact.

$\mathcal{C}[\![$ `((lambda` $\langle$formals$\rangle$ $\langle$body$\rangle$`)` $\langle$exp$_1\rangle$ ... $\langle$exp$_n\rangle$`)` $]\!] =$

- `SaveCont` $\langle$label$\rangle$
- $\mathcal{C}_A[\![$ $\langle$exp$_1\rangle$, ..., $\langle$exp$_n\rangle$ $]\!]$
- `MakeBlockXX` $\langle$size$\rangle$
- $\mathcal{C}^*[\![$ $\langle$body$\rangle$ $]\!]$
- `DefLabel` $\langle$label$\rangle$

$\mathcal{C}^*[\![$ `((lambda` $\langle$formals$\rangle$ $\langle$body$\rangle$`)` $\langle$exp$_1\rangle$ ... $\langle$exp$_n\rangle$`)` $]\!] =$

- $\mathcal{C}_A[\![$ $\langle$exp$_1\rangle$, ..., $\langle$exp$_n\rangle$ $]\!]$
- `MakeBlockXX` $\langle$size$\rangle$
- $\mathcal{C}^*[\![$ $\langle$body$\rangle$ $]\!]$

L'instruction `MakeBlockXX` est toujours nécessaire, contrairement au cas de la compilation des lambda-expressions. Une lambda-expression en position appelante possède toujours des paramètres. Sinon, l'uniformisation des formes de base (section 5.2.1) l'aurait éliminée plus tôt.

6.2.8 Qualité de la compilation

La méthode de compilation que nous venons de décrire n'est pas parfaite. Si on est attentif, on peut se rendre compte que plusieurs formes pourraient être mieux compilées.

Pour ne citer qu'un exemple, la compilation des lambda-expressions n'est pas très bonne. Le code du corps est généré sur place et il faut faire un saut par-dessus une fois que l'on a créé la fermeture. Si la lambda-expression est en position terminale, le code généré comporte un autre défaut. Le saut est fait par-dessus le corps afin de systématiquement tomber sur une instruction `RestoreCont`.

On peut trouver plusieurs autres défauts. Il faudrait de nouvelles instructions pour pallier à la plupart des défauts. C'est pourquoi la machine virtuelle est modifiée à partir de la version suivante de l'interprète.

6.3 La deuxième machine virtuelle

Cette machine est plus complexe et résulte de nombreux ajouts et modifications effectués sur la première machine. Nous ne présentons pas son jeu d'instructions car il est long et très ennuyeux. Nous ne tenons pas non plus à donner d'exemples de compilation pour les différentes formes syntaxiques. Nous voulons plutôt discuter de certains des changements apportés à la première machine qui ont permis d'aboutir à la deuxième.

6.3.1 Instructions spécialisées

Plusieurs instructions spécialisées sont ajoutées. Elles consistent en une version courte de certaines instructions utilisées fréquemment.

Il y a des instructions servant à retourner la liste vide, chaque booléen, un caractère ou un nombre représentable sur un octet. Une constante construite dont le numéro est inférieur à 256 (chose très fréquente) peut être obtenue grâce à une instruction courte.

Les lectures de variables lexicales sont fréquentes, surtout en certaines coordonnées. Nous avons créé 6 instructions spécialisées pour les coordonnées les plus utilisées. En notation `(sauts de blocs, position)`, les cas les plus fréquents sont, dans l'ordre, `(0, 0)`, `(0, 1)`, `(1, 0)`, `(2, 0)`, `(0, 2)` et `(1, 1)`. Respectivement, ces cas représentent 52%, 17%, 10%, 5%, 3% et 3% des lectures de variables lexicales. Donc, au total, environ 90% de ces lectures. Le code utilisé pour amasser ces statistiques est un ensemble de fonctions de la librairie.

Les instructions de fabrication de blocs d'activation et de création de fermeture ont aussi des versions spécialisées pour quelques cas.

6.3.2 Fusion d'instructions

La compilation vers le code de la première machine place souvent certaines instructions avant ou après certaines autres. Plusieurs instructions ont été créées pour remplacer deux anciennes instructions, ou plus.

L'exemple suivant montre des instructions qui se suivent. Toutes les fois où l'état de la machine est sauvée dans une continuation (instruction `SaveCont`), c'est parce qu'il y a un appel qui doit être fait. Or, la première étape d'un appel consiste toujours à initialiser le registre ARGS (instruction `InitArgs`). Nous avons donc remplacé l'instruction `SaveCont` par une instruction

amalgamée `SaveCont-InitArgs`.

6.3.3 Diverses nouvelles instructions

Quelques exemples de fonctions diverses sont présentés.

Une nouvelle instruction permet d'éliminer le premier bloc de liaison dans l'environnement. Ainsi, il n'est plus nécessaire de créer une continuation pour restaurer l'environnement après l'exécution d'une forme de liaison en position non-terminale. Une forme de liaison réduite et en position non-terminale (fin de la section 6.2.7) ne nécessite plus une sauvegarde de l'état dans une continuation. L'environnement initial peut être retrouvé grâce à la nouvelle instruction.

Lorsque l'état est sauvé avant de faire un appel, le point de retour qui est spécifié est toujours l'adresse qui suit l'instruction `Apply`. Le protocole est légèrement changé. L'instruction `SaveCont` ne spécifie plus de point de retour. C'est l'instruction `Apply` qui fixe le point de retour avant d'effectuer l'application comme telle. Il n'y a donc plus de point de retour à indiquer explicitement pour ce type d'appels.

Une série d'instructions d'un octet ont été ajoutées. Une par fonction du noyau. L'invocation directe d'une fonction du noyau prend maintenant un octet de moins. Cette série d'instructions ne complique pratiquement pas l'implantation de la machine virtuelle. En effet, il suffit de donner des numéros consécutifs aux instructions de la série et l'implantation n'a pas à traiter chacune en particulier. Nous avons donné le numéro $255 - n$ à l'instruction qui invoque la fonction numéro n du noyau.

6.3.4 Empilement automatique

L'instruction qui est la plus fréquente est `PushArg`, qui cause l'empilement dans ARGS de la valeur contenue dans VAL. Maintenant, l'empilement est automatique à la suite de l'évaluation des expressions. En fait, l'empilement est intégré aux instructions qui terminent normalement le code d'une expression.

Lorsqu'un empilement n'est pas voulu, ce qui est le cas le plus rare, un dépilement explicite est généré. Un empilement n'est pas désiré, par exemple, après l'exécution des $n - 1$ premières expressions d'une expression `begin` qui en contient n. En effet, seul le résultat de la dernière expression doit être conservé.

6.4 Importance de la définition des instructions

Le passage de la première machine virtuelle à la seconde permet de réduire la taille du code-octets de façon appréciable. Nous utilisons deux programmes Scheme afin de faire les tests.

Le premier programme est minuscule. Il ne comporte qu'une expression. Celle-ci crée une liste de toutes les fonctions visibles de la librairie. Ceci cause l'inclusion de toutes ces fonctions. En effet, le compilateur ne fait pas d'analyses assez poussées pour déterminer que seule la fonction `list` est *effectivement* utilisée. Il est important de mentionner que le fichier source de la librairie est un fichier de 23 kilo-octets.

Le second programme est un module d'un programme réel: un générateur d'analyseur lexical. Le module ne cause pas l'inclusion de toutes les fonctions de la librairie, mais c'est un fichier d'environ 15 500 octets (environ 560 lignes) avec peu de commentaires.

Les deux "programmes" produisent chacun près de 10 500 octets, quand ils sont compilés avec le jeu d'instructions de la première machine. Tandis qu'avec le jeu d'instruction de la seconde machine, ils produisent près de 5400 et 5600 octets respectivement. Il s'agit donc d'une nette amélioration.

On peut faire deux observations à partir de ces chiffres. Premièrement, la librairie au grand complet n'occupe pas plus de 5400 octets, ce qui est très peu d'espace. Deuxièmement, il est possible de compiler des programmes relativement imposants et de les faire entrer sur le micro-contrôleur. Toutefois, les programmes doivent être assez économes en consommation de mémoire: dans tous les cas, le tas ne peut contenir qu'un petit nombre d'objets. Par exemple, un minuscule programme qui ne fait que construire une longue liste peut allouer environ 8500 paires au maximum (en supposant qu'il n'y a pas la limite des 8192 paires imposée par le taggage, voir figure 3.1).

Chapitre 7

L'état actuel des travaux

Nous commençons par décrire concrètement l'état de l'interprète et du compilateur. Ensuite, nous mentionnons plusieurs améliorations qui pourraient être apportées à l'ensemble du système.

7.1 Détails d'implantation

7.1.1 Le compilateur

Le compilateur est implanté en Scheme. Il n'est pas conçu pour être ultra-rapide, bien que, malgré tout, il se trouve à être relativement efficace. Certaines parties devraient être reprogrammées différemment si l'efficacité devenait cruciale.

Afin de donner une idée de sa rapidité, nous décrivons l'environnement sur lequel nous l'avons testé et quel genre de programme il a compilé. Nous avons employé l'interprète Scheme `scm` ([IntScm]) écrit par Aubrey Jaffer pour faire fonctionner notre compilateur. La machine employée est un DEC AXP 3000, doté d'un microprocesseur DEC Alpha tournant à 150 MHz sous OSF/1 version 3.1 et possédant 160 Moctets de mémoire vive. Le programme test à compiler est le fichier de test d'implantation écrit par Aubrey Jaffer ([TestImp]). Il s'agit d'un source d'environ 27 kilo-octets. Environ une centaine parmi le millier de lignes ont été mises en commentaire parce que nous ne voulons pas tester les longs nombres, les nombres en point flottant et les entrées/sorties. Ce programme, en testant presque toutes les fonctions force l'inclusion de la quasi-totalité de la librairie. La compilation de ce programme prend environ 8 secondes. Notre compilateur a donc une efficacité bien tolérable en pratique.

7.1.2 L'interprète

De ce côté, le travail n'est pas aussi complet. Nous n'avons pas véritablement introduit l'interprète sur le micro-contrôleur. Toutefois, nous nous sommes assurés que cette tâche est possible.

Le noyau est implanté en C et nous avons faits des tests sur la taille de l'interprète à l'aide d'un compilateur C qui compile vers des instructions de notre micro-contrôleur. Il s'agit du compilateur `gcc` de GNU muni d'un générateur de code pour le micro-contrôleur. Cette adaptation vient de Otto Lind ([CompGcc]).

Ce compilateur ne génère pas du code binaire aussi compact qu'un programmeur humain pourrait le faire. Un des gros problèmes de ce compilateur vient du fait que le module de génération du code "ment" au compilateur en déclarant qu'il y a plusieurs registres. Or, ces registres sont simulés par le générateur de code. Toute instruction de manipulation de registres que le compilateur veut produire est traduite en deux ou trois instructions réelles. De plus, la gestion de la pile C requiert plusieurs instructions par fonction. Or, notre interprète n'utilise pas la récursivité. Une implantation de l'interprète directement en assembleur serait beaucoup plus compacte.

Malgré tout, la compilation de l'interprète pour le micro-contrôleur produit un code binaire d'environ 22 Koctets, librairie incluse. Ce qui nous permet de conclure que l'interprète pourrait être introduit dans le micro-contrôleur. Le programme que nous utilisons pour ce test est le premier programme décrit à la section 6.4.

Un des manques de l'interprète concerne les constantes littérales. La technique de transmission des constantes allouées prévoit de se débarrasser de la description des constantes une fois que celles-ci sont reconstruites (section 5.5.3). Notre implantation ne peut malheureusement pas se débarrasser de la description.

7.2 Mesures de taille et de performance

7.2.1 Comparaison de la taille des exécutables

Dans l'introduction, nous mentionnons que nous n'avons pas trouvé d'interprète Scheme suffisamment petit pour être utilisé directement sur un micro-contrôleur. C'est en compilant plusieurs interprètes Scheme sur une station de travail DEC (décrite à la section 7.1.1) que nous avons pu le vérifier. Tous ces interprètes proviennent du *Scheme Repository*[1], un site où la plupart des programmes et informations les plus intéressants à propos de Scheme sont accumulées.

La figure 7.1 présente la taille de l'exécutable dérivant de plusieurs implantations. Les im-

1. `ftp://ftp.cs.indiana.edu/pub/scheme-repository/`

Implantation	Taille de l'exéc.
`fools 1.3.2`	288 Koctets
`minischeme 0.85`	95 Koctets
`scm 4e1`	368 Koctets
`siod 3.0`	166 Koctets
μ`scheme`	72 Koctets

FIG. 7.1 - *Taille de l'exécutable de plusieurs interprètes*

plantations ne sont pas toutes mentionnées. Ca peut être le cas si elles sont absentes du *Scheme Repository* ou trop volumineuses pour être possiblement adaptables à un espace très réduit. Notre interprète est mentionné sous le nom μ`scheme` (pour micro-contrôleur) dans la figure. Il est compilé avec la librairie complète. Evidemment, il faut s'attendre à ce que la taille de l'exécutable pour DEC soit différente de celle de l'exécutable pour le micro-contrôleur. Sur DEC, les instructions sont plus volumineuses et la librairie C standard occupe un espace considérable.

Clairement, la seule implantation qui ait une taille qui se rapproche de celle de notre interprète est `minischeme`. Toutefois, celle-ci est passablement incomplète. Il manque de nombreuses fonctions à la librairie. Tout particulièrement en ce qui concerne le traitement des chaînes de caractères. Certaines fonctions sont implantées mais ne permettent pas de traiter un nombre variable d'arguments comme l'exige le standard.

Nous tenons à préciser que ces tests montrent seulement que les autres interprètes, tels que distribués, ne peuvent être utilisés dans le cadre d'un micro-contrôleur. Nous n'excluons pas que certains d'entre eux puissent être modifiés en des versions non-interactives suffisamment petites. C'est d'ailleurs le cas avec notre interprète: seules les versions compilées sont adéquates; les versions interactives sont trop volumineuses.

7.2.2 Tests de performance

Même si nous n'avons pas introduit l'interprète dans le micro-contrôleur, il importe d'en vérifier les performances. Nous avons donc effectué des tests sur une station de travail DEC (voir section 7.1.1). Les tests consistent à comparer, sur quelques programmes, les temps d'exécution de notre interprète avec ceux de deux interprètes connus.

Les deux interprètes de référence sont l'interprète `scm` et l'interprète Gambit (`gsi`) de Marc Feeley ([IntGsi]). La figure 7.2 montre les temps d'exécution de notre interprète (nommé μ`scheme` dans la figure) par rapport à ceux des interprètes de référence.

On peut constater que notre interprète est passablement lent. Le cas du programme `browse` est nettement plus pathétique que les autres. La raison de cette différence marquée est l'utilisation fréquente de la fonction `string->symbol`. Contrairement aux implantations qui vise l'efficacité, notre interprète ne stocke pas l'ensemble des symboles dans une table de hashage. La recherche d'un nom de symbole implique une recherche dans toute la liste des noms. La

	$T_{\mu\texttt{scheme}}$	$T_{\texttt{scm}}$	$T_{\mu\texttt{scheme}}/T_{\texttt{scm}}$	$T_{\texttt{gsi}}$	$T_{\mu\texttt{scheme}}/T_{\texttt{gsi}}$
`browse`	512.81 s	6.69 s	76.7	9.87 s	52.0
`cpstak`	54.38 s	6.02 s	9.0	11.68 s	4.7
`destruc`	150.96 s	5.72 s	26.4	12.78 s	11.8
`fib`	59.31 s	4.79 s	12.4	10.24 s	5.8
`nqueens`	101.06 s	7.97 s	12.7	16.11 s	6.3
`primes`	159.51 s	5.02 s	31.8	8.75 s	18.2

FIG. 7.2 - *Tests de performance comparatifs*

création d'un nouveau symbole demande donc un temps proportionnel au nombre de symbole pré-existants. Donc, n appels à la fonction `string->symbol` demandent un temps dans $O(n^2)$, le pire cas étant atteint quand de nouveaux symboles sont créés à chaque appel.

Les principales causes de la lenteur de notre interprète sont le GC temps réel, l'implantation des fonctions de la librairie en réutilisant le code au maximum et une tendance générale à favoriser la compacité à la performance dans le noyau et le code-octets.

7.3 Améliorations possibles

Le défi de réaliser une implantation la plus compacte possible peut être aussi dur à relever que le défi de rendre une implantation la plus efficace possible. Comme il n'y a pas beaucoup de travaux effectués du côté de la compacité, il faut explorer toutes les directions à la fois. Nous n'avons donc pas pu approfondir chaque aspect, étant donné qu'ils sont nombreux.

La liste de choses qui pourrait être améliorées pourrait devenir très longue si l'on voulait tout mentionner. Nous nous concentrons sur les points principaux.

- Les nombres en point flottant et les grands entiers pourraient être implantés. Comme ces ajouts peuvent augmenter sensiblement la taille de l'interprète, l'utilisateur du système devrait pouvoir contrôler l'inclusion de ces services.
- Lorsque les noms des symboles présents dans un programme ne sont pas requis, l'interprète ne devrait pas les conserver. Les symboles dont nous parlons ici, ce sont les symboles sous formes de données et non pas ceux servant d'identificateurs. Le compilateur pourrait détecter si les fonctions qui créent des symboles ou qui retournent leur épellation sont utilisées. Lorsqu'elles ne le sont pas, il devient inutile de conserver l'épellation du nom de chaque symbole. Un numéro taggé serait suffisant pour les distinguer les uns des autres.
- On a vu que les blocs chaînés sont plus compacts lorsque des variables sont partagées par plusieurs fermetures. La représentation plate est plus compacte lorsque peu de variables doivent être conservées parmi toutes celles de l'environnement courant. Il existe probablement une façon de combiner le partage des blocs de variables communs et la sélection des

variables.

- Le compilateur pourrait employer certaines analyses déjà existantes qui servent à éviter d'allouer des fermetures lorsque ce n'est pas nécessaire. Ces analyses sont conçues à l'origine pour l'efficacité, mais elles peuvent éventuellement servir à limiter la consommation de mémoire. Par exemple, des techniques possiblement utiles sont le *lambda-lifting* ([PJ87]) et l'analyse *0cfa* ([Shi91]).

- La substitution des fonctions par des fonctions plus simples peut causer l'existence de code mort. Par exemple, si à tous les endroits où la fonction `+` est utilisée, elle est remplacée par la fonction `math+2`, alors le code-octets décrivant la fonction `+` ne peut plus être atteint. Une passe dans le code-octets pourrait permettre d'éliminer le code inutile.

- La fonction de reconstruction des constantes allouées est écrite en Scheme et cause l'inclusion de quelques autres fonctions Scheme. Or, selon la nature des constantes, ces autres fonctions ne sont pas toutes utilisées. Il serait intéressant d'adapter la fonction de reconstruction pour chaque programme afin de limiter l'inclusion des fonctions à celles qui sont effectivement utilisées.

- Le noyau utilisé pourrait être adapté aux besoins d'un programme grâce à des options spécifiées par le programmeur lors de la compilation. Entre autre, le choix d'utiliser un GC temps réel ou non devrait être laissé au programmeur. Un GC bloquant est bien plus efficace. Aussi, le traitement des continuations comme objets de première classe a une influence certaine sur l'interprète. Si le programmeur certifie que son programme n'utilise jamais les continuations pour effectuer autre choses que des fonctions de sortie non-locale à usage unique, alors le noyau peut être implanté plus efficacement. On pourrait permettre de choisir une représentation des données parmi plusieurs. La liste des choses paramétrables pourrait s'allonger encore.

Chapitre 8

Conclusion

Notre travail visait à étudier les différents aspects reliés à un système Scheme compact. Le but principal consistait à montrer qu'on peut faire une implantation suffisamment compacte pour pouvoir fonctionner sur un micro-contrôleur. Les buts secondaires étaient de trouver ou développer un GC temps réel implantable de façon compacte et de choisir des techniques portables.

Les différents aspects que nous avons étudiés sont: la représentation des données, principalement en ce qui concerne le typage, les symboles, les continuations et les environnements lexicaux; le domaine des GC temps réel; le développement d'un modèle d'exécution très compact utilisant la compilation.

Nous avons obtenu des résultats positifs à deux points de vue. Premièrement, lors de la recherche d'un GC temps réel adéquat, nous avons développé une nouvelle technique. Il s'agit d'un GC temps réel compactant à un espace. Cette technique a été publiée dans une conférence au début de l'année ([DFS96]).

Deuxièmement, malgré que notre implantation du système n'ait pas tourné sur un micro-contrôleur comme tel, nous avons pu montrer qu'une implantation d'un système de programmation Scheme pour micro-contrôleur est tout à fait possible. En effet, l'exécutable produit par la compilation de l'interprète complet a une taille de seulement 22 Koctets.

Nous estimons que nos objectifs sont atteints.

Remerciements

Je tiens surtout à remercier mon directeur de recherches, Marc Feeley, pour toute l'aide apportée, tant durant le travail de recherche que durant la rédaction de ce mémoire. L'intérêt qu'il portait à mon travail était toujours une source de motivation. Et enfin, il faut l'admettre, sa patience est une qualité qui a été sollicitée souvent dans nos relations.

Je tiens aussi à remercier mes camarades d'études, pour leur support moral et pour leur aide en général. Leurs encouragements et leurs suggestions ont été fort appréciées.

Enfin, je tiens à remercier les membres de ma famille pour leur support moral. Ils m'ont toujours soutenu, surtout dans les moments les plus durs.

Remerciements au CRSNG qui a supporté financièrement mes études de maîtrise.

Bibliographie

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21 (4): 280-294, avril 1978.

[Bak92] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27 (3): 66-70, mars 1992.

[Bar88] Joel Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, Février 1988.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, 108-113, août 1984. Austin, Texas.

[BW88] Hans-Juergen Boehm et Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18 (9) : 807-820, Septembre 1988.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13 (11): 677-678, novembre 1970.

[CN83] Jacques Cohen et Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5 (4): 532-553, octobre 1983.

[CompGcc] Auteur: Otto Lind. Source:
`ftp://wattson.ee.ualberta.ca/pub/motorola/68hc11/gcc/gcc-6811-fsf.tar.gz`

[Del90] Delacour, Vincent. Un gestionnaire mémoire indépendant pour K2. *Rapport de recherche de l'ICSLA*, LIX, Ecole polytechnique, Laboratoire d'informatique, 23-32, octobre 1990.

[DFS96] Danny Dubé, Marc Feeley et Manuel Serrano. Un GC temps réel semi-compactant. *Actes des Journées Francophones des Langages Applicatifs 1996*, Val-Morin, Québec, Canada, janvier 1996.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, et E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21 (11) : 966-975, Novembre 1978.

[FL87] Marc Feeley, Guy Lapalme. Using Closures for Code Generation. *Journal of Computer Languages*, Vol. 12, No. 1, pp. 47-66, Pergamon Press, 1987.

[FL92] Marc Feeley, Guy Lapalme. Closure Generation Based on Viewing Lambda as Epsilon plus Compile, *Journal of Computer Languages*, Vol. 17, No. 4, pp. 251-267, Pergamon Press, 1992.

[Gud93] David Gudeman, Representing Type Information in Dynamically Typed Language. Department of Computer Science, The University of Arizona, TR 93-27, October 1993.

[IntGsi] Auteur: Marc Feeley. Source:
`ftp://ftp.iro.umontreal.ca/pub/parallele/gambit/gambc.tgz`

[IntMini] Auteur: Atsushi Moriwaki. Source:
`ftp://ftp.cs.indiana.edu/pub/scheme-repository/imp/minischeme.tar.gz`

[IntScm] Auteur: Aubrey Jaffer. Source:
`ftp://ftp.cs.indiana.edu/pub/scheme-repository/imp/scm4e1.tar.gz`

[IntSiod] Auteur: George Carrette. Source:
`ftp://ftp.cs.indiana.edu/pub/scheme-repository/imp/siod-3.0.tar.gz`

[Mor78] Morris, F. L. A time- and space-efficient garbage compaction algorithm. *Comm. ACM* 8 (1978), 662-665.

[Nil88] Nilsen, Kelvin. Garbage collection of strings and linked data structures in real time. *Software, Practice and Experience*, 18 (7): 613-640, juillet 1988.

[NS92] K. D. Nilsen et W. J. Schmidt, A High-Performance Hardware-Assisted Real-Time Garbage Collection System. En soumission.

[PJ87] Peyton Jones, S. L. *The Implementation of Functional Programming Languages.* Englewood Cliffs, N.J.: Prentice-Hall; 1987.

[R^{4}RS] *Revised^4 Report on the Algorithmic Language Scheme*, William Clinger et Jonathan Rees (éditeurs), 1991.

[Shi91] O. Shivers. The Semantics of Scheme Control-Flow Analysis. *Proceedings of PEPM '91, ACM Sigplan Notices*, 26(9):190-198, septembre 1991.

[Ste75] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18 (9): 495-508, septembre 1975.

[TestImp] Auteur: Aubrey Jaffer. Source:
`ftp://ftp.cs.indiana.edu/pub/scheme-repository/code/lang/test.scm`

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. Dans Yves Bekkers et Jacques Cohen, éditeurs, *International Workshop on Memory Management*, numéro 637 dans Lecture Notes in Computer Science, pages 1-42, St Malo, France, Septembre 1992. Springer-Verlag.

[Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11: 181-198, 1990.

Annexe A

Sources du système

Les différents fichiers composant le système de programmation Scheme pour micro-contrôleur sont recopiés ici. Il s'agit du système final. Il comporte le compilateur vers du code-octets et le noyau est doté du GC temps réel et de l'interprète pour le second jeu d'instructions de la machine virtuelle. Tout d'abord, les fichiers sources du noyau sont présentés. Ensuite, vient le fichier contenant la librairie, suivi du compilateur vers du code-octets. Enfin, à titre d'illustration, le résultat de la compilation d'un petit programme est présenté.

A.1 Sources du noyau

Le noyau est constitué de deux fichiers: `noyau.h` et `noyau.c`. Tel qu'illustré à la figure 5.1, un exécutable est formé en compilant le noyau avec un fichier produit par le compilateur de code-octets.

A.1.1 `noyau.h`

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3. #include <string.h>
 4.
 5. /* Personnalisation */
 6. #define TRUE 1
 7. #define FALSE 0
 8.
 9. /* Configuration */
10. #define CHARSPERINT (sizeof(int))
11. #define LOGCPI 2
12. #define DEFAULTHEAPSIZE 65536
13.
14. /* Variables globales C */
15. extern int *mem;
16. extern int  mem_len, nb_handles, nb_obj, handle1;
17. extern int  mem_next, mem_free, mem_stack;
18. extern int  gc_mark, gc_compact, gc_ratio, gc_time;
19. extern int  gc_trav, gc_vecting, gc_cut;
20. extern int  gc_old, gc_new, gc_len, gc_src, gc_dst, gc_state;
21. extern int  nb_symbols;
22. extern int  eval_pc;
23. extern int (*cprim0[])(void);
24. extern int (*cprim1[])(int);
25. extern int (*cprim2[])(int, int);
26. extern int (*cprim3[])(int, int, int);
27.
28. /* Variables globales SCM */
29. #define NBTEMPS 6
30. #define cons_car            (globs[0])
31. #define cons_cdr            (globs[1])
32. #define string_copy_s1      (globs[0])
33. #define make_clos_env       (globs[0])
```

```
34. #define list_copy_tete        (globs[2])
35. #define list_copy_courant     (globs[3])
36. #define list_copy_l           (globs[4])
37. #define make_rest_frame_frame (globs[5])
38.
39. #define NBSINGLES 10
40. #define scm_empty            (globs[NBTEMPS + 0])
41. #define scm_false            (globs[NBTEMPS + 1])
42. #define scm_true             (globs[NBTEMPS + 2])
43. #define scm_symbol_names     (globs[NBTEMPS + 3])
44. #define scm_constants        (globs[NBTEMPS + 4])
45.
46. #define eval_val             (globs[NBTEMPS + 5])
47. #define eval_env             (globs[NBTEMPS + 6])
48. #define eval_args            (globs[NBTEMPS + 7])
49. #define eval_prev_args       (globs[NBTEMPS + 8])
50. #define eval_cont            (globs[NBTEMPS + 9])
51.
52. #define NBGLOBS (NBTEMPS + NBSINGLES)
53. extern int globs[];
54.
55. /* Description des types et macros du GC */
56. #define BITH1 0x80000000
57. #define BITH2 0x40000000
58. #define BITB4 0x8
59. #define BITB3 0x4
60. #define BITB2 0x2
61. #define BITB1 0x1
62.
63. #define NUMMASK     BITB1
64. #define NUMVAL      BITB1
65. #define TYPEMASK    (BITH1 | BITH2 | BITB1)
66. #define TYPEPAIR    0
67. #define TYPECLOS    BITH2
68. #define TYPEOTHER   BITH1
69. #define TYPESPEC    (BITH1 | BITH2)
70. #define CONTMASK    BITB1
71. #define CONTVAL     BITB1
72. #define VECTINGMASK (BITB2 | BITB1)
73. #define VECTORVAL   0
74. #define STRINGVAL   BITB2
75. #define SYMBMASK    (BITH1 | BITH2 | BITB2 | BITB1)
76. #define SYMBVAL     (BITH1 | BITH2 | BITB2)
77. #define SPECMASK    (BITH1 | BITH2 | BITB4 | BITB3 | BITB2 | BITB1)
78. #define SPECCHAR    (BITH1 | BITH2)
79. #define SPECCPRIM   (BITH1 | BITH2 | BITB3)
80. #define SPECBOOL    (BITH1 | BITH2 | BITB4)
81. #define NULLVAL     0xfffffffc
82. #define FALSEVAL    0xfffffff8
83. #define TRUEVAL     0xfffffff8
84. #define HANDLEMASK  (~TYPEMASK)
85.
86. #define C_PRED_NUMBER(d)  (((d) & NUMMASK) == NUMVAL)
87. #define C_PRED_PAIR(d)    (((d) & TYPEMASK) == TYPEPAIR)
88. #define C_PRED_CLOSURE(d) (((d) & TYPEMASK) == TYPECLOS)
89. #define C_PRED_OTHER(d)   (((d) & TYPEMASK) == TYPEOTHER)
90. #define C_PRED_SPECIAL(d) (((d) & TYPEMASK) == TYPESPEC)
91. #define C_PRED_SYMBOL(d)  (((d) & SYMBMASK) == SYMBVAL)
92. #define C_PRED_CHAR(d)    (((d) & SPECMASK) == SPECCHAR)
93. #define C_PRED_CPRIM(d)   (((d) & SPECMASK) == SPECCPRIM)
94. #define C_PRED_BOOLEAN(d) (((d) & SPECMASK) == SPECBOOL)
95.
96. #define GC_MARK         1
97. #define IS_MARKED(data) (mem[mem[SCM_TO_HANDLE(data)]] & GC_MARK)
98.
99. /* Fonctions de manipulation des donnees */
100.
101. #define SCM_TO_HANDLE(d)     (((d) & HANDLEMASK) >> 1)
102. #define HANDLE_TO_PAIR(i)    (((i) << 1) | TYPEPAIR)
103. #define HANDLE_TO_CLOSURE(i) (((i) << 1) | TYPECLOS)
104. #define HANDLE_TO_OTHER(i)   (((i) << 1) | TYPEOTHER)
105. #define TO_SCM_CHAR(c)       (((c) << 4) | SPECCHAR)
106. #define TO_C_CHAR(c)         (((c) >> 4) & 0xff)
107. #define C_STRING_LEN(s)      (mem[mem[SCM_TO_HANDLE(s)] + 1] >> 2)
108. #define SYMBOL_NUMBER(s)     (((s) & ~SYMBMASK) >> 2)
109. #define NUMBER_TO_SYMBOL(n)  (((n) << 2) | SYMBVAL)
110. #define TO_SCM_NUMBER(n)     (((n) << 1) | NUMVAL)
111. #define C_VECTOR_LEN(v)      (mem[mem[SCM_TO_HANDLE(v)] + 1] >> 2)
112. #define C_MAKE_CPRIM(n)      (((n) << 4) | SPECCPRIM)
113. #define CPRIM_NUMBER(c)      (((c) & ~SPECMASK) >> 4)
114. #define PC_TO_PCTAG(p)       (((p) << 1) | CONTVAL)
115. #define PCTAG_TO_PC(p)       ((p) >> 1)
116. #define CHAMP(d,champ)       (mem[mem[SCM_TO_HANDLE(d)] + 1 + (champ)])
117.
118. /* Description des primitives C */
119. #define NBCPRIM0 4
120. #define FIRSTCPRIM0 0
121. #define NBCPRIM1 21
122. #define FIRSTCPRIM1 (FIRSTCPRIM0 + NBCPRIM0)
123. #define NBCPRIM2 14
124. #define FIRSTCPRIM2 (FIRSTCPRIM1 + NBCPRIM1)
125. #define NBCPRIM3 2
126. #define FIRSTCPRIM3 (FIRSTCPRIM2 + NBCPRIM2)
127. #define APPLY_CPRIM_NO 36
128.
129. /* Variables du module Scheme compile */
130. extern int bytecode_len;
131. extern unsigned char bytecode[];
132. extern int const_desc_len;
133. extern unsigned char const_desc[];
134. extern int nb_scm_globs;
135. extern int scm_globs[];
136.
137. /* Prototypes de bit.c */
138. void  alloc_heap(int taille);
139. int   is_allocated(int d);
140. void  mark_it(int d);
141. void  gc(int size);
142. int   alloc_cell(int len, int bitpattern);
143. int   pred_boolean(int d);
144. int   pred_pair(int d);
145. int   cons(int car, int cdr);
146. int   car(int p);
147. int   cdr(int p);
148. int   set_car(int p, int d);
149. int   set_cdr(int p, int d);
150. int   list_copy(int l);
151. int   pred_char(int d);
152. int   integer_to_char(int n);
153. int   char_to_integer(int c);
154. int   c_pred_string(int d);
155. int   pred_string(int d);
156. int   c_make_string(int len);
157. int   make_string(int len);
158. char *find_string_base(int s, int pos);
159. int   c_string_ref(int s, int pos);
160. int   string_ref(int s, int pos);
161. void  c_string_set(int s, int pos, int c);
```

```
162. int   string_set(int s, int pos, int c);
163. int   string_length(int s);
164. int   string_len_to_int_string_len(int len);
165. int   c_string_int_len(int s);
166. int   to_scm_string(char *cs, int len);
167. int   c_string_equal(int s1, int s2);
168. int   string_equal(int s1, int s2);
169. int   string_copy(int s1);
170. int   pred_symbol(int d);
171. void  stretch_symbol_vector(void);
172. int   make_symbol(int nom);
173. int   symbol_to_string(int s);
174. int   string_to_symbol(int string);
175. int   pred_number(int d);
176. int   to_c_number(int n);
177. int   cppoe2(int n1, int n2);
178. int   cplus2(int n1, int n2);
179. int   cmoins2(int n1, int n2);
180. int   cfois2(int n1, int n2);
181. int   cdivise2(int n1, int n2);
182. int   c_pred_vector(int d);
183. int   pred_vector(int d);
184. int   c_make_vector(int len);
185. int   make_vector(int len);
186. int   find_vector_location(int v, int pos);
187. int   c_vector_ref(int v, int pos);
188. int   vector_ref(int v, int pos);
189. void  c_vector_set(int v, int pos, int val);
190. int   vector_set(int v, int pos, int val);
191. int   vector_length(int v);
192. int   vector_len_to_int_vector_len(int len);
193. int   c_vector_int_len(int v);
194. int   pred_procedure(int d);
195. int   make_closure(int entry, int env);
196. int   closure_entry(int c);
197. int   closure_env(int c);
198. int   apply(int c, int args);
199. int   c_pred_cont(int d);
200. int   make_cont(void);
201. void  set_cont_pc(int k, int dest);
202. void  restore_cont(int k);
203. int   ll_input(void);
204. int   c_peek_char(void);
205. int   peek_char(void);
206. int   c_read_char(void);
207. int   read_char(void);
208. int   write_char(int c);
209. int   eq(int d1, int d2);
210. int   quit(void);
211. int   return_current_continuation(void);
212. int   return_there_with_this(int complete_cont, int val);
213. int   introspection(int arg);
214. void  init_bc_reader(void);
215. int   read_bc_byte(void);
216. int   read_bc_int(void);
217. int   get_frame(int step);
218. int   get_var(int frame, int offset);
219. void  set_var(int frame, int offset, int val);
220. void  make_normal_frame(int size);
221. void  make_rest_frame(int size);
222. void  push_arg(int arg);
223. int   pop_arg(void);
224. void  pop_n_args(int n);
225. int   apply_cprim(int cprim_no, int args);
226. int   apply_closure(int c, int args);
227. void  c_apply(int c, int args);
228. void  execute_bc(void);
229. int   init_scm_glob_1(int code);
230. int   init_scm_glob_2(int code);
231. int   main(int argc, char *argv[]);
```

A.1.2 noyau.c

```
 1. /* Mini interprete Scheme. Troisieme d'une serie. */
 2. /* Ne traite pas: */
 3. /*     les ports, */
 4. /*     les points flottants, */
 5. /*     les entiers de taille illimitee, */
 6. /*     l'entree de nombres non-decimaux. */
 7.
 8. #include "noyau.h"
 9.
10.
11.
12.
13. /* Variables globales C ---------------------------------------- */
14.
15. int *mem;
16. int  mem_len, nb_handles, nb_obj, handle1, mem_next, mem_free, mem_stack;
17. int  gc_mark, gc_compact, gc_ratio, gc_time, gc_trav, gc_vecting, gc_cut;
18. int  gc_old, gc_new, gc_len, gc_src, gc_dst, gc_state;
19.
20. int nb_symbols;
21.
22. int eval_pc;
23.
24. int (*cprim0[NBCPRIM0])(void) =
25. {
26.   peek_char,
27.   read_char,
28.   quit,
29.   return_current_continuation
30. };
31.
32. int (*cprim1[NBCPRIM1])(int) =
33. {
34.   pred_boolean,
35.   pred_pair,
36.   car,
37.   cdr,
38.   pred_char,
39.   integer_to_char,
40.   char_to_integer,
41.   pred_string,
42.   make_string,
43.   string_length,
44.   string_copy,
45.   pred_symbol,
46.   symbol_to_string,
47.   string_to_symbol,
48.   pred_number,
49.   pred_vector,
50.   make_vector,
51.   vector_length,
```

```
52.   pred_procedure,
53.   write_char,
54.   introspection
55. };
56.
57. int (*cprim2[NBCPRIM2])(int, int) =
58. {
59.   cons,
60.   set_car,
61.   set_cdr,
62.   string_ref,
63.   string_equal,
64.   cppoe2,
65.   cplus2,
66.   cmoins2,
67.   cfois2,
68.   cdivise2,
69.   vector_ref,
70.   apply,
71.   eq,
72.   return_there_with_this
73. };
74.
75. int (*cprim3[NBCPRIM3])(int, int, int) =
76. {
77.   string_set,
78.   vector_set
79. };
80.
81.
82.
83.
84. /* Variables globales SCM ------------------------------------- */
85.
86. /* Les variables contenues dans stack */
87. /* doivent etre mises a jour par le gc */
88. int globs[NBGLOBS];
89.
90.
91.
92.
93. /* Allocation du monceau ------------------------------------- */
94.
95. void alloc_heap(int taille)
96. {
97.   int j;
98.
99.   mem_len = taille;
100.   mem = malloc(mem_len * (sizeof(int)));
101.   if (mem == NULL)
102.     {
103.       fprintf(stderr, "Incapable d'allouer le monceau\n");
104.       exit(1);
105.     }
106.   nb_handles = (mem_len + 4) / 5;
107.   nb_obj = 0;
108.   handle1 = 0;
109.   for (j=0 ; j<nb_handles ; j++)
110.     mem[j] = j + 1;
111.   mem[nb_handles - 1] = -1;
112.   mem_next = nb_handles;
113.   mem_free = mem_len - mem_next;
114.   gc_mark = FALSE;
115.   gc_compact = FALSE;
116.   gc_ratio = 2;
117.   gc_time = 0;
118.   gc_vecting = FALSEVAL;
119.   gc_state = 0;
120. }
121.
122.
123.
124.
125. /* Garbage collector ----------------------------------------- */
126.
127. int is_allocated(int d)
128. {
129.   return !C_PRED_NUMBER(d) && !C_PRED_SPECIAL(d);
130. }
131.
132. void mark_it(int d)
133. {
134.   if (!is_allocated(d))
135.     return;
136.   if (IS_MARKED(d))
137.     return;
138.   mem[mem[SCM_TO_HANDLE(d)]] |= GC_MARK;
139.   mem_stack --;
140.   mem[mem_stack] = d;
141. }
142.
143. /* La description des etats est ailleurs */
144. /* Note: mem_free n'est maintenu qu'entre deux phases de GC */
145. /* Note: meme chose pour nb_obj */
146. void gc(int size)
147. {
148.   int court, start, j, nbcases, marque, d, handle, subnlen;
149.
150.   gc_time += size + size;    /* Simuler une multiplication rapide */
151.   if (gc_ratio >= 3)
152.     {
153.       gc_time += size;
154.       if (gc_ratio >= 4)
155.         {
156.           gc_time += size;
157.           if (gc_ratio >=5)
158.             gc_time += size * (gc_ratio - 4);
159.         }
160.     }
161.
162.   while (gc_time > 0)
163.     switch (gc_state)
164.       {
165.       case 0:
166.         {
167.           if (mem_free == 0)
168.             gc_ratio = mem_len;
169.           else
170.             gc_ratio = (2 * (mem_next - nb_handles) +
171.                         2 * nb_obj +
172.                         2 * nb_scm_globs +
173.                         (5 * mem_free + 1) / 2 -
174.                         1) / mem_free;
175.           gc_time += size * gc_ratio;   /* Car le ratio etait insuffisant */
176.           gc_time -= (gc_ratio *
177.                       (((2 * gc_ratio - 3) * mem_free -
178.                         4 * (mem_next - nb_handles) -
179.                         4 * nb_obj -
```

```
180.                      4 * nb_scm_globs) /
181.                     (2 * gc_ratio + 1)));
182.          gc_time += 50;                /* Securite! */
183.          gc_state = 1;
184.          break;
185.        }
186.      case 1:
187.        {
188.          mem_stack = mem_len;
189.          gc_mark = TRUE;
190.          gc_cut = 0;
191.          gc_state = 2;
192.          break;
193.        }
194.      case 2:
195.        {
196.          nbcases = ((nb_scm_globs - gc_cut <= gc_time) ?
197.                     nb_scm_globs - gc_cut :
198.                     gc_time);
199.          for (j=gc_cut ; j<gc_cut+nbcases ; j++)
200.            mark_it(scm_globs[j]);
201.          gc_time -= nbcases;
202.          gc_cut += nbcases;
203.          gc_state = (gc_cut == nb_scm_globs) ? 5 : 2;
204.          break;
205.        }
206.      case 3:
207.        {
208.          if (mem_stack == mem_len)
209.            gc_state = 5;
210.          else
211.            {
212.              gc_trav = mem[mem_stack];
213.              mem_stack ++;
214.              gc_src = mem[SCM_TO_HANDLE(gc_trav)];
215.              switch (gc_trav & TYPEMASK)
216.                {
217.                case TYPEPAIR:                      /* PAIR */
218.                  {
219.                    start = 1;
220.                    gc_len = 3;
221.                    break;
222.                  }
223.                case TYPECLOS:                      /* CLOSURE */
224.                  {
225.                    start = 2;
226.                    gc_len = 3;
227.                    break;
228.                  }
229.                default:
230.                  {
231.                    subnlen = mem[gc_src + 1];
232.                    if (subnlen & CONTMASK)         /* CONT */
233.                      {
234.                        start = 2;
235.                        gc_len = 5;
236.                      }
237.                    else if (subnlen & VECTINGMASK)  /* STRING */
238.                      {
239.                        start =
240.                          2 + string_len_to_int_string_len(subnlen >> 2);
241.                        gc_len = start;
242.                      }
243.                    else                            /* VECTOR */
244.                      {
245.                        gc_len =
246.                          2 + vector_len_to_int_vector_len(subnlen >> 2);
247.                        gc_cut = 2;
248.                        gc_time -= 3;
249.                        gc_state = 4;
250.                        goto label001;
251.                      }
252.                  }
253.                }
254.              for (j=start ; j<gc_len ; j++)
255.                mark_it(mem[gc_src + j]);
256.              gc_time -= gc_len + 1;
257.              gc_state = 3;
258.            }
259.        label001:
260.          break;
261.        }
262.      case 4:
263.        {
264.          nbcases = (gc_len - gc_cut <= gc_time) ? gc_len - gc_cut : gc_time;
265.          for (j=0 ; j<nbcases ; j++)
266.            mark_it(mem[gc_src + gc_cut + j]);
267.          gc_time -= nbcases;
268.          gc_cut += nbcases;
269.          gc_state = (gc_cut == gc_len) ? 3 : 4;
270.          break;
271.        }
272.      case 5:
273.        {
274.          for (j=0 ; j<NBGLOBS ; j++)
275.            mark_it(globs[j]);
276.          gc_state = (mem_stack == mem_len) ? 6 : 3;
277.          break;
278.        }
279.      case 6:
280.        {
281.          gc_mark = FALSE;
282.          gc_compact = TRUE;
283.          gc_src = nb_handles;
284.          gc_dst = nb_handles;
285.          gc_state = 7;
286.          nb_obj = 0;
287.          break;
288.        }
289.      case 7:
290.        {
291.          gc_state = (gc_src == mem_next) ? 10 : 8;
292.          break;
293.        }
294.      case 8:
295.        {
296.          d = mem[gc_src];
297.          marque = d & GC_MARK;
298.          d &= ~GC_MARK;
299.          mem[gc_src] = d;
300.          switch(d & TYPEMASK)
301.            {
302.            case TYPEPAIR:                      /* PAIR */
303.            case TYPECLOS:                      /* CLOSURE */
304.              {
305.                court = TRUE;
306.                gc_len = 3;
307.                break;
```

```
308.             }
309.           default:
310.             {
311.               subnlen = mem[gc_src + 1];
312.               if (subnlen & CONTMASK)         /* CONT */
313.                 {
314.                   court = TRUE;
315.                   gc_len = 5;
316.                   break;
317.                 }
318.               else if (subnlen & VECTINGMASK)  /* STRING */
319.                 {
320.                   court = FALSE;
321.                   gc_len =
322.                     2 + string_len_to_int_string_len(subnlen >> 2);
323.                   break;
324.                 }
325.               else                            /* VECTOR */
326.                 {
327.                   court = FALSE;
328.                   gc_len =
329.                     2 + vector_len_to_int_vector_len(subnlen >> 2);
330.                   break;
331.                 }
332.             }
333.           }
334.         handle = SCM_TO_HANDLE(d);
335.         if (!marque)
336.           {
337.             mem[handle] = handle1;
338.             handle1 = handle;
339.             gc_time -= gc_len + 1;
340.             gc_src += gc_len;
341.             gc_state = 7;
342.           }
343.         else if (court)
344.           {
345.             mem[handle] = gc_dst;
346.             for (j=0 ; j<gc_len ; j++)
347.               mem[gc_dst + j] = mem[gc_src + j];
348.             nb_obj ++;
349.             gc_time -= gc_len + 1;
350.             gc_src += gc_len;
351.             gc_dst += gc_len;
352.             gc_state = 7;
353.           }
354.         else
355.           {
356.             mem[handle] = gc_dst;
357.             mem[gc_dst] = mem[gc_src];
358.             mem[gc_dst + 1] = mem[gc_src + 1];
359.             nb_obj ++;
360.             gc_time -= 3;
361.             gc_vecting = d;
362.             gc_cut = 2;
363.             gc_old = gc_src;
364.             gc_new = gc_dst;
365.             gc_dst += gc_len;
366.             gc_state = 9;
367.           }
368.         break;
369.       }
370.     case 9:
371.       {
372.         nbcases = (gc_len - gc_cut <= gc_time) ? gc_len - gc_cut : gc_time;
373.         for (j=0 ; j<nbcases ; j++)
374.           mem[gc_new + gc_cut + j] = mem[gc_old + gc_cut + j];
375.         gc_time -= nbcases;
376.         gc_cut += nbcases;
377.         if (gc_cut == gc_len)
378.           {
379.             gc_vecting = scm_false;
380.             gc_src += gc_len;
381.             gc_state = 7;
382.           }
383.         else
384.           gc_state = 9;
385.         break;
386.       }
387.     case 10:
388.       {
389.         mem_next = gc_dst;
390. /*        printf("%s %d cases pour %d objets, ratio de %d, reste %d\n",
391.                "GC completed:", mem_next - nb_handles,
392.                nb_obj, gc_ratio, mem_free); */
393.         mem_free = mem_len - mem_next - nb_obj;
394.         gc_compact = FALSE;
395.         gc_ratio = 1;
396.         gc_time = 0;
397.         gc_state = 0;
398.         if (mem_free < size)
399.           {
400.             fprintf(stderr, "Manque de memoire\n");
401.             exit(1);
402.           }
403.         break;
404.       }
405.     }
406. }
407.
408.
409.
410.
411. /* Fonction d'allocation de memoire ---------------------------- */
412.
413. /* Un appel a cette fonction peut declencher le gc */
414. int alloc_cell(int len, int bitpattern)
415. {
416.   int j, pos, handle, d, marque;
417.
418.   gc(len + 1);
419.
420.   if (gc_compact)
421.     {
422.       if (gc_dst + len <= gc_src)
423.         {
424.           pos = gc_dst;
425.           gc_dst += len;
426.           marque = 0;
427.         }
428.       else
429.         {
430.           pos = mem_next;
431.           mem_next += len;
432.           marque = GC_MARK;
433.         }
434.     }
435.   else
```

```
436.     {
437.       pos = mem_next;
438.       mem_next += len;
439.       marque = 0;
440.     }
441.
442.   handle = handle1;
443.   handle1 = mem[handle1];
444.   mem[handle] = pos;
445.
446.   d = (handle << 1) | bitpattern;
447.
448.   mem[pos] = d | marque;
449.
450.   return d;
451. }
452.
453.
454.
455.
456. /* Fonctions relatives a BOOLEAN ------------------------------- */
457.
458. int pred_boolean(int d)
459. {
460.   return C_PRED_BOOLEAN(d) ? scm_true : scm_false;
461. }
462.
463.
464.
465.
466. /* Fonctions relatives a PAIR ---------------------------------- */
467.
468. int pred_pair(int d)
469. {
470.   return C_PRED_PAIR(d) ? scm_true : scm_false;
471. }
472.
473. int cons(int car, int cdr)
474. {
475.   int handle, cell, pair;
476.
477.   cons_car = car;
478.   cons_cdr = cdr;
479.   pair = alloc_cell(3, TYPEPAIR);
480.   handle = SCM_TO_HANDLE(pair);
481.   cell = mem[handle];
482.   mem[cell + 1] = cons_car;
483.   mem[cell + 2] = cons_cdr;
484.   return pair;
485. }
486.
487. int car(int p)
488. {
489.   return CHAMP(p, 0);
490. }
491.
492. int cdr(int p)
493. {
494.   return CHAMP(p, 1);
495. }
496.
497. int set_car(int p, int d)
498. {
499.   if (gc_mark && IS_MARKED(p))
500.     mark_it(d);
501.   CHAMP(p, 0) = d;
502.   return scm_true;
503. }
504.
505. int set_cdr(int p, int d)
506. {
507.   if (gc_mark && IS_MARKED(p))
508.     mark_it(d);
509.   CHAMP(p, 1) = d;
510.   return scm_true;
511. }
512.
513. int list_copy(int l)
514. {
515.   int temp;
516.
517.   if (l == scm_empty)
518.     return l;
519.   else
520.     {
521.       list_copy_l = l;
522.       list_copy_courant = list_copy_tete = cons(car(list_copy_l), scm_empty);
523.       list_copy_l = cdr(list_copy_l);
524.       while (list_copy_l != scm_empty)
525.         {
526.           temp = cons(car(list_copy_l), scm_empty);
527.           set_cdr(list_copy_courant, temp);
528.           list_copy_l = cdr(list_copy_l);
529.           list_copy_courant = cdr(list_copy_courant);
530.         }
531.       return list_copy_tete;
532.     }
533. }
534.
535.
536.
537.
538. /* Fonctions relatives a CHAR ---------------------------------- */
539.
540. int pred_char(int d)
541. {
542.   return C_PRED_CHAR(d) ? scm_true : scm_false;
543. }
544.
545. int integer_to_char(int n)
546. {
547.   return TO_SCM_CHAR(to_c_number(n));
548. }
549.
550. int char_to_integer(int c)
551. {
552.   return TO_SCM_NUMBER(TO_C_CHAR(c));
553. }
554.
555.
556.
557.
558. /* Fonctions relatives a STRING -------------------------------- */
559.
560. int c_pred_string(int d)
561. {
562.   return (C_PRED_OTHER(d) &&
563.           ((mem[mem[SCM_TO_HANDLE(d)] + 1] & VECTINGMASK) == STRINGVAL));
```

```
564. }
565.
566. int pred_string(int d)
567. {
568.   return c_pred_string(d) ? scm_true : scm_false;
569. }
570.
571. int c_make_string(int len)
572. {
573.   int intlen, handle, cell, string;
574.
575.   intlen = string_len_to_int_string_len(len);
576.   string = alloc_cell(2 + intlen, TYPEOTHER);
577.   handle = SCM_TO_HANDLE(string);
578.   cell = mem[handle];
579.   mem[cell + 1] = (len << 2) | STRINGVAL;
580.   return string;
581. }
582.
583. int make_string(int len)
584. {
585.   return c_make_string(to_c_number(len));
586. }
587.
588. char *find_string_base(int s, int pos)
589. {
590.   int intpos;
591.
592.   if (s != gc_vecting)
593.     return (char *) (&mem[mem[SCM_TO_HANDLE(s)] + 2]);
594.   else
595.     {
596.       intpos = (pos >> LOGCPI) + 2;
597.       if (intpos >= gc_cut)
598.         return (char *) (&mem[gc_old + 2]);
599.       else
600.         return (char *) (&mem[gc_new + 2]);
601.     }
602. }
603.
604. int c_string_ref(int s, int pos)
605. {
606.   char *base;
607.
608.   base = find_string_base(s, pos);
609.   return (int) (*(base + pos));
610. }
611.
612. int string_ref(int s, int pos)
613. {
614.   return TO_SCM_CHAR(c_string_ref(s, to_c_number(pos)));
615. }
616.
617. void c_string_set(int s, int pos, int c)
618. {
619.   char *base;
620.
621.   base = find_string_base(s, pos);
622.   *(base + pos) = (char) c;
623. }
624.
625. int string_set(int s, int pos, int c)
626. {
627.   c_string_set(s, to_c_number(pos), TO_C_CHAR(c));
628.   return scm_true;
629. }
630.
631. int string_length(int s)
632. {
633.   return TO_SCM_NUMBER(C_STRING_LEN(s));
634. }
635.
636. int string_len_to_int_string_len(int len)
637. {
638.   int intlen;
639.
640.   intlen = (len + CHARSPERINT - 1) >> LOGCPI;
641.   return (intlen == 0) ? 1 : intlen;
642. }
643.
644. int c_string_int_len(int s)
645. {
646.   return string_len_to_int_string_len(C_STRING_LEN(s));
647. }
648.
649. int to_scm_string(char *cs, int len)
650. {
651.   int i, scms;
652.
653.   scms = c_make_string(len);
654.   for (i=0 ; i<len ; i++)
655.     c_string_set(scms, i, cs[i]);
656.   return scms;
657. }
658.
659. int c_string_equal(int s1, int s2)
660. {
661.   int   len1, len2, i;
662.
663.   len1 = C_STRING_LEN(s1);
664.   len2 = C_STRING_LEN(s2);
665.
666.   if (len1 != len2)
667.     return FALSE;
668.
669.   for (i=0 ; i<len1 ; i++)
670.     if (c_string_ref(s1, i) != c_string_ref(s2, i))
671.       return FALSE;
672.   return TRUE;
673. }
674.
675. int string_equal(int s1, int s2)
676. {
677.   return c_string_equal(s1, s2) ? scm_true : scm_false;
678. }
679.
680. int string_copy(int s1)
681. {
682.   int   len, i;
683.   int   s2;
684.
685.   len = C_STRING_LEN(s1);
686.   string_copy_s1 = s1;
687.   s2 = c_make_string(len);
688.   for (i=0 ; i<len ; i++)
689.     c_string_set(s2, i, c_string_ref(string_copy_s1, i));
690.   return s2;
691. }
```

```
692.
693.
694.
695.
696. /* Fonctions relatives a SYMBOL ------------------------------- */
697.
698. int pred_symbol(int d)
699. {
700.   return C_PRED_SYMBOL(d) ? scm_true : scm_false;
701. }
702.
703. void stretch_symbol_vector(void)
704. {
705.   int oldlen, newlen, j;
706.   int newnames;
707.
708.   oldlen = C_VECTOR_LEN(scm_symbol_names);
709.   newlen = (4 * oldlen) / 3 + 1;
710.   newnames = c_make_vector(newlen);
711.   for (j=0 ; j<nb_symbols ; j++)
712.     c_vector_set(newnames, j, c_vector_ref(scm_symbol_names, j));
713.   scm_symbol_names = newnames;
714. }
715.
716. /* Cette fonction ne doit etre utilisee */
717. /* que par string->symbol */
718. int make_symbol(int nom)
719. {
720.   int numero;
721.
722.   if (nb_symbols == C_VECTOR_LEN(scm_symbol_names))
723.     stretch_symbol_vector();
724.
725.   numero = nb_symbols;
726.   nb_symbols ++;
727.   c_vector_set(scm_symbol_names, numero, nom);
728.
729.   return NUMBER_TO_SYMBOL(numero);
730. }
731.
732. int symbol_to_string(int s)
733. {
734.   return c_vector_ref(scm_symbol_names, SYMBOL_NUMBER(s));
735. }
736.
737. int string_to_symbol(int string)
738. {
739.   int j;
740.
741.   for (j=0 ; j<nb_symbols ; j++)
742.     if (c_string_equal(string, c_vector_ref(scm_symbol_names, j)))
743.       return NUMBER_TO_SYMBOL(j);
744.
745.   return make_symbol(string_copy(string));
746. }
747.
748.
749.
750.
751. /* Fonctions relatives a NUMBER ------------------------------- */
752.
753. int pred_number(int d)
754. {
755.   return C_PRED_NUMBER(d) ? scm_true : scm_false;
756. }
757.
758. int to_c_number(int n)
759. {
760.   if (n < 0)
761.     return (n >> 1) | BIT31;
762.   else
763.     return n >> 1;
764. }
765.
766. /* Les op. se font AVEC les tags */
767. int cppoe2(int n1, int n2)
768. {
769.   return (n1 <= n2) ? scm_true : scm_false;
770. }
771.
772. int cplus2(int n1, int n2)
773. {
774.   return n1 + n2 - NUMVAL;
775. }
776.
777. int cmoins2(int n1, int n2)
778. {
779.   return n1 - n2 + NUMVAL;
780. }
781.
782. int cfois2(int n1, int n2)
783. {
784.   return TO_SCM_NUMBER(to_c_number(n1) * to_c_number(n2));
785. }
786.
787. int cdivise2(int n1, int n2)
788. {
789.   return TO_SCM_NUMBER(to_c_number(n1) / to_c_number(n2));
790. }
791.
792.
793.
794.
795. /* Fonctions relatives a VECTOR ------------------------------- */
796.
797. int c_pred_vector(int d)
798. {
799.   return (C_PRED_OTHER(d) &&
800.           ((mem[mem[SCM_TO_HANDLE(d)] + 1] & VECTINGMASK) == VECTORVAL));
801. }
802.
803. int pred_vector(int d)
804. {
805.   return c_pred_vector(d) ? scm_true : scm_false;
806. }
807.
808. int c_make_vector(int len)
809. {
810.   int intlen, j;
811.   int cell, handle, vector;
812.
813.   intlen = vector_len_to_int_vector_len(len);
814.   vector = alloc_cell(2 + intlen, TYPEOTHER);
815.   handle = SCM_TO_HANDLE(vector);
816.   cell = mem[handle];
817.   mem[cell + 1] = (len << 2) | VECTORVAL;
818.   for (j=0 ; j<intlen ; j++)
819.     mem[cell + j + 2] = scm_false;
```

```
820.   return vector;
821. }
822.
823. int make_vector(int len)
824. {
825.   return c_make_vector(to_c_number(len));
826. }
827.
828. int find_vector_location(int v, int pos)
829. {
830.   if (v != gc_vecting)
831.     return mem[SCM_TO_HANDLE(v)];
832.   else if (2 + pos >= gc_cut)
833.     return gc_old;
834.   else
835.     return gc_new;
836. }
837.
838. int c_vector_ref(int v, int pos)
839. {
840.   int location;
841.
842.   location = find_vector_location(v, pos);
843.   return mem[location + 2 + pos];
844. }
845.
846. int vector_ref(int v, int pos)
847. {
848.   return c_vector_ref(v, to_c_number(pos));
849. }
850.
851. void c_vector_set(int v, int pos, int val)
852. {
853.   int location;
854.
855.   if (gc_mark && IS_MARKED(v))
856.     mark_it(val);
857.   location = find_vector_location(v, pos);
858.   mem[location + 2 + pos] = val;
859. }
860.
861. int vector_set(int v, int pos, int val)
862. {
863.   c_vector_set(v, to_c_number(pos), val);
864.   return scm_true;
865. }
866.
867. int vector_length(int v)
868. {
869.   return TO_SCM_NUMBER(C_VECTOR_LEN(v));
870. }
871.
872. int vector_len_to_int_vector_len(int len)
873. {
874.   return (len == 0) ? 1 : len;
875. }
876.
877. int c_vector_int_len(int v)
878. {
879.   return vector_len_to_int_vector_len(C_VECTOR_LEN(v));
880. }
881.
882.
883.
884.
885. /* Fonctions relatives a PROCEDURE ----------------------------- */
886.
887. int pred_procedure(int d)
888. {
889.   return (C_PRED_CPRIM(d) || C_PRED_CLOSURE(d)) ? scm_true : scm_false;
890. }
891.
892. int make_closure(int entry, int env)
893. {
894.   int cell, handle, closure;
895.
896.   make_clos_env = env;
897.   closure = alloc_cell(3, TYPECLOS);
898.   handle = SCM_TO_HANDLE(closure);
899.   cell = mem[handle];
900.   mem[cell + 1] = entry;
901.   mem[cell + 2] = make_clos_env;
902.   return closure;
903. }
904.
905. int closure_entry(int c)
906. {
907.   return CHAMP(c, 0);
908. }
909.
910. int closure_env(int c)
911. {
912.   return CHAMP(c, 1);
913. }
914.
915. /* Cette fonction ne peut appeler direct. c_apply */
916. /* Note: elle utilise le apply-hook a la fin du bytecode */
917. int apply(int c, int args)
918. {
919.   eval_pc = bytecode_len - 1;
920.   eval_args = cons(c, args);
921.   return scm_false;
922. }
923.
924.
925.
926.
927. /* Fonctions relatives a CONT ---------------------------------- */
928.
929. int c_pred_cont(int d)
930. {
931.   return (C_PRED_OTHER(d) &&
932.           ((mem[mem[SCM_TO_HANDLE(d)] + 1] & CONTMASK) == CONTVAL));
933. }
934.
935. int make_cont(void)
936. {
937.   int cell, handle, k;
938.
939.   k = alloc_cell(5, TYPEOTHER);
940.   handle = SCM_TO_HANDLE(k);
941.   cell = mem[handle];
942.   mem[cell + 1] = PC_TO_PCTAG(eval_pc);
943.   mem[cell + 2] = eval_env;
944.   mem[cell + 3] = eval_args;
945.   mem[cell + 4] = eval_cont;
946.   return k;
947. }
```

```
948.
949. void set_cont_pc(int k, int dest)
950. {
951.   CHAMP(k, 0) = PC_TO_PCTAG(dest);
952. }
953.
954. void restore_cont(int k)
955. {
956.   int cell;
957.
958.   cell = mem[SCM_TO_HANDLE(k)];
959.   eval_pc = PCTAG_TO_PC(mem[cell + 1]);
960.   eval_env = mem[cell + 2];
961.   eval_args = mem[cell + 3];
962.   eval_cont = mem[cell + 4];
963. }
964.
965.
966.
967.
968. /* Fonctions d'entree/sortie ------------------------------------ */
969.
970. int ll_input(void)
971. {
972.   int c;
973.
974.   c = getchar();
975.   if (c == EOF)
976.     {
977.       printf("EOF\nQuit\n");
978.       exit(0);
979.     }
980.   return c;
981. }
982.
983. static int look_ahead;
984. static int look_ahead_valide;
985.
986. int c_peek_char(void)
987. {
988.   if (!look_ahead_valide)
989.     {
990.       look_ahead = ll_input();
991.       look_ahead_valide = TRUE;
992.     }
993.   return look_ahead;
994. }
995.
996. int peek_char(void)
997. {
998.   return TO_SCM_CHAR(c_peek_char());
999. }
1000.
1001. int c_read_char(void)
1002. {
1003.   if (look_ahead_valide)
1004.     {
1005.       look_ahead_valide = FALSE;
1006.       return look_ahead;
1007.     }
1008.   else
1009.     return ll_input();
1010. }
1011.
1012. int read_char(void)
1013. {
1014.   return TO_SCM_CHAR(c_read_char());
1015. }
1016.
1017. int write_char(int c)
1018. {
1019.   printf("%c", TO_C_CHAR(c));
1020.   return scm_true;
1021. }
1022.
1023.
1024.
1025.
1026. /* Autres fonctions de la librairie ---------------------------- */
1027.
1028. int eq(int d1, int d2)
1029. {
1030.   return (d1 == d2) ? scm_true : scm_false;
1031. }
1032.
1033. int quit(void)
1034. {
1035.   printf("Quit\n");
1036.   exit(0);
1037. }
1038.
1039. int return_current_continuation(void)
1040. {
1041.   int temp;
1042.
1043.   temp = make_cont();
1044.   return cons(temp, eval_prev_args);
1045. }
1046.
1047. int return_there_with_this(int complete_cont, int val)
1048. {
1049.   eval_prev_args = cdr(complete_cont);
1050.   restore_cont(car(complete_cont));
1051.   return val;
1052. }
1053.
1054.
1055.
1056.
1057. /* Fonctions relatives au coeur de l'interprete ----------------- */
1058.
1059. static int intro_state;
1060.
1061. int introspection(int arg)
1062. {
1063.   if (intro_state == 0)
1064.     {
1065.       intro_state = 1;
1066.       return scm_constants;
1067.     }
1068.   else
1069.     {
1070.       scm_constants = arg;
1071.       return scm_false;
1072.     }
1073. }
1074.
1075.
```

```
1076.
1077.
1078. /* Le coeur de l'interprete ------------------------------------- */
1079.
1080. void init_bc_reader(void)
1081. {
1082.   eval_pc = 0;
1083.   return;
1084. }
1085.
1086. int read_bc_byte(void)
1087. {
1088.   int b;
1089.
1090.   b = (int) bytecode[eval_pc];
1091.   eval_pc++;
1092.   return b;
1093. }
1094.
1095. int read_bc_int(void)
1096. {
1097.   int msb;
1098.
1099.   msb = read_bc_byte();
1100.   return (256 * msb + read_bc_byte());
1101. }
1102.
1103. int get_frame(int step)
1104. {
1105.   int frame;
1106.
1107.   frame = eval_env;
1108.   while (step > 0)
1109.     {
1110.       if (C_PRED_PAIR(frame))
1111.         frame = car(frame);
1112.       else
1113.         frame = c_vector_ref(frame, 0);
1114.       step --;
1115.     }
1116.   return frame;
1117. }
1118.
1119. int get_var(int frame, int offset)
1120. {
1121.   int f;
1122.
1123.   f = get_frame(frame);
1124.   if (C_PRED_PAIR(f))
1125.     return cdr(f);
1126.   else
1127.     return c_vector_ref(f, 1 + offset);
1128. }
1129.
1130. void set_var(int frame, int offset, int val)
1131. {
1132.   int f;
1133.
1134.   f = get_frame(frame);
1135.   if (C_PRED_PAIR(f))
1136.     set_cdr(f, val);
1137.   else
1138.     c_vector_set(f, 1 + offset, val);
1139. }
1140.
1141. void make_normal_frame(int size)
1142. {
1143.   int pos;
1144.   int larg, frame;
1145.
1146.   if (size == 1)
1147.     eval_env = cons(eval_env, car(eval_args));
1148.   else
1149.     {
1150.       frame = c_make_vector(size + 1);
1151.       c_vector_set(frame, 0, eval_env);
1152.       larg = eval_args;
1153.       for (pos=1 ; pos<=size ; pos++)
1154.         {
1155.           c_vector_set(frame, pos, car(larg));
1156.           larg = cdr(larg);
1157.         }
1158.       eval_env = frame;
1159.     }
1160. }
1161.
1162. void make_rest_frame(int size)
1163. {
1164.   int pos;
1165.   int larg;
1166.   int temp;
1167.
1168.   if (size == 1)
1169.     {
1170.       temp = list_copy(eval_args);
1171.       eval_env = cons(eval_env, temp);
1172.     }
1173.   else
1174.     {
1175.       make_rest_frame_frame = c_make_vector(1 + size);
1176.       c_vector_set(make_rest_frame_frame, 0, eval_env);
1177.       larg = eval_args;
1178.       for (pos=1 ; pos<size ; pos++)
1179.         {
1180.           c_vector_set(make_rest_frame_frame, pos, car(larg));
1181.           larg = cdr(larg);
1182.         }
1183.       temp = list_copy(larg);
1184.       c_vector_set(make_rest_frame_frame, size, temp);
1185.       eval_env = make_rest_frame_frame;
1186.     }
1187. }
1188.
1189. void push_arg(int arg)
1190. {
1191.   eval_args = cons(arg, eval_args);
1192. }
1193.
1194. int pop_arg(void)
1195. {
1196.   int temp;
1197.
1198.   temp = car(eval_args);
1199.   eval_args = cdr(eval_args);
1200.   return temp;
1201. }
1202.
1203. void pop_n_args(int n)
```

```
1204. {
1205.   int i;
1206.
1207.   for (i=0 ; i<n ; i++)
1208.     pop_arg();
1209. }
1210.
1211. int apply_cprim(int cprim_no, int args)
1212. {
1213.   int arg1, arg2, arg3;
1214.
1215.   if (cprim_no < FIRSTCPRIM1)
1216.     return (*cprim0[cprim_no])();
1217.   arg1 = car(args);
1218.   if (cprim_no < FIRSTCPRIM2)
1219.     return (*cprim1[cprim_no - FIRSTCPRIM1])(arg1);
1220.   args = cdr(args);
1221.   arg2 = car(args);
1222.   if (cprim_no < FIRSTCPRIM3)
1223.     return (*cprim2[cprim_no - FIRSTCPRIM2])(arg1, arg2);
1224.   args = cdr(args);
1225.   arg3 = car(args);
1226.   return (*cprim3[cprim_no - FIRSTCPRIM3])(arg1, arg2, arg3);
1227. }
1228.
1229. int apply_closure(int c, int args)
1230. {
1231.   eval_pc = closure_entry(c);
1232.   eval_args = args;
1233.   eval_env = closure_env(c);
1234.   return scm_false;
1235. }
1236.
1237. void c_apply(int c, int args)
1238. {
1239.   int cprim_no;
1240.   int temp;
1241.
1242.   if (C_PRED_CPRIM(c))  /* Dans une application generale, */
1243.     {                   /* il faut forcer la restoration */
1244.       cprim_no = CPRIM_NUMBER(c);
1245.       temp = apply_cprim(cprim_no, args);
1246.       if (cprim_no != APPLY_CPRIM_NO)
1247.         {
1248.           restore_cont(eval_cont);
1249.           push_arg(temp);
1250.         }
1251.     }
1252.   else
1253.     apply_closure(c, args);
1254. }
1255.
1256. void execute_bc(void)
1257. {
1258.   int op, dest, frameno, offsetno;
1259.   int frame;
1260.   int (*read_bc_info)(void);
1261.
1262.   init_bc_reader();
1263.   while (TRUE)
1264.     {
1265.       op = read_bc_byte();
1266.       switch (op)
1267.         {
1268.         case 0:
1269.           {
1270.             read_bc_info = read_bc_byte;
1271.             goto label002;
1272.           }
1273.         case 1:
1274.           {
1275.             read_bc_info = read_bc_int;
1276.           label002:
1277.             push_arg(c_vector_ref(scm_constants, read_bc_info()));
1278.             break;
1279.           }
1280.         case 2:
1281.           {
1282.             read_bc_info = read_bc_byte;
1283.             goto label003;
1284.           }
1285.         case 3:
1286.           {
1287.             read_bc_info = read_bc_int;
1288.           label003:
1289.             frame = get_frame(read_bc_info());
1290.             if (C_PRED_PAIR(frame))
1291.               eval_val = cdr(frame);
1292.             else
1293.               eval_val = c_vector_ref(frame, 1 + read_bc_info());
1294.             push_arg(eval_val);
1295.             break;
1296.           }
1297.         case 4:
1298.           {
1299.             read_bc_info = read_bc_byte;
1300.             goto label004;
1301.           }
1302.         case 5:
1303.           {
1304.             read_bc_info = read_bc_int;
1305.           label004:
1306.             push_arg(scm_globs[read_bc_info()]);
1307.             break;
1308.           }
1309.         case 6:
1310.           {
1311.             read_bc_info = read_bc_byte;
1312.             goto label005;
1313.           }
1314.         case 7:
1315.           {
1316.             read_bc_info = read_bc_int;
1317.           label005:
1318.             frame = get_frame(read_bc_info());
1319.             eval_val = pop_arg();
1320.             if (C_PRED_PAIR(frame))
1321.               set_cdr(frame, eval_val);
1322.             else
1323.               c_vector_set(frame, 1 + read_bc_info(), eval_val);
1324.             push_arg(scm_false);
1325.             break;
1326.           }
1327.         case 8:
1328.           {
1329.             read_bc_info = read_bc_byte;
1330.             goto label006;
1331.           }
```

```
1332.         case 9:
1333.           {
1334.             read_bc_info = read_bc_int;
1335.           label006:
1336.             eval_val = pop_arg();
1337.             if (gc_mark)
1338.               mark_it(eval_val);
1339.             scm_globs[read_bc_info()] = eval_val;
1340.             push_arg(scm_false);
1341.             break;
1342.           }
1343.         case 10:
1344.           {
1345.             eval_val = make_closure(eval_pc, eval_env);
1346.             restore_cont(eval_cont);
1347.             push_arg(eval_val);
1348.             break;
1349.           }
1350.         case 11:
1351.           {
1352.             dest = read_bc_int();
1353.             if (pop_arg() == scm_false)
1354.               eval_pc = dest;
1355.             break;
1356.           }
1357.         case 19:
1358.           {
1359.             set_cont_pc(eval_cont, eval_pc + 2);
1360.             /* continue to 13 */
1361.           }
1362.         case 13:
1363.           {
1364.             eval_env = scm_empty;
1365.             /* continue to 12 */
1366.           }
1367.         case 12:
1368.           {
1369.             eval_pc = read_bc_int();
1370.             break;
1371.           }
1372.         case 14:
1373.           {
1374.             eval_val = eval_args;
1375.             restore_cont(eval_cont);
1376.             set_cdr(eval_val, eval_args);
1377.             eval_args = eval_val;
1378.             break;
1379.           }
1380.         case 25:
1381.           {
1382.             eval_prev_args = cons(eval_args, eval_prev_args);
1383.             /* continue to 15 */
1384.           }
1385.         case 15:
1386.           {
1387.             eval_args = scm_empty;
1388.             break;
1389.           }
1390.         case 16:
1391.           {
1392.             fprintf(stderr, "Ce n'est plus suppose etre utilise!\n");
1393.             break;
1394.           }
1395.         case 46:
1396.           {
1397.             set_cont_pc(eval_cont, eval_pc);
1398.             /* continue to 17 */
1399.           }
1400.         case 17:
1401.           {
1402.             c_apply(car(eval_args), cdr(eval_args));
1403.             break;
1404.           }
1405.         case 18:
1406.           {
1407.             fprintf(stderr, "Ce n'est plus suppose etre utilise!\n");
1408.             break;
1409.           }
1410.         case 20:
1411.           {
1412.             read_bc_info = read_bc_byte;
1413.             goto label007;
1414.           }
1415.         case 21:
1416.           {
1417.             read_bc_info = read_bc_int;
1418.           label007:
1419.             make_normal_frame(read_bc_info());
1420.             break;
1421.           }
1422.         case 22:
1423.           {
1424.             read_bc_info = read_bc_byte;
1425.             goto label008;
1426.           }
1427.         case 23:
1428.           {
1429.             read_bc_info = read_bc_int;
1430.           label008:
1431.             make_rest_frame(read_bc_info());
1432.             break;
1433.           }
1434.         case 24:
1435.           {
1436.             return;
1437.             break;
1438.           }
1439.         case 26:
1440.           {
1441.             eval_val = eval_args;
1442.             eval_args = car(eval_prev_args);
1443.             eval_prev_args = cdr(eval_prev_args);
1444.             set_cdr(eval_val, eval_args);
1445.             eval_args = eval_val;
1446.             break;
1447.           }
1448.         case 27:
1449.           {
1450.             push_arg(scm_empty);
1451.             break;
1452.           }
1453.         case 28:
1454.           {
1455.             push_arg(scm_false);
1456.             break;
1457.           }
1458.         case 29:
1459.           {
```

```
1460.            push_arg(scm_true);
1461.            break;
1462.          }
1463.        case 30:
1464.          {
1465.            push_arg(TO_SCM_CHAR(read_bc_byte()));
1466.            break;
1467.          }
1468.        case 31:
1469.          {
1470.            read_bc_info = read_bc_byte;
1471.            goto label009;
1472.          }
1473.        case 32:
1474.          {
1475.            read_bc_info = read_bc_byte;
1476.            goto label010;
1477.          }
1478.        case 33:
1479.          {
1480.            read_bc_info = read_bc_int;
1481.          label009:
1482.            push_arg(TO_SCM_NUMBER(read_bc_info()));
1483.            break;
1484.          }
1485.        case 34:
1486.          {
1487.            read_bc_info = read_bc_int;
1488.          label010:
1489.            push_arg(TO_SCM_NUMBER(-read_bc_info()));
1490.            break;
1491.          }
1492.        case 35:
1493.          {
1494.            eval_env = get_frame(1);
1495.            break;
1496.          }
1497.        case 36:
1498.          {
1499.            frameno = 0;
1500.            offsetno = 0;
1501.            goto label011;
1502.          }
1503.        case 37:
1504.          {
1505.            frameno = 0;
1506.            offsetno = 1;
1507.            goto label011;
1508.          }
1509.        case 38:
1510.          {
1511.            frameno = 0;
1512.            offsetno = 2;
1513.            goto label011;
1514.          }
1515.        case 39:
1516.          {
1517.            frameno = 1;
1518.            offsetno = 0;
1519.            goto label011;
1520.          }
1521.        case 40:
1522.          {
1523.            frameno = 1;
1524.            offsetno = 1;
1525.            goto label011;
1526.          }
1527.        case 41:
1528.          {
1529.            frameno = 2;
1530.            offsetno = 0;
1531.          label011:
1532.            push_arg(get_var(frameno, offsetno));
1533.            break;
1534.          }
1535.        case 42:
1536.          {
1537.            make_normal_frame(1);
1538.            break;
1539.          }
1540.        case 43:
1541.          {
1542.            make_normal_frame(2);
1543.            break;
1544.          }
1545.        case 44:
1546.          {
1547.            make_rest_frame(1);
1548.            break;
1549.          }
1550.        case 45:
1551.          {
1552.            eval_cont = make_cont();
1553.            eval_args = scm_empty;
1554.            break;
1555.          }
1556.        case 47:
1557.          {
1558.            fprintf(stderr, "Ce n'est plus suppose etre utilise!\n");
1559.            break;
1560.          }
1561.        case 48:
1562.          {
1563.            dest = read_bc_int();
1564.            push_arg(make_closure(eval_pc, eval_env));
1565.            eval_pc = dest;
1566.            break;
1567.          }
1568.        case 49:
1569.          {
1570.            read_bc_info = read_bc_byte;
1571.            goto label012;
1572.          }
1573.        case 50:
1574.          {
1575.            read_bc_info = read_bc_int;
1576.          label012:
1577.            pop_n_args(read_bc_info());
1578.            break;
1579.          }
1580.        case 51:
1581.          {
1582.            pop_arg();
1583.            break;
1584.          }
1585.        case 54:
1586.          {
1587.            set_cont_pc(eval_cont, eval_pc + 1);
```

```
1588.            /* continue to 52 */
1589.          }
1590.        case 52:
1591.          {
1592.            read_bc_info = read_bc_byte;
1593.            goto label013;
1594.          }
1595.        case 55:
1596.          {
1597.            set_cont_pc(eval_cont, eval_pc + 2);
1598.            /* continue to 53 */
1599.          }
1600.        case 53:
1601.          {
1602.            read_bc_info = read_bc_int;
1603.          label013:
1604.            dest = scm_globs[read_bc_info()];
1605.            c_apply(dest, eval_args);
1606.            break;
1607.          }
1608.        default:
1609.          {
1610.            push_arg(apply_cprim(255 - op, eval_args));
1611.            break;
1612.          }
1613.        }
1614.    }
1615. }
1616.
1617.
1618.
1619.
1620. /* Le programme principal ------------------------------------- */
1621.
1622. int init_scm_glob_1(int code)
1623. {
1624.   if (code == -1)      /* Sans valeur initiale */
1625.     return scm_false;
1626.   else if (code < 0)   /* Cprim */
1627.     return C_MAKE_CPRIM(-2 - code);
1628.   else                 /* Fermeture */
1629.     return TO_SCM_NUMBER(code);
1630. }
1631.
1632. int init_scm_glob_2(int code)
1633. {
1634.   int entry;
1635.
1636.   if (C_PRED_NUMBER(code))
1637.     {
1638.       entry = code >> 1;  /* Et non pas to_c_number(code) */
1639.       return make_closure(entry, scm_empty);
1640.     }
1641.   else
1642.     return code;
1643. }
1644.
1645. int main(int argc, char *argv[])
1646. {
1647.   int i;
1648.
1649.   if ((1 << LOGCPI) != CHARSPERINT)
1650.     {
1651.       fprintf(stderr, "Verifier LOGCPI.");
1652.       exit(1);
1653.     }
1654.
1655.   alloc_heap(DEFAULTHEAPSIZE);
1656.
1657.   /* Attention, l'ordre est important! GC, scm_false, etc. */
1658.   scm_false = FALSEVAL;
1659.   for (i=0 ; i<NBGLOBS ; i++)
1660.     globs[i] = scm_false;
1661.   scm_empty = NULLVAL;
1662.   scm_true = TRUEVAL;
1663.   eval_prev_args = scm_empty;
1664.
1665.   for (i=0 ; i<nb_scm_globs ; i++)  /* Temporairement */
1666.     scm_globs[i] = init_scm_glob_1(scm_globs[i]);
1667.
1668.   nb_symbols = 0;
1669.   scm_symbol_names = c_make_vector(1);
1670.   scm_constants = to_scm_string(const_desc, const_desc_len);
1671.
1672.   for (i=0 ; i<nb_scm_globs ; i++)
1673.     scm_globs[i] = init_scm_glob_2(scm_globs[i]);
1674.
1675.   intro_state = 0;
1676.   look_ahead_valide = FALSE;
1677.
1678.   execute_bc();
1679.   return 0;
1680. }
1681.
1682. /* Ajuster la limite de taille du bytecode dans "analyse.scm" */
```

A.2 Le fichier de librairie

Tel que décrit à la section 5.4.2, le fichier de librairie se divise en quatre parties: déclaration des fonctions primitives, définition des fonctions non-standard cachées, définition des fonctions non-standard visibles, définition des fonctions standard.

```
 1. ;
 2. ; Fonctions implantees dans le noyau. Pour savoir lesquelles
 3. ; sont visibles, voir sections plus bas
 4. ;
 5.
 6. (peek-char                    .  0)
 7. (read-char                    .  1)
 8. (quit                         .  2)
 9. (return-current-continuation  .  3)
10. (boolean?                     .  4)
11. (pair?                        .  5)
12. (car                          .  6)
13. (cdr                          .  7)
14. (char?                        .  8)
15. (integer->char                .  9)
16. (char->integer                . 10)
17. (string?                      . 11)
18. (make-string1                 . 12)
19. (string-length                . 13)
20. (string-copy                  . 14)
21. (symbol?                      . 15)
22. (symbol->string               . 16)
23. (string->symbol               . 17)
24. (number?                      . 18)
25. (vector?                      . 19)
26. (make-vector1                 . 20)
27. (vector-length                . 21)
28. (procedure?                   . 22)
29. (write-char                   . 23)
30. (introspection                . 24)
31. (cons                         . 25)
32. (set-car!                     . 26)
33. (set-cdr!                     . 27)
34. (string-ref                   . 28)
35. (string=?                     . 29)
36. (cppoe2                       . 30)
37. (cplus2                       . 31)
38. (cmoins2                      . 32)
39. (cfois2                       . 33)
40. (cdivise2                     . 34)
41. (vector-ref                   . 35)
42. (apply1                       . 36)
43. (eq?                          . 37)
44. (return-there-with-this       . 38)
45. (string-set!                  . 39)
46. (vector-set!                  . 40)
47.
48. #f ; Fin des definitions des primitives C
49.
50. ;
51. ; Fonctions cachees de la librairie
52. ;
53.
54. (define foldl
55.   (lambda (binop start l)
56.     (if (null? l)
57.         start
58.         (foldl binop (binop start (car l)) (cdr l)))))
59. (define foldl1
60.   (lambda (binop l)
61.     (if (null? (cdr l))
62.         (car l)
63.         (foldl1 binop (cons (binop (car l) (cadr l))
64.                             (cddr l))))))
65. (define foldr1
66.   (lambda (binop l)
67.     (if (null? (cdr l))
68.         (car l)
69.         (binop (car l) (foldr1 binop (cdr l))))))
70.
71. (define generic-member
72.   (lambda (releq obj list)
73.     (if (null? list)
74.         #f
75.         (if (releq (car list) obj)
76.             list
77.             (generic-member releq obj (cdr list))))))
78.
79. (define generic-assoc
80.   (lambda (releq obj alist)
81.     (cond ((null? alist)
82.            #f)
83.           ((releq (car (car alist)) obj)
84.            (car alist))
85.           (else
86.            (generic-assoc releq obj (cdr alist))))))
87.
88. (define math=2
89.   (lambda (x y)
90.     (if (math<=2 x y) (math<=2 y x) #f)))
91.
92. (define math<2
93.   (lambda (x y)
94.     (not (math<=2 y x))))
95.
96. (define math>2
97.   (lambda (x y)
98.     (not (math<=2 x y))))
99.
100. (define math<=2 cppoe2)
101.
102. (define math>=2 (lambda (x y) (math<=2 y x)))
103.
104. (define generic-compare
105.   (lambda (binrel l)
106.     (if (null? (cddr l))
107.         (binrel (car l) (cadr l))
108.         (and (binrel (car l) (cadr l))
109.              (generic-compare binrel (cdr l))))))
110.
111. (define max2 (lambda (x y) (if (> x y) x y)))
```

```
(define min2 (lambda (x y) (if (< x y) x y)))

(define math+2 cplus2)

(define math*2 cfois2)

(define math-2 cmoins2)

(define math/2 cdivise2)

(define math%2
  (lambda (num den)
    (math-2 num (math*2 den (math/2 num den)))))

(define mathgcd2
  (lambda (n1 n2)
    (let loop ((n1 (abs n1)) (n2 (abs n2)))
      (cond ((zero? n1) n2)
            ((zero? n2) n1)
            (else
             (let ((grand (max n1 n2))
                   (petit (min n1 n2)))
               (loop petit (modulo grand petit))))))))

(define mathlcm2
  (lambda (n1 n2)
    (cond ((zero? n1) (abs n2))
          ((zero? n2) (abs n1))
          (else
           (let ((n1 (abs n1))
                 (n2 (abs n2)))
             (/ (* n1 n2) (mathgcd2 n1 n2)))))))

(define string-compare
  (lambda (rel<? rel=? s1 s2)
    (let* ((len1 (string-length s1))
           (len2 (string-length s2))
           (len (min len1 len2)))
      (let loop ((pos 0))
        (if (< pos len)
            (let* ((c1 (string-ref s1 pos))
                   (c2 (string-ref s2 pos)))
              (cond ((rel<? c1 c2)
                     -1)
                    ((rel=? c1 c2)
                     (loop (+ pos 1)))
                    (else
                     1)))
            (cond ((< len1 len2)
                   -1)
                  ((= len1 len2)
                   0)
                  (else
                   1)))))))

(define map1
  (lambda (f l)
    (if (null? l)
        l
        (cons (f (car l)) (map1 f (cdr l))))))

(define write-many-chars
  (lambda l
    (for-each write-char l)))

(define write-cdr
  (lambda (d parent)
    (cond ((eq? d '())
           (write-char #\)))
          ((pair? d)
           (write-char #\space)
           (parent (car d))
           (write-cdr (cdr d) parent))
          (else
           (write-many-chars #\space #\. #\space)
           (parent d)
           (write-char #\)))))))

(define write-vector
  (lambda (d parent)
    (let ((len (vector-length d)))
      (write-many-chars #\# #\()
      (if (> len 0)
          (begin
            (parent (vector-ref d 0))
            (let loop ((pos 1))
              (if (< pos len)
                  (begin
                    (write-char #\space)
                    (parent (vector-ref d pos))
                    (loop (+ pos 1)))))))
      (write-char #\)))))

(define make-byte-reader
  (lambda (s)
    (let ((pos 0))
      (lambda ()
        (let ((c (string-ref s pos)))
          (set! pos (+ pos 1))
          c))))))
(define make-number-reader
  (lambda (read-const-byte)
    (lambda ()
      (let* ((msc (read-const-byte))
             (lsc (read-const-byte))
             (msb (char->integer msc))
             (lsb (char->integer lsc)))
        (+ (* msb 256) lsb)))))
(define read-const-desc
  (lambda (const-v pos read-const-byte read-const-number)
    (let ((type (read-const-byte)))
      (cond
       ((char=? type #\0)  ; EMPTY
        '())
       ((char=? type #\1)  ; PAIR
        (let* ((incar (vector-ref const-v (read-const-number)))
               (incdr (vector-ref const-v (read-const-number))))
          (cons incar incdr)))
       ((char=? type #\2)  ; BOOLEAN
        (char=? (read-const-byte) #\t))
       ((char=? type #\3)  ; CHAR
        (read-const-byte))
       ((char=? type #\4)  ; STRING
        (let* ((len (read-const-number))
               (s (make-string len)))
          (let loop ((pos 0))
            (if (< pos len)
                (begin
```

```
240.                  (string-set! s pos (read-const-byte))
241.                  (loop (+ pos 1)))))
242.          s))
243.       ((char=? type #\5)  ; SYMBOL
244.        (string->symbol (vector-ref const-v (read-const-number))))
245.       ((char=? type #\6)  ; NUMBER
246.        (let* ((sign (read-const-byte))
247.               (valabs (read-const-number)))
248.          (if (char=? sign #\+)
249.              valabs
250.              (- valabs))))
251.       ((char=? type #\7)  ; VECTOR
252.        (let* ((len (read-const-number))
253.               (v (make-vector len)))
254.          (let loop ((pos 0))
255.            (if (< pos len)
256.                (begin
257.                  (vector-set! v pos (vector-ref const-v (read-const-number)))
258.                  (loop (+ pos 1)))))
259.          v))))))
260. (define extract-top-const
261.   (lambda (const-v read-const-number)
262.     (let* ((nbtop (read-const-number))
263.            (top-v (make-vector nbtop)))
264.       (let loop ((pos 0))
265.         (if (< pos nbtop)
266.             (begin
267.               (vector-set! top-v pos (vector-ref const-v (read-const-number)))
268.               (loop (+ pos 1)))))
269.       top-v)))
270.
271. #f ; Fin des definitions des fonctions internes
272.
273. ;
274. ; Les fonctions non-standard mais visibles tout
275. ; de meme pour les programmes compiles
276. ;
277.
278. (define append2
279.   (lambda (l1 l2)
280.     (if (null? l1)
281.         l2
282.         (let ((tete (cons (car l1) l2)))
283.           (let loop ((cur tete) (l1 (cdr l1)))
284.             (if (null? l1)
285.                 tete
286.                 (begin
287.                   (set-cdr! cur (cons (car l1) l2))
288.                   (loop (cdr cur) (cdr l1)))))))))
289.
290. quit
291.
292. (define make-promise
293.   (lambda (proc)
294.     (let ((result-ready? #f)
295.           (result #f))
296.       (lambda ()
297.         (if result-ready?
298.             result
299.             (let ((x (proc)))
300.               (if result-ready?
301.                   result
302.                   (begin (set! result-ready? #t)
303.                          (set! result x)
304.                          result))))))))
305.
306. ; Note tres importante: cette fonction sert a reconstituer les constantes
307. ; du programme avant le debut de son execution. Toute fonction appelee
308. ; durant l'execution de celle-ci ne doit pas comporter de constantes etant
309. ; donne qu'elles ne sont pas encore baties.
310. (define install-const
311.   (lambda ()
312.     (let* ((const-string (introspection #f))  ; Porte secrete!
313.            (read-const-byte (make-byte-reader const-string))
314.            (read-const-number (make-number-reader read-const-byte))
315.            (nbconst (read-const-number))
316.            (const-v (make-vector nbconst)))
317.       (let loop ((pos 0))
318.         (if (< pos nbconst)
319.             (begin
320.               (vector-set! const-v
321.                            pos
322.                            (read-const-desc const-v
323.                                             pos
324.                                             read-const-byte
325.                                             read-const-number))
326.               (loop (+ pos 1)))))
327.       (let ((top-v (extract-top-const const-v read-const-number)))
328.         (introspection top-v)))))            ; Porte secrete!
329.
330. #f ; Fin des definitions des fonctions non-standard visibles
331.
332. ;
333. ; Debut des fonctions Scheme standard de la librairie
334. ;
335.
336. ; 6.1
337. (define not (lambda (x) (if x #f #t)))
338. boolean?
339.
340. ; 6.2
341. (define eqv?
342.   (lambda (d1 d2)
343.     (cond ((and (number? d1) (number? d2))
344.            (= d1 d2))
345.           ((and (char? d1) (char? d2))
346.            (char=? d1 d2))
347.           (else
348.            (eq? d1 d2)))))
349. eq?
350. (define equal?
351.   (lambda (d1 d2)
352.     (cond ((eqv? d1 d2)
353.            #t)
354.           ((and (pair? d1) (pair? d2))
355.            (and (equal? (car d1) (car d2)) (equal? (cdr d1) (cdr d2))))
356.           ((and (vector? d1) (vector? d2))
357.            (let ((len (vector-length d1)))
358.              (if (not (= len (vector-length d2)))
359.                  #f
360.                  (let loop ((pos 0))
361.                    (cond ((>= pos len)
362.                           #t)
363.                          ((equal? (vector-ref d1 pos) (vector-ref d2 pos))
364.                           (loop (+ pos 1)))
365.                          (else
366.                           #f))))))
367.           ((and (string? d1) (string? d2))
```

```
368.           (string=? d1 d2))
369.          (else
370.           #f))))
371.
372. ; 6.3
373. pair?
374. cons
375. car
376. cdr
377. set-car!
378. set-cdr!
379. (define caar (lambda (p) (car (car p))))
380. (define cadr (lambda (p) (car (cdr p))))
381. (define cdar (lambda (p) (cdr (car p))))
382. (define cddr (lambda (p) (cdr (cdr p))))
383. (define caaar (lambda (p) (caar (car p))))
384. (define caadr (lambda (p) (caar (cdr p))))
385. (define cadar (lambda (p) (cadr (car p))))
386. (define caddr (lambda (p) (cadr (cdr p))))
387. (define cdaar (lambda (p) (cdar (car p))))
388. (define cdadr (lambda (p) (cdar (cdr p))))
389. (define cddar (lambda (p) (cddr (car p))))
390. (define cdddr (lambda (p) (cddr (cdr p))))
391. (define caaaar (lambda (p) (caaar (car p))))
392. (define caaadr (lambda (p) (caaar (cdr p))))
393. (define caadar (lambda (p) (caadr (car p))))
394. (define caaddr (lambda (p) (caadr (cdr p))))
395. (define cadaar (lambda (p) (cadar (car p))))
396. (define cadadr (lambda (p) (cadar (cdr p))))
397. (define caddar (lambda (p) (caddr (car p))))
398. (define cadddr (lambda (p) (caddr (cdr p))))
399. (define cdaaar (lambda (p) (cdaar (car p))))
400. (define cdaadr (lambda (p) (cdaar (cdr p))))
401. (define cdadar (lambda (p) (cdadr (car p))))
402. (define cdaddr (lambda (p) (cdadr (cdr p))))
403. (define cddaar (lambda (p) (cddar (car p))))
404. (define cddadr (lambda (p) (cddar (cdr p))))
405. (define cdddar (lambda (p) (cdddr (car p))))
406. (define cddddr (lambda (p) (cdddr (cdr p))))
407. (define null?
408.   (lambda (x) (eq? x '())))
409. (define list?
410.   (lambda (l)
411.     (cond ((null? l)
412.            #t)
413.           ((not (pair? l))
414.            #f)
415.           (else
416.            (let loop ((slow l) (fast (cdr l)) (phase 2))
417.              (cond ((null? fast)
418.                     #t)
419.                    ((not (pair? fast))
420.                     #f)
421.                    ((eq? slow fast)
422.                     #f)
423.                    ((= phase 1)
424.                     (loop slow (cdr fast) 2))
425.                    (else
426.                     (loop (cdr slow) (cdr fast) 1))))))))
427. (define list (lambda l l))
428. (define length
429.   (lambda (l)
430.     (let loop ((l l) (len 0))
431.       (if (null? l)
432.           len
433.           (loop (cdr l) (+ len 1))))))
434. (define append
435.   (lambda ll
436.     (foldr1 append2 (cons '() ll))))
437. (define reverse
438.   (lambda (l)
439.     (let loop ((l l) (rl '()))
440.       (if (null? l)
441.           rl
442.           (loop (cdr l) (cons (car l) rl))))))
443. (define list-tail
444.   (lambda (l pos)
445.     (if (= pos 0)
446.         l
447.         (list-tail (cdr l) (- pos 1)))))
448. (define list-ref (lambda (l pos) (car (list-tail l pos))))
449. (define memq
450.   (lambda (obj list)
451.     (generic-member eq? obj list)))
452. (define memv
453.   (lambda (obj list)
454.     (generic-member eqv? obj list)))
455. (define member
456.   (lambda (obj list)
457.     (generic-member equal? obj list)))
458. (define assq (lambda (obj alist) (generic-assoc eq? obj alist)))
459. (define assv (lambda (obj alist) (generic-assoc eqv? obj alist)))
460. (define assoc (lambda (obj alist) (generic-assoc equal? obj alist)))
461.
462. ; 6.4
463. symbol?
464. symbol->string
465. string->symbol
466.
467. ; 6.5
468. number?
469. (define complex? number?)
470. (define real? number?)
471. (define rational? number?)
472. (define integer? number?)
473. (define exact? (lambda (n) #t))
474. (define inexact? (lambda (n) #f))
475. (define =  (lambda l (generic-compare math=2 l)))
476. (define <  (lambda l (generic-compare math<2 l)))
477. (define >  (lambda l (generic-compare math>2 l)))
478. (define <= (lambda l (generic-compare math<=2 l)))
479. (define >= (lambda l (generic-compare math>=2 l)))
480. (define zero? (lambda (n) (= n 0)))
481. (define positive? (lambda (n) (> n 0)))
482. (define negative? (lambda (n) (< n 0)))
483. (define odd? (lambda (n) (= (math%2 (abs n) 2) 1)))
484. (define even? (lambda (n) (= (math%2 (abs n) 2) 0)))
485. (define max (lambda l (foldl1 max2 l)))
486. (define min (lambda l (foldl1 min2 l)))
487. (define + (lambda l (foldl math+2 0 l)))
488. (define * (lambda l (foldl math*2 1 l)))
489. (define - (lambda l (if (null? (cdr l)) (math-2 0 (car l)) (foldl1 math-2 l))))
490. (define /
491.   (lambda l (if (null? (cdr l)) (quotient 1 (car l)) (foldl1 quotient l))))
492. (define abs (lambda (n) (if (negative? n) (- n) n)))
493. (define quotient
494.   (lambda (n d)
495.     (if (= d 0)
```

```
496.         1
497.         (if (>= n 0)
498.             (if (> d 0)
499.                 (math/2 n d)
500.                 (- (math/2 n (- d))))
501.             (if (> d 0)
502.                 (- (math/2 (- n) d))
503.                 (math/2 (- n) (- d)))))))
504. (define remainder (lambda (n d) (- n (* d (quotient n d)))))
505. (define modulo
506.   (lambda (n d)
507.     (if (= d 0)
508.         n
509.         (if (> d 0)
510.             (if (>= n 0)
511.                 (remainder n d)
512.                 (remainder (+ (remainder n d) d) d))
513.             (- (modulo (- n) (- d)))))))
514. (define gcd (lambda l (foldl mathgcd2 0 l)))
515. (define lcm (lambda l (foldl mathlcm2 1 l)))
516. (define numerator   (lambda (q) q))
517. (define denominator (lambda (q) 1))
518. (define floor numerator)
519. (define ceiling numerator)
520. (define truncate numerator)
521. (define round numerator)
522. (define rationalize (lambda (x y) x))
523. (define sqrt
524.   (lambda (x)
525.     (cond ((not (positive? x))
526.            0)
527.           ((= x 1)
528.            1)
529.           (else
530.            (let loop ((sous 1) (sur x))
531.              (if (<= (- sur sous) 1)
532.                  sous
533.                  (let* ((new (/ (+ sous sur) 2)))
534.                    (if (<= (* new new) x)
535.                        (loop new sur)
536.                        (loop sous new)))))))))
537. (define expt
538.   (lambda (base exp)
539.     (if (not (positive? exp))
540.         1
541.         (let* ((facteur (if (odd? exp) base 1))
542.                (reste (expt (* base base) (/ exp 2))))
543.           (* facteur reste)))))
544. (define exact->inexact (lambda (z) z))
545. (define inexact->exact (lambda (z) z))
546. (define number->string
547.   (lambda (n . lradix)
548.     (let* ((radix (if (null? lradix) 10 (car lradix)))
549.            (negative (negative? n))
550.            (absn (abs n)))
551.       (if (= n 0)
552.           (string-copy "0")
553.           (letrec ((decomp (lambda (n digits)
554.                              (if (= n 0)
555.                                  digits
556.                                  (decomp (/ n radix)
557.                                          (cons (modulo n radix) digits))))))
558.             (let* ((nd->ad (lambda (n)
559.                              (if (< n 10)
560.                                  (+ n (char->integer #\0))
561.                                  (+ (- n 10) (char->integer #\a)))))
562.                    (digits (decomp absn '()))
563.                    (adigits (map nd->ad digits))
564.                    (cdigits (map integer->char adigits))
565.                    (signedchars (if negative
566.                                     (cons #\- cdigits)
567.                                     cdigits)))
568.               (list->string signedchars)))))))
569. (define string->number
570.   (lambda (str . lradix)
571.     (let* ((radix (if (null? lradix) 10 (car lradix)))
572.            (maxnum (if (<= radix 10)
573.                        (integer->char (+ (- radix 1) (char->integer #\0)))
574.                        #\9))
575.            (len (string-length str)))
576.       (letrec ((checkdigit
577.                 (lambda (d)
578.                   (if (<= radix 10)
579.                       (and (char<=? #\0 d) (char<=? d maxnum))
580.                       (or (and (char<=? #\0 d) (char<=? d maxnum))
581.                           (and (char<=? #\a (char-downcase d))
582.                                (char<=? (char-downcase d) #\f))))))
583.                (checksyntax
584.                 (lambda (min pos)
585.                   (if (>= pos len)
586.                       (>= pos min)
587.                       (let ((d (string-ref str pos)))
588.                         (cond ((checkdigit d)
589.                                (checksyntax min (+ pos 1)))
590.                               ((or (char=? d #\+) (char=? d #\-))
591.                                (and (= pos 0) (checksyntax 2 1)))
592.                               (else #f))))))
593.                (recomp (lambda (acc digits)
594.                          (if (null? digits)
595.                              acc
596.                              (recomp (+ (* acc radix) (car digits))
597.                                      (cdr digits)))))
598.                (cd->nd (lambda (c)
599.                          (if (char-numeric? c)
600.                              (- (char->integer c) (char->integer #\0))
601.                              (+ (- (char->integer (char-downcase c))
602.                                    (char->integer #\a))
603.                                 10)))))
604.         (and (checksyntax 1 0)
605.              (let* ((signedchars (string->list str))
606.                     (negative (char=? (car signedchars) #\-))
607.                     (positive (char=? (car signedchars) #\+))
608.                     (cdigits (if (or negative positive)
609.                                  (cdr signedchars)
610.                                  signedchars))
611.                     (digits (map cd->nd cdigits))
612.                     (absn (recomp 0 digits)))
613.                (if negative (- absn) absn)))))))
614.
615. ; 6.6
616. char?
617. (define char=? (lambda (c1 c2) (= (char->integer c1) (char->integer c2))))
618. (define char<? (lambda (c1 c2) (not (char<=? c2 c1))))
619. (define char>? (lambda (c1 c2) (not (char<=? c1 c2))))
620. (define char<=? (lambda (c1 c2) (<= (char->integer c1) (char->integer c2))))
621. (define char>=? (lambda (c1 c2) (char<=? c2 c1)))
622. (define char-ci=?
623.   (lambda (c1 c2) (char=? (char-downcase c1) (char-downcase c2))))
```

```
624. (define char-ci<?
625.   (lambda (c1 c2) (char<? (char-downcase c1) (char-downcase c2))))
626. (define char-ci>?
627.   (lambda (c1 c2) (char>? (char-downcase c1) (char-downcase c2))))
628. (define char-ci<=?
629.   (lambda (c1 c2) (char<=? (char-downcase c1) (char-downcase c2))))
630. (define char-ci>=?
631.   (lambda (c1 c2) (char>=? (char-downcase c1) (char-downcase c2))))
632. (define char-alphabetic?
633.   (lambda (c) (and (char-ci<=? #\a c) (char-ci<=? c #\z))))
634. (define char-numeric? (lambda (c) (and (char<=? #\0 c) (char<=? c #\9))))
635. (define char-whitespace?
636.   (lambda (c)
637.     (or (char=? c #\space)
638.         (char=? c (integer->char 9))     ; Tab
639.         (char=? c #\newline)
640.         (char=? c (integer->char 12))    ; FF
641.         (char=? c (integer->char 13))))) ; CR
642. (define char-upper-case? (lambda (c) (and (char<=? #\A c) (char<=? c #\Z))))
643. (define char-lower-case? (lambda (c) (and (char<=? #\a c) (char<=? c #\z))))
644. char->integer
645. integer->char
646. (define char-upcase
647.   (lambda (c)
648.     (if (char-lower-case? c)
649.         (integer->char (+ (char->integer c)
650.                           (- (char->integer #\A) (char->integer #\a))))
651.         c)))
652. (define char-downcase
653.   (lambda (c)
654.     (if (char-upper-case? c)
655.         (integer->char (+ (char->integer c)
656.                           (- (char->integer #\a) (char->integer #\A))))
657.         c)))
658.
659. ; 6.7
660. string?
661. (define make-string
662.   (lambda (len . lfill)
663.     (let ((str (make-string1 len)))
664.       (if (not (null? lfill))
665.           (string-fill! str (car lfill)))
666.       str)))
667. (define string (lambda l (list->string l)))
668. string-length
669. string-ref
670. string-set!
671. string=?
672. (define string<?
673.   (lambda (s1 s2) (<  (string-compare char<? char=? s1 s2) 0)))
674. (define string>?
675.   (lambda (s1 s2) (>  (string-compare char<? char=? s1 s2) 0)))
676. (define string<=?
677.   (lambda (s1 s2) (<= (string-compare char<? char=? s1 s2) 0)))
678. (define string>=?
679.   (lambda (s1 s2) (>= (string-compare char<? char=? s1 s2) 0)))
680. (define string-ci=?
681.   (lambda (s1 s2) (=  (string-compare char-ci<? char-ci=? s1 s2) 0)))
682. (define string-ci<?
683.   (lambda (s1 s2) (<  (string-compare char-ci<? char-ci=? s1 s2) 0)))
684. (define string-ci>?
685.   (lambda (s1 s2) (>  (string-compare char-ci<? char-ci=? s1 s2) 0)))
686. (define string-ci<=?
687.   (lambda (s1 s2) (<= (string-compare char-ci<? char-ci=? s1 s2) 0)))
688. (define string-ci>=?
689.   (lambda (s1 s2) (>= (string-compare char-ci<? char-ci=? s1 s2) 0)))
690. (define substring
691.   (lambda (str start end)
692.     (let* ((len (- end start))
693.            (newstr (make-string len)))
694.       (let loop ((pos 0))
695.         (if (< pos len)
696.             (begin
697.               (string-set! newstr pos (string-ref str (+ start pos)))
698.               (loop (+ pos 1)))))
699.       newstr)))
700. (define string-append
701.   (lambda ls
702.     (let* ((llen (map string-length ls))
703.            (totlen (foldl + 0 llen))
704.            (newstring (make-string totlen))
705.            (iter (lambda (iter ls llen from to)
706.                    (if (< to totlen)
707.                        (if (< from (car llen))
708.                            (begin
709.                              (string-set! newstring
710.                                           to
711.                                           (string-ref (car ls) from))
712.                              (iter iter ls llen (+ from 1) (+ to 1)))
713.                            (iter iter (cdr ls) (cdr llen) 0 to))))))
714.       (iter iter ls llen 0 0)
715.       newstring)))
716. (define string->list
717.   (lambda (str)
718.     (let loop ((pos (- (string-length str) 1)) (l '()))
719.       (if (< pos 0)
720.           l
721.           (loop (- pos 1) (cons (string-ref str pos) l))))))
722. (define list->string
723.   (lambda (l)
724.     (let* ((len (length l))
725.            (newstring (make-string1 len))
726.            (iter (lambda (iter l to)
727.                    (if (< to len)
728.                        (begin
729.                          (string-set! newstring to (car l))
730.                          (iter iter (cdr l) (+ to 1)))))))
731.       (iter iter l 0)
732.       newstring)))
733. string-copy
734. (define string-fill!
735.   (lambda (str fill)
736.     (let loop ((pos (- (string-length str) 1)))
737.       (if (>= pos 0)
738.           (begin
739.             (string-set! str pos fill)
740.             (loop (- pos 1))))))
741.
742. ; 6.8
743. vector?
744. (define make-vector
745.   (lambda (len . lfill)
746.     (let ((v (make-vector1 len)))
747.       (if (not (null? lfill))
748.           (vector-fill! v (car lfill)))
749.       v)))
750. (define vector (lambda l (list->vector l)))
751. vector-length
```

```
752. vector-ref
753. vector-set!
754. (define vector->list
755.   (lambda (v)
756.     (let loop ((pos (- (vector-length v) 1)) (l '()))
757.       (if (< pos 0)
758.           l
759.           (loop (- pos 1) (cons (vector-ref v pos) l))))))
760. (define list->vector
761.   (lambda (l)
762.     (let* ((len (length l))
763.            (v (make-vector len)))
764.       (let loop ((l l) (pos 0))
765.         (if (not (null? l))
766.             (begin
767.               (vector-set! v pos (car l))
768.               (loop (cdr l) (+ pos 1)))))
769.       v)))
770. (define vector-fill!
771.   (lambda (v fill)
772.     (let loop ((pos (- (vector-length v) 1)))
773.       (if (>= pos 0)
774.           (begin
775.             (vector-set! v pos fill)
776.             (loop (- pos 1))))))
777.
778. ; 6.9
779. procedure?
780. (define apply
781.   (lambda (proc . llargs)
782.     (let ((largs (if (null? (cdr llargs))
783.                      (car llargs)
784.                      (foldr1 cons llargs))))
785.       (apply1 proc largs))))
786. (define map
787.   (lambda (proc . ll)
788.     (if (null? (car ll))
789.         '()
790.         (let ((tetes (map1 car ll))
791.               (queues (map1 cdr ll)))
792.           (cons (apply proc tetes)
793.                 (apply map (cons proc queues)))))))
794. (define for-each
795.   (lambda (proc . ll)
796.     (if (null? (car ll))
797.         #f
798.         (let* ((tetes (map car ll))
799.                (queues (map cdr ll)))
800.           (apply proc tetes)
801.           (apply for-each (cons proc queues))))))
802. (define force (lambda (promise) (promise)))
803. (define call-with-current-continuation
804.   (lambda (proc)
805.     (let ((cc (return-current-continuation)))
806.       (if (vector? cc)
807.           (vector-ref cc 0)
808.           (let ((escape-proc (lambda (val)
809.                                (let ((v (vector val)))
810.                                  (return-there-with-this cc v)))))
811.             (proc escape-proc))))))
812. (define call/cc call-with-current-continuation)
813.
814. ; 6.10
815. read-char
816. peek-char
817. (define write
818.   (lambda (d)
819.     (cond ((eq? d #f)
820.            (write-many-chars #\# #\f))
821.           ((eq? d #t)
822.            (write-many-chars #\# #\t))
823.           ((symbol? d)
824.            (apply write-many-chars (string->list (symbol->string d))))
825.           ((eqv? d #\space)
826.            (write-many-chars #\# #\\ #\s #\p #\a #\c #\e))
827.           ((eqv? d #\newline)
828.            (write-many-chars #\# #\\ #\n #\e #\w #\l #\i #\n #\e))
829.           ((char? d)
830.            (write-many-chars #\# #\\ d))
831.           ((vector? d)
832.            (write-vector d write))
833.           ((pair? d)
834.            (write-char #\()
835.            (write (car d))
836.            (write-cdr (cdr d) write))
837.           ((number? d)
838.            (apply write-many-chars (string->list (number->string d))))
839.           ((string? d)
840.            (write-char #\")
841.            (let ((len (string-length d)))
842.              (let loop ((pos 0))
843.                (if (< pos len)
844.                    (let ((c (string-ref d pos)))
845.                      (cond ((char=? c #\")
846.                             (write-many-chars #\\ #\") (loop (+ pos 1)))
847.                            ((char=? c #\\)
848.                             (write-many-chars #\\ #\\) (loop (+ pos 1)))
849.                            (else
850.                             (write-char c)))))))
851.            (write-char #\"))
852.           ((procedure? d)
853.            (write-many-chars #\# #\< #\p #\r #\o #\c #\e #\d #\u #\r #\e #\>))
854.           ((null? d)
855.            (write-many-chars #\( #\)))
856.           (else
857.            #f))))
858. (define display
859.   (lambda (d)
860.     (cond ((char? d)
861.            (write-char d))
862.           ((vector? d)
863.            (write-vector d display))
864.           ((pair? d)
865.            (write-char #\()
866.            (display (car d))
867.            (write-cdr (cdr d) display))
868.           ((string? d)
869.            (apply write-many-chars (string->list d)))
870.           (else
871.            (write d)))))
872. (define newline (lambda () (write-char #\newline)))
873. write-char
874.
875. #f ; Fin des definitions des fonctions standard
```

A.3 Le compilateur vers du code-octets

Les différentes phases de la compilation sont, autant que possible, implantées dans le même ordre qu'elles sont exécutées.

```
 1. ; Les fonctions utilitaires generales
 2.
 3. ; Suppose que les deux arguments sont deja des ensembles de symboles
 4. (define symbol-set-union
 5.   (lambda (ss1 ss2)
 6.     (cond ((null? ss1)
 7.            ss2)
 8.           ((memq (car ss1) ss2)
 9.            (symbol-set-union (cdr ss1) ss2))
10.           (else
11.            (cons (car ss1) (symbol-set-union (cdr ss1) ss2))))))
12.
13. (define symbol-set-intersection
14.   (lambda (ss1 ss2)
15.     (cond ((null? ss1)
16.            '())
17.           ((memq (car ss1) ss2)
18.            (cons (car ss1) (symbol-set-intersection (cdr ss1) ss2)))
19.           (else
20.            (symbol-set-intersection (cdr ss1) ss2)))))
21.
22. (define foldr
23.   (lambda (binop start l)
24.     (if (null? l)
25.         start
26.         (binop (car l) (foldr binop start (cdr l))))))
27.
28. (define foldr1
29.   (lambda (binop l)
30.     (if (null? (cdr l))
31.         (car l)
32.         (binop (car l) (foldr1 binop (cdr l))))))
33.
34. (define filter
35.   (lambda (pred? l)
36.     (cond ((null? l) '())
37.           ((pred? (car l)) (cons (car l) (filter pred? (cdr l))))
38.           (else (filter pred? (cdr l))))))
39.
40. (define formals->varlist
41.   (lambda (formals)
42.     (cond ((symbol? formals)
43.            (list formals))
44.           ((null? formals)
45.            '())
46.           (else
47.            (cons (car formals) (formals->varlist (cdr formals)))))))
48.
49. (define prefix?
50.   (lambda (s1 s2)
51.     (let ((l1 (string-length s1))
52.           (l2 (string-length s2)))
53.       (if (> l1 l2)
54.           #f
55.           (let loop ((i 0))
56.             (cond ((= i l1)
57.                    #t)
58.                   ((char=? (string-ref s1 i) (string-ref s2 i))
59.                    (loop (+ i 1)))
60.                   (else
61.                    #f)))))))
62.
63. (define unprefix
64.   (lambda (s1 s2)
65.     (let ((l1 (string-length s1)))
66.       (cond ((= l1 (string-length s2))
67.              (string-append s1 "a"))
68.             ((char=? #\a (string-ref s2 l1))
69.              (string-append s1 "b"))
70.             (else
71.              (string-append s1 "a"))))))
72.
73.
74.
75.
76. ; Initialiser les variables globales du programme
77.
78. (define init-glob-vars
79.   (lambda ()
80.     (set! safe-name-memv                '<memv>)
81.     (set! safe-name-make-promise '<make-promise>)
82.     (set! safe-name-list->vector '<list->vector>)
83.     (set! safe-name-list                '<list>)
84.     (set! safe-name-append2           '<append2>)
85.     (set! safe-name-cons                '<cons>)
86.     (set! gen-sym-pref   #f)
87.     (set! gen-sym-number  0)
88.     (set! libcprims      #f)
89.     (set! libalias       #f)
90.     (set! libclos        #f)
91.     (set! libpublics     #f)
92.     (set! libnames       #f)
93.     (set! apply1-cprim-no #f)
94.     (set! dirreq        #f)
95.     (set! allreq        #f)
96.     (set! req-clos-nodes #f)
97.     (set! const-counter 0)
98.     (set! const-alist '())
99.     (set! top-counter  0)
100.    (set! top-alist   '())
101.    (set! const-desc-string #f)
102.    (set! glob-counter  0)
103.    (set! glob-hidden '())
104.    (set! glob-public '())
105.    (set! glob-source '())
106.    (set! glob-v     '#())
107.    (set! glob-v-len   0)
108.    (set! phys-glob-no 0)
109.    (set! program-bytecode #f)
110.    (set! label-counter 0)
111.    (set! label-v    #())
112.    (set! label-v-len   0)
113.    (set! flat-program-bytecode #f)
114.    (set! final-program-bytecode #f)
```

```
115.     (set! glob-var-init-codes #f)))
116.
117.
118.
119.
120. ; Lire le programme source
121.
122. (define read-source
123.   (lambda (name)
124.     (let ((port (open-input-file name)))
125.       (let loop ()
126.         (let ((exp (read port)))
127.           (if (eof-object? exp)
128.               (begin
129.                 (close-input-port port)
130.                 '())
131.               (cons exp (loop))))))))
132.
133.
134.
135.
136. ; Trouver des noms pour memv, make-promise, list->vector, list,
137. ; append2 et cons. De cette facon, trans-case & cie plus loin
138. ; pourront inserer un symbole representant une fonction
139. ; du systeme aux endroits generes par les macros-expansion
140.
141. (define safe-name-memv          '<memv>)
142. (define safe-name-make-promise '<make-promise>)
143. (define safe-name-list->vector '<list->vector>)
144. (define safe-name-list         '<list>)
145. (define safe-name-append2      '<append2>)
146. (define safe-name-cons         '<cons>)
147.
148. ; Cette fonction ne fonctionne que pour les struc. non-circ.
149. (define find-all-symbols
150.   (lambda (d)
151.     (cond ((symbol? d)
152.            (list d))
153.           ((pair? d)
154.            (symbol-set-union (find-all-symbols (car d))
155.                              (find-all-symbols (cdr d))))
156.           ((vector? d)
157.            (let loop ((pos (- (vector-length d) 1)))
158.              (if (< pos 0)
159.                  '()
160.                  (symbol-set-union (find-all-symbols (vector-ref d pos))
161.                                    (loop (- pos 1))))))
162.           (else
163.            '())))
164.
165. ; Trouver un prefixe unique pour avoir un gen-sym correct
166. (define find-uniq-prefix
167.   (lambda (ss)
168.     (let loop ((pref "") (names (map symbol->string ss)))
169.       (if (null? names)
170.           pref
171.           (let ((name (car names)))
172.             (if (prefix? pref name)
173.                 (loop (unprefix pref name) (cdr names))
174.                 (loop pref (cdr names)))))))
175.
176. ; La fonction gen-sym
177. (define gen-sym-pref #f)
178. (define gen-sym-number 0)
179.
180. (define gen-sym
181.   (lambda ()
182.     (let* ((str-num (number->string gen-sym-number))
183.            (sym-name (string-append gen-sym-pref str-num))
184.            (sym (string->symbol sym-name)))
185.       (set! gen-sym-number (+ gen-sym-number 1))
186.       sym)))
187.
188.
189.
190.
191. ; Traduction des expressions Scheme en expressions plus simples
192. ; Il ne reste que des references de var., des quotes, des literaux
193. ; s'evaluant a eux-memes, des appels de procedure, des lambda exp.,
194. ; des if, des set!, des begins, des defines se rapportant a des
195. ; var. glob. uniquement
196.
197. ; A ne pas inserer dans la liste des translators
198. (define trans-define
199.   (lambda (l)
200.     (if (symbol? (cadr l))
201.         l
202.         (let ((procname (caadr l))
203.               (formals (cdadr l))
204.               (exps (cddr l)))
205.           `(define ,procname (lambda ,formals ,@exps))))))
206.
207. ; A ne pas inserer dans la liste des translators
208. (define flatten-begin
209.   (lambda (expbegin)
210.     (cons 'begin
211.           (let loop ((l (cdr expbegin)) (flat '()))
212.             (if (null? l)
213.                 flat
214.                 (let ((tete (car l))
215.                       (queue (cdr l)))
216.                   (if (and (pair? tete) (eq? (car tete) 'begin))
217.                       (loop (cdr tete) (loop queue flat))
218.                       (cons tete (loop queue flat)))))))))
219.
220. ; A ne pas inserer dans la liste des translators
221. (define extract-define
222.   (lambda (lexp)
223.     (let ((tete (car lexp))
224.           (reste (cdr lexp)))
225.       (if (and (pair? tete) (eq? (car tete) 'define))
226.           (let ((result (extract-define reste)))
227.             (cons (cons tete (car result)) (cdr result)))
228.           (cons '() lexp)))))
229.
230. ; A ne pas inserer dans la liste des translators
231. (define trans-body
232.   (lambda (l)
233.     (let* ((flatbody (flatten-begin l))
234.            (result (extract-define (cdr flatbody)))
235.            (rawdefines (car result))
236.            (exps (cdr result))
237.            (defines (map trans-define rawdefines))
238.            (decls (map cdr defines))
239.            (body `(begin ,@exps)))
240.       (if (null? decls)
241.           body
242.           `(letrec ,decls ,body)))))
```

```
243.
244. ; A ne pas inserer dans la liste des translators
245. (define trans-lambda
246.   (lambda (l)
247.     (list 'lambda
248.           (cadr l)
249.           (trans-body (cons 'begin (cddr l))))))
250.
251. ; A ne pas inserer dans la liste des translators
252. (define trans-begin
253.   (lambda (l)
254.     (let ((flat (flatten-begin l)))
255.       (if (null? (cddr flat))
256.           (cadr flat)
257.           flat))))
258.
259. ; A ne pas inserer dans la liste des translators
260. (define trans-normal-let
261.   (lambda (l)
262.     (let* ((bindings (cadr l))
263.            (body (cddr l))
264.            (vars (map car bindings))
265.            (inits (map cadr bindings)))
266.       `((lambda ,vars ,@body) ,@inits))))
267.
268. ; A ne pas inserer dans la liste des translators
269. (define trans-let-loop
270.   (lambda (l)
271.     (let* ((loop-name (cadr l))
272.            (bindings (caddr l))
273.            (body (cdddr l))
274.            (vars (map car bindings))
275.            (inits (map cadr bindings)))
276.       `((letrec ((,loop-name (lambda ,vars ,@body)))
277.           ,loop-name)
278.         ,@inits))))
279.
280. (define trans-let
281.   (lambda (l)
282.     (if (symbol? (cadr l))
283.         (trans-let-loop l)
284.         (trans-normal-let l))))
285.
286. (define trans-let*
287.   (lambda (l)
288.     (let ((bindings (cadr l))
289.           (body (cddr l)))
290.       (if (or (null? bindings) (null? (cdr bindings)))
291.           `(let ,bindings ,@body)
292.           (let ((prem (car bindings))
293.                 (reste (cdr bindings)))
294.             `(let (,prem) (let* ,reste ,@body)))))))
295.
296. (define trans-letrec
297.   (lambda (l)
298.     (let ((bindings (cadr l))
299.           (body (cddr l)))
300.       (if (null? bindings)
301.           `(let () ,@body)
302.           (let* ((vars (map car bindings))
303.                  (inits (map cadr bindings))
304.                  (falsebind (map (lambda (v) `(,v #f)) vars))
305.                  (set!s (map (lambda (v i) `(set! ,v ,i)) vars inits)))
306.             `(let ,falsebind
307.                ,@set!s
308.                (let () ,@body)))))))
309.
310. (define trans-and
311.   (lambda (l)
312.     (cond ((null? (cdr l))
313.            #t)
314.           ((null? (cddr l))
315.            (cadr l))
316.           (else
317.            `(if ,(cadr l) (and ,@(cddr l)) #f)))))
318.
319. (define trans-or
320.   (lambda (l)
321.     (cond ((null? (cdr l))
322.            #f)
323.           ((null? (cddr l))
324.            (cadr l))
325.           (else
326.            (let* ((e-hd (cadr l))
327.                   (e-tl (cddr l))
328.                   (tmp (gen-sym)))
329.              `(let ((,tmp ,e-hd))
330.                 (if ,tmp
331.                     ,tmp
332.                     (or ,@e-tl))))))))
333.
334. (define trans-cond
335.   (lambda (l)
336.     (if (null? (cdr l))
337.         #f
338.         (let* ((clause (cadr l))
339.                (autres (cddr l))
340.                (newcond (cons 'cond autres)))
341.           (cond ((eq? (car clause) 'else)
342.                  (cons 'begin (cdr clause)))
343.                 ((null? (cdr clause))
344.                  (list 'or (car clause) newcond))
345.                 ((eq? (cadr clause) '=>)
346.                  (let* ((test (car clause))
347.                         (recipient (caddr clause))
348.                         (tmp (gen-sym)))
349.                    `(let ((,tmp ,test))
350.                       (if ,tmp
351.                           (,recipient ,tmp)
352.                           ,newcond))))
353.                 (else
354.                  (let* ((test (car clause))
355.                         (actions (cdr clause))
356.                         (conseq (cons 'begin actions)))
357.                    `(if ,test ,conseq ,newcond))))))))
358.
359. (define trans-case
360.   (lambda (l)
361.     (let* ((tmp-key (gen-sym))
362.            (trans-test
363.             (lambda (test)
364.               (if (eq? test 'else) 'else `(,safe-name-memv ,tmp-key ',test))))
365.            (key (cadr l))
366.            (clauses (cddr l))
367.            (tests (map car clauses))
368.            (expr-lists (map cdr clauses))
369.            (memv-tests (map trans-test tests))
370.            (cond-clauses (map cons memv-tests expr-lists)))
```

```
371.       `(let ((,tmp-key ,key)) (cond ,@cond-clauses)))))
372.
373. (define trans-do
374.   (lambda (l)
375.     (let* ((normalize-step (lambda (v sf)
376.                              (if (null? sf) v (car sf))))
377.            (bindings (cadr l))
378.            (testnsequence (caddr l))
379.            (commands (cdddr l))
380.            (vars (map car bindings))
381.            (inits (map cadr bindings))
382.            (steps-fac (map cddr bindings))
383.            (steps (map normalize-step vars steps-fac))
384.            (test (car testnsequence))
385.            (sequence (cdr testnsequence))
386.            (loop-var (gen-sym))
387.            (loop-call (cons loop-var steps))
388.            (loop-bindings (map list vars inits)))
389.       `(let ,loop-var ,loop-bindings
390.            (if ,test
391.                (begin
392.                  #f
393.                  ,@sequence)
394.                (begin
395.                  ,@commands
396.                  ,loop-call))))))
397.
398. (define trans-delay
399.   (lambda (exp)
400.     (let ((delexp (cadr exp)))
401.       `(,safe-name-make-promise (lambda () ,delexp)))))
402.
403. ; A ne pas inclure dans la liste des translators
404. (define detect-unquote
405.   (lambda (exp level)
406.     (cond ((vector? exp)
407.            (let loop ((pos (- (vector-length exp) 1)))
408.              (cond (((< pos 0)
409.                     #f)
410.                    ((detect-unquote (vector-ref exp pos) level)
411.                     #t)
412.                    (else
413.                     (loop (- pos 1))))))
414.           ((pair? exp)
415.            (let ((tete (car exp)))
416.              (cond ((eq? tete 'quasiquote)
417.                     (detect-unquote (cadr exp) (+ level 1)))
418.                    ((or (eq? tete 'unquote) (eq? tete 'unquote-splicing))
419.                     (if (= level 1)
420.                         #t
421.                         (detect-unquote (cadr exp) (- level 1))))
422.                    (else
423.                     (or (detect-unquote tete level)
424.                         (detect-unquote (cdr exp) level))))))
425.           (else
426.            #f))))
427.
428. (define trans-quasiquote
429.   (lambda (l)
430.     (let loop ((exp (cadr l)) (level 1))
431.       (cond ((not (detect-unquote exp level))
432.              (list 'quote exp))
433.             ((vector? exp)
434.              (list safe-name-list->vector
435.                    (loop (vector->list exp) level)))
436.             ((pair? exp)
437.              (let ((tete (car exp)))
438.                (cond ((eq? tete 'quasiquote)
439.                       (list safe-name-list
440.                             ''quasiquote
441.                             (loop (cadr exp) (+ level 1))))
442.                      ((eq? tete 'unquote)
443.                       (if (= level 1)
444.                           (cadr exp)
445.                           (list safe-name-list
446.                                 ''unquote
447.                                 (loop (cadr exp) (- level 1)))))
448.                      ((and (pair? tete)
449.                            (eq? (car tete) 'unquote-splicing)
450.                            (= level 1))
451.                       (if (null? (cdr exp))
452.                           (cadr tete)
453.                           (list safe-name-append2
454.                                 (cadr tete)
455.                                 (loop (cdr exp) level))))
456.                      ((eq? tete 'unquote-splicing)
457.                       (list safe-name-list
458.                             ''unquote-splicing
459.                             (loop (cadr exp) (- level 1))))
460.                      (else
461.                       (list safe-name-cons
462.                             (loop (car exp) level)
463.                             (loop (cdr exp) level))))))))))
464.
465. (define translators
466.   (list (cons 'let trans-let)
467.         (cons 'let* trans-let*)
468.         (cons 'letrec trans-letrec)
469.         (cons 'and trans-and)
470.         (cons 'or trans-or)
471.         (cons 'cond trans-cond)
472.         (cons 'case trans-case)
473.         (cons 'do trans-do)
474.         (cons 'delay trans-delay)
475.         (cons 'quasiquote trans-quasiquote)))
476.
477. (define trans-sub
478.   (lambda (exp)
479.     (if (or (boolean? exp)
480.             (symbol? exp)
481.             (char? exp)
482.             (number? exp)
483.             (string? exp))
484.         exp
485.         (let ((tete (car exp)))
486.           (cond ((eq? tete 'quote)
487.                  exp)
488.                 ((eq? tete 'lambda)
489.                  (let ((new-lambda (trans-lambda exp)))
490.                    (list 'lambda
491.                          (cadr new-lambda)
492.                          (trans-sub (caddr new-lambda)))))
493.                 ((eq? tete 'if)
494.                  (cons 'if (map trans-sub (cdr exp))))
495.                 ((eq? tete 'set!)
496.                  (list 'set! (cadr exp) (trans-sub (caddr exp))))
497.                 ((eq? tete 'begin)
498.                  (trans-begin (cons 'begin (map trans-sub (cdr exp)))))
```

```
499.                ((eq? tete 'define)
500.                 (let ((new-define (trans-define exp)))
501.                   (list 'define
502.                         (cadr new-define)
503.                         (trans-sub (caddr new-define)))))
504.                (else
505.                 (let ((ass (assq tete translators)))
506.                   (if ass
507.                       (trans-sub ((cdr ass) exp))
508.                       (let ((new-exp (map trans-sub exp)))
509.                         (if (and (pair? (car new-exp))
510.                                  (eq? (caar new-exp) 'lambda)
511.                                  (null? (cadar new-exp)))
512.                             (caddar new-exp)
513.                             new-exp)))))))))
514.
515.
516.
517.
518. ; Operations sur les nodes
519.
520. (define make-cte-node
521.   (lambda (cte)              (vector 0 cte #f)))
522. (define make-ref-node
523.   (lambda (symbol)           (vector 1 symbol #f #f #f)))
524. (define make-ref-node-full
525.   (lambda (symbol loc val glob?) (vector 1 symbol loc val glob?)))
526. (define make-lambda-node
527.   (lambda (formals body)     (vector 2 formals #f body #f)))
528. (define make-if-node
529.   (lambda (test conseq altern)  (vector 3 test conseq altern)))
530. (define make-set!-node
531.   (lambda (symbol exp)        (vector 4 symbol #f exp #f)))
532. (define make-begin-node
533.   (lambda (lexp)             (vector 5 lexp)))
534. (define make-def-node
535.   (lambda (symbol exp)        (vector 6 symbol #f exp)))
536. (define make-call-node
537.   (lambda (op larg)           (vector 7 op larg)))
538. (define make-globdesc-node
539.   (lambda (symbol lib? nbaff)  (vector 8 symbol lib? nbaff #f #f #f)))
540.
541. (define node-type
542.   (lambda (node)
543.     (vector-ref node 0)))
544.
545. (define cte-node?      (lambda (node) (= (node-type node) 0)))
546. (define ref-node?      (lambda (node) (= (node-type node) 1)))
547. (define lambda-node?   (lambda (node) (= (node-type node) 2)))
548. (define if-node?       (lambda (node) (= (node-type node) 3)))
549. (define set!-node?     (lambda (node) (= (node-type node) 4)))
550. (define begin-node?    (lambda (node) (= (node-type node) 5)))
551. (define def-node?      (lambda (node) (= (node-type node) 6)))
552. (define call-node?     (lambda (node) (= (node-type node) 7)))
553. (define globdesc-node? (lambda (node) (= (node-type node) 8)))
554.
555. (define getter1 (lambda (node) (vector-ref node 1)))
556. (define getter2 (lambda (node) (vector-ref node 2)))
557. (define getter3 (lambda (node) (vector-ref node 3)))
558. (define getter4 (lambda (node) (vector-ref node 4)))
559. (define getter5 (lambda (node) (vector-ref node 5)))
560. (define getter6 (lambda (node) (vector-ref node 6)))
561.
562. (define get-cte     getter1)
563. (define get-no      getter2)
564. (define get-symbol  getter1)
565. (define get-loc     getter2)
566. (define get-val     getter3)
567. (define get-glob?   getter4)
568. (define get-formals getter1)
569. (define get-fdesc   getter2)
570. (define get-body    getter3)
571. (define get-label   getter4)
572. (define get-test    getter1)
573. (define get-conseq  getter2)
574. (define get-altern  getter3)
575. (define get-exp     getter3)
576. (define get-lexp    getter1)
577. (define get-op      getter1)
578. (define get-larg    getter2)
579. (define get-lib?    getter2)
580. (define get-nbaff   getter3)
581. (define get-init    getter4)
582. (define get-libno   getter5)
583. (define get-srcno   getter6)
584.
585. (define setter1 (lambda (node val) (vector-set! node 1 val)))
586. (define setter2 (lambda (node val) (vector-set! node 2 val)))
587. (define setter3 (lambda (node val) (vector-set! node 3 val)))
588. (define setter4 (lambda (node val) (vector-set! node 4 val)))
589. (define setter5 (lambda (node val) (vector-set! node 5 val)))
590. (define setter6 (lambda (node val) (vector-set! node 6 val)))
591.
592. (define set-no!    setter2)
593. (define set-glob?! setter4)
594. (define set-loc!   setter2)
595. (define set-fdesc! setter2)
596. (define set-nbaff! setter3)
597. (define set-init!  setter4)
598. (define set-op!    setter1)
599. (define set-val!   setter3)
600. (define set-libno! setter5)
601. (define set-srcno! setter6)
602. (define set-label! setter4)
603.
604.
605.
606.
607. ; Transformation du code source en noeuds fonctionnels.
608.
609. (define lnode #f)
610.
611. (define exp->node
612.   (lambda (exp)
613.     (cond ((or (boolean? exp) (char? exp) (number? exp) (string? exp))
614.            (make-cte-node exp))
615.           ((symbol? exp)
616.            (make-ref-node exp))
617.           (else  ; pair
618.            (let ((tete (car exp)))
619.              (cond ((eq? tete 'quote)
620.                     (make-cte-node (cadr exp)))
621.                    ((eq? tete 'lambda)
622.                     (make-lambda-node (cadr exp) (exp->node (caddr exp))))
623.                    ((eq? tete 'if)
624.                     (make-if-node (exp->node (cadr exp))
625.                                   (exp->node (caddr exp))
626.                                   (exp->node
```

```
627.                                    (if (null? (cdddr exp)) #f (cadddr exp)))))
628.                    ((eq? tete 'set!)
629.                     (make-set!-node (cadr exp) (exp->node (caddr exp))))
630.                    ((eq? tete 'begin)
631.                     (make-begin-node (map exp->node (cdr exp))))
632.                    ((eq? tete 'define)
633.                     (make-def-node (cadr exp) (exp->node (caddr exp))))
634.                    (else  ; procedure call
635.                     (make-call-node (exp->node (car exp))
636.                                     (map exp->node (cdr exp))))))))))
637.
638.
639.
640.
641. ; Ramassage des variables globales definies, modifiees ou lues
642.
643. (define extract-glob-names
644.   (let ((action-v
645.          (vector
646.           (lambda (node loop env)  ; cte
647.             '())
648.           (lambda (node loop env)  ; ref
649.             (let ((refsym (get-symbol node)))
650.               (if (memq refsym env)
651.                   '()
652.                   (list refsym))))
653.           (lambda (node loop env)  ; lambda
654.             (loop (get-body node)
655.                   (symbol-set-union env (formals->varlist
656.                                          (get-formals node)))))
657.           (lambda (node loop env)  ; if
658.             (let ((test-globs (loop (get-test node) env))
659.                   (conseq-globs (loop (get-conseq node) env))
660.                   (altern-globs (loop (get-altern node) env)))
661.               (symbol-set-union (symbol-set-union test-globs conseq-globs)
662.                                 altern-globs)))
663.           (lambda (node loop env)  ; set!
664.             (let* ((set!sym (get-symbol node))
665.                    (l (if (memq set!sym env) '() (list set!sym))))
666.               (symbol-set-union l (loop (get-exp node) env))))
667.           (lambda (node loop env)  ; begin
668.             (let* ((lnode (get-lexp node))
669.                    (llglob (map (lambda (node) (loop node env)) lnode)))
670.               (foldr1 symbol-set-union llglob)))
671.           (lambda (node loop env)  ; def
672.             (symbol-set-union (list (get-symbol node))
673.                               (loop (get-exp node) env)))
674.           (lambda (node loop env)  ; call
675.             (let* ((lnode (cons (get-op node) (get-larg node)))
676.                    (llglob (map (lambda (node) (loop node env)) lnode)))
677.               (foldr1 symbol-set-union llglob))))))
678.     (lambda (node)
679.       (let loop ((node node) (env '()))
680.         ((vector-ref action-v (node-type node)) node loop env)))))
681.
682.
683.
684.
685. ; Chargement de la librairie
686.
687. (define libcprims #f)
688. (define libalias #f)
689. (define libclos #f)
690. (define libpublics #f)
691. (define libnames #f)
692. (define apply1-cprim-no #f)
693.
694. (define read-lib
695.   (lambda (libname)
696.     (let ((port (open-input-file libname)))
697.       (let loop1 ((n 0))
698.         (if (= n 4)
699.             (begin
700.               (close-input-port port)
701.               '())
702.             (let loop2 ()
703.               (let* ((datum (read port)))
704.                 (if datum
705.                     (let ((reste (loop2)))
706.                       (cons (cons datum (car reste)) (cdr reste)))
707.                     (cons '() (loop1 (+ n 1))))))))))
708.
709. (define get-lib-cprims
710.   (lambda (libpart1)
711.     (map (lambda (def)
712.            (let ((name (car def))
713.                  (no (cdr def)))
714.              (if (eq? name 'apply1)
715.                  (set! apply1-cprim-no no))
716.              (cons name no)))
717.          libpart1)))
718.
719. (define get-lib-alias
720.   (lambda (libpart2 libpart3 libpart4)
721.     (let* ((defs (append libpart2 libpart3 libpart4))
722.            (defals (filter (lambda (def)
723.                              (and (not (symbol? def))
724.                                   (symbol? (caddr def))))
725.                            defs)))
726.       (map (lambda (defal) (cons (cadr defal) (caddr defal))) defals))))
727.
728. (define get-lib-clos
729.   (lambda (libpart2 libpart3 libpart4)
730.     (let* ((defs (append libpart2 libpart3 libpart4))
731.            (defcls (filter (lambda (def)
732.                              (and (not (symbol? def))
733.                                   (not (symbol? (caddr def)))))
734.                            defs)))
735.       (map (lambda (defcl) (cons (cadr defcl) (caddr defcl))) defcls))))
736.
737. (define get-lib-publics
738.   (lambda (libpart3 libpart4)
739.     (map (lambda (sym-or-def)
740.            (if (symbol? sym-or-def)
741.                sym-or-def
742.                (cadr sym-or-def)))
743.          (append libpart3 libpart4))))
744.
745. (define get-lib-names
746.   (lambda (libpart1 libpart2 libpart3 libpart4)
747.     (let* ((cprim-names (map car libpart1))
748.            (defs (append libpart2 libpart3 libpart4))
749.            (truedefs (filter (lambda (def) (not (symbol? def))) defs)))
750.       (append cprim-names (map cadr truedefs)))))
751.
752. (define load-lib
753.   (lambda ()
754.     (let* ((alllib (read-lib "librairie.scm"))
```

```
755.            (libpart1 (list-ref alllib 0))
756.            (libpart2 (list-ref alllib 1))
757.            (libpart3 (list-ref alllib 2))
758.            (libpart4 (list-ref alllib 3)))
759.       (set! libcprims (get-lib-cprims libpart1))
760.       (set! libalias (get-lib-alias libpart2 libpart3 libpart4))
761.       (set! libclos (get-lib-clos libpart2 libpart3 libpart4))
762.       (set! libpublics (get-lib-publics libpart3 libpart4))
763.       (set! libnames (get-lib-names libpart1 libpart2 libpart3 libpart4)))))
764.
765.
766.
767.
768. ; Capture de la librairie necessaire
769.
770. (define dirreq #f)
771. (define allreq #f)
772. (define req-clos-nodes #f)
773.
774. (define grab-lib
775.   (lambda (dirreq)
776.     (let loop ((toadd dirreq) (added '()) (clos-nodes '()))
777.       (cond ((null? toadd)
778.              (set! allreq added)
779.              (set! req-clos-nodes clos-nodes))
780.             ((memq (car toadd) added)
781.              (loop (cdr toadd) added clos-nodes))
782.             (else
783.              (let* ((newfun (car toadd))
784.                     (ass-cprim (assq newfun libcprims)))
785.                (if ass-cprim
786.                    (loop (cdr toadd)
787.                          (cons newfun added)
788.                          clos-nodes)
789.                    (let ((ass-alias (assq newfun libalias)))
790.                      (if ass-alias
791.                          (loop (cons (cdr ass-alias) (cdr toadd))
792.                                (cons newfun added)
793.                                clos-nodes)
794.                          (let ((ass-clos (assq newfun libclos)))
795.                            (let* ((code (cdr ass-clos))
796.                                   (sub-code (trans-sub code))
797.                                   (node (exp->node sub-code))
798.                                   (node-globs (extract-glob-names node))
799.                                   (new-clos (cons newfun node)))
800.                              (loop (append node-globs (cdr toadd))
801.                                    (cons newfun added)
802.                                    (cons new-clos clos-nodes))))))))))
803.
804.
805.
806.
807. ; Ramassage et codage des constantes du programme source
808.
809. (define const-counter 0)
810. (define const-alist '())
811. ; Chaque ass: (original numero . desc)
812. (define top-counter 0)
813. (define top-alist '())
814. ; Chaque ass: (const-no . top-const-no)
815. (define const-desc-string #f)
816.
817. (define const-no
818.   (lambda (d)
819.     (let ((ass (assoc d const-alist)))
820.       (if ass
821.           (cadr ass)
822.           (begin
823.             (cond ((or (null? d) (boolean? d) (char? d) (number? d))
824.                    (set! const-alist
825.                          (cons (cons d (cons const-counter d)) const-alist)))
826.                   ((pair? d)
827.                    (let* ((leftno (const-no (car d)))
828.                           (rightno (const-no (cdr d)))
829.                           (desc (cons leftno rightno)))
830.                      (set! const-alist (cons (cons d (cons const-counter desc))
831.                                              const-alist))))
832.                   ((string? d)
833.                    (set! const-alist
834.                          (cons (cons d (cons const-counter d)) const-alist)))
835.                   ((symbol? d)
836.                    (let* ((nom (symbol->string d))
837.                           (nomno (const-no nom))
838.                           (desc (string->symbol (number->string nomno))))
839.                      (set! const-alist (cons (cons d (cons const-counter desc))
840.                                              const-alist))))
841.                   ((vector? d)
842.                    (let* ((listd (vector->list d))
843.                           (listno (map const-no listd))
844.                           (desc (list->vector listno)))
845.                      (set! const-alist (cons (cons d (cons const-counter desc))
846.                                              const-alist)))))
847.             (set! const-counter (+ const-counter 1))
848.             (- const-counter 1))))))
849.
850. (define top-const-no
851.   (lambda (d)
852.     (if (or (null? d) (boolean? d) (char? d) (number? d))
853.         #f
854.         (let* ((cno (const-no d))
855.                (ass (assv cno top-alist)))
856.           (if ass
857.               (cdr ass)
858.               (begin
859.                 (set! top-alist (cons (cons cno top-counter) top-alist))
860.                 (set! top-counter (+ top-counter 1))
861.                 (- top-counter 1)))))))
862.
863. (define code-abs-number
864.   (lambda (n)
865.     (let* ((msb (quotient n 256))
866.            (lsb (modulo n 256))
867.            (msc (integer->char msb))
868.            (lsc (integer->char lsb)))
869.       (string msc lsc))))
870.
871. (define code-one-const
872.   (lambda (desc)
873.     (cond ((null? desc)
874.            "0")                          ; 0 pour EMPTY
875.           ((pair? desc)
876.            (string-append "1"            ; 1 pour PAIR
877.                           (code-abs-number (car desc))
878.                           (code-abs-number (cdr desc))))
879.           ((boolean? desc)
880.            (if desc "2t" "2f"))          ; 2 pour BOOLEAN
881.           ((char? desc)
882.            (string #\3 desc))            ; 3 pour CHAR
```

```
883.          ((string? desc)
884.           (string-append "4"                ; 4 pour STRING
885.                          (code-abs-number (string-length desc))
886.                          desc))
887.          ((symbol? desc)
888.           (string-append "5"                ; 5 pour SYMBOL
889.                          (code-abs-number
890.                           (string->number (symbol->string desc)))))
891.          ((number? desc)
892.           (string-append "6"                ; 6 pour NUMBER
893.                          (if (< desc 0) "-" "+")
894.                          (code-abs-number (abs desc))))
895.          ((vector? desc)
896.           (let* ((listref (vector->list desc))
897.                  (listcodes (map code-abs-number listref))
898.                  (listallcodes
899.                   (cons "7"                 ; 7 pour VECTOR
900.                         (cons (code-abs-number (vector-length desc))
901.                               listcodes))))
902.             (apply string-append listallcodes))))))
903.
904. (define code-in-const
905.   (lambda (nbconst const-alist)
906.     (let* ((right-alist (reverse const-alist))
907.            (listdesc (map cddr right-alist))
908.            (listcodes (map code-one-const listdesc))
909.            (listallcodes (cons (code-abs-number nbconst) listcodes)))
910.       (apply string-append listallcodes))))
911.
912. (define code-top-const
913.   (lambda (nbtop top-alist)
914.     (let* ((right-alist (reverse top-alist))
915.            (listtop (map car right-alist))
916.            (listcodes (map code-abs-number listtop))
917.            (listallcodes (cons (code-abs-number nbtop) listcodes)))
918.       (apply string-append listallcodes))))
919.
920. (define code-const
921.   (lambda ()
922.     (string-append (code-in-const const-counter const-alist)
923.                    (code-top-const top-counter top-alist))))
924.
925.
926.
927.
928. ; Enregistrement des variables globales
929.
930. (define glob-counter 0)
931. (define glob-hidden '())  ; Variables cachees de la librairie
932. (define glob-public '())  ; Variables visibles de la librairie
933. (define glob-source '())  ; Variables introduites par le source
934. ; Chaque assoc: (nom . numero)
935. (define glob-v '#())
936. (define glob-v-len 0)
937.
938. (define glob-var-no
939.   (lambda (name lib?)
940.     (or
941.      (cond ((memq name libpublics)
942.             (let ((ass (assq name glob-public)))
943.               (if ass
944.                   (cdr ass)
945.                   (let ((newass (cons name glob-counter)))
946.                     (set! glob-public (cons newass glob-public))
947.                     #f))))
948.            (lib?
949.             (let ((ass (assq name glob-hidden)))
950.               (if ass
951.                   (cdr ass)
952.                   (let ((newass (cons name glob-counter)))
953.                     (set! glob-hidden (cons newass glob-hidden))
954.                     #f))))
955.            (else
956.             (let ((ass (assq name glob-source)))
957.               (if ass
958.                   (cdr ass)
959.                   (let ((newass (cons name glob-counter)))
960.                     (set! glob-source (cons newass glob-source))
961.                     #f)))))
962.      (begin
963.        (if (= glob-counter glob-v-len)
964.            (let* ((newlen (+ (* 2 glob-v-len) 1))
965.                   (newv (make-vector newlen)))
966.              (let loop ((pos 0))
967.                (if (< pos glob-v-len)
968.                    (begin
969.                      (vector-set! newv pos (vector-ref glob-v pos))
970.                      (loop (+ pos 1)))))
971.              (set! glob-v newv)
972.              (set! glob-v-len newlen)))
973.        (vector-set! glob-v glob-counter (make-globdesc-node name lib? 0))
974.        (set! glob-counter (+ glob-counter 1))
975.        (- glob-counter 1)))))
976.
977.
978.
979.
980. ; Localisation des variables
981.
982. ; Un resultat (glob . name) signifie que name est globale
983. ; Un resultat (lex #frame . #offset) donne le numero de frame et la
984. ;     position sur ce frame
985. ; Un resultat (lex #frame . #f) indique que name est seule sur son frame
986. (define where-var
987.   (lambda (name env)
988.     (if (null? env)
989.         (cons 'glob name)
990.         (let* ((locals (car env))
991.                (membership (memq name locals)))
992.           (if membership
993.               (let* ((nblocals (length locals))
994.                      (pos (- nblocals (length membership)))
995.                      (declpos (if (= nblocals 1) #f pos)))
996.                 (cons 'lex (cons 0 declpos)))
997.               (let ((result (where-var name (cdr env))))
998.                 (if (eq? (car result) 'glob)
999.                     result
1000.                    (let ((frame (cadr result))
1001.                          (offset (cddr result)))
1002.                      (cons 'lex (cons (+ frame 1) offset)))))))))
1003.
1004.
1005.
1006.
1007. ; Parcours des noeuds pour:
1008. ;     Numeroter les constantes si nec.
1009. ;     Identifier chaque variable
1010. ;     Compter les definitions et affectations des var. glob
```

```
1011. ;     Resumer les parametres formels
1012.
1013. (define traverse1-cte-node
1014.   (lambda (node env lib?)
1015.     (set-no! node (top-const-no (get-cte node)))))
1016.
1017. (define traverse1-ref-node
1018.   (lambda (node env lib?)
1019.     (let* ((name (get-symbol node))
1020.            (pos (where-var name env))
1021.            (glob? (eq? (car pos) 'glob))
1022.            (loc (if glob? (glob-var-no name lib?) (cdr pos))))
1023.       (set-glob?! node glob?)
1024.       (set-loc! node loc))))
1025.
1026. (define formals->fdesc
1027.   (lambda (formals)
1028.     (let loop ((nbreq 0) (formals formals))
1029.       (cond ((null? formals)
1030.              (cons nbreq #f))
1031.             ((symbol? formals)
1032.              (cons nbreq #t))
1033.             (else
1034.              (loop (+ nbreq 1) (cdr formals)))))))
1035.
1036. (define traverse1-lambda-node
1037.   (lambda (node env lib?)
1038.     (let* ((formals (get-formals node))
1039.            (varlist (formals->varlist formals))
1040.            (newenv (if (null? varlist) env (cons varlist env)))
1041.            (fdesc (formals->fdesc formals)))
1042.       (set-fdesc! node fdesc)
1043.       (traverse1-node (get-body node) newenv lib?))))
1044.
1045. (define traverse1-if-node
1046.   (lambda (node env lib?)
1047.     (traverse1-node (get-test node)   env lib?)
1048.     (traverse1-node (get-conseq node) env lib?)
1049.     (traverse1-node (get-altern node) env lib?)))
1050.
1051. (define traverse1-set!-node
1052.   (lambda (node env lib?)
1053.     (let* ((name (get-symbol node))
1054.            (pos (where-var name env))
1055.            (glob? (eq? (car pos) 'glob))
1056.            (loc (if glob? (glob-var-no name lib?) (cdr pos))))
1057.       (set-glob?! node glob?)
1058.       (set-loc! node loc)
1059.       (if glob?
1060.           (let* ((desc (vector-ref glob-v loc))
1061.                  (oldnb (get-nbaff desc)))
1062.             (set-nbaff! desc (+ oldnb 2)))))  ; Declare la var. mut.
1063.     (traverse1-node (get-exp node) env lib?)))
1064.
1065. (define traverse1-begin-node
1066.   (lambda (node env lib?)
1067.     (let ((lnode (get-lexp node)))
1068.       (for-each (lambda (node) (traverse1-node node env lib?)) lnode))))
1069.
1070. (define traverse1-def-node
1071.   (lambda (node env lib?)
1072.     (let* ((loc (glob-var-no (get-symbol node) lib?))
1073.            (desc (vector-ref glob-v loc))
1074.            (oldnb (get-nbaff desc)))
1075.       (set-loc! node loc)
1076.       (set-nbaff! desc (+ oldnb 1)))
1077.     (traverse1-node (get-exp node) env lib?)))
1078.
1079. (define traverse1-call-node
1080.   (lambda (node env lib?)
1081.     (let ((lnode (get-larg node)))
1082.       (traverse1-node (get-op node) env lib?)
1083.       (for-each (lambda (node) (traverse1-node node env lib?)) lnode))))
1084.
1085. (define traverse1-node
1086.   (let ((action-v
1087.          (vector
1088.           traverse1-cte-node
1089.           traverse1-ref-node
1090.           traverse1-lambda-node
1091.           traverse1-if-node
1092.           traverse1-set!-node
1093.           traverse1-begin-node
1094.           traverse1-def-node
1095.           traverse1-call-node)))
1096.     (lambda (node env lib?)
1097.       ((vector-ref action-v (node-type node)) node env lib?))))
1098.
1099. (define traverse1
1100.   (lambda ()
1101.     (for-each
1102.      (lambda (name)
1103.        (let* ((no (glob-var-no name #t))
1104.               (desc (vector-ref glob-v no)))
1105.          (set-nbaff! desc 1)))
1106.      allreq)
1107.     (for-each (lambda (node) (traverse1-node node '() #t))
1108.               (map cdr req-clos-nodes))
1109.     (for-each (lambda (node) (traverse1-node node '() #f))
1110.               lnode)))
1111.
1112.
1113.
1114.
1115. ; Determiner la valeur initiale des fonctions de la librairie
1116.
1117. (define find-an-init
1118.   (lambda (name)
1119.     (let ((asscprim (assq name libcprims)))
1120.       (if asscprim
1121.           (cons 'cprim (cdr asscprim))
1122.           (let ((assalias (assq name libalias)))
1123.             (if assalias
1124.                 (find-an-init (cdr assalias))
1125.                 (let ((node (cdr (assq name req-clos-nodes))))
1126.                   (if (lambda-node? node)
1127.                       (cons 'clos node)
1128.                       (begin
1129.                         (display "Error: fonct. de la lib. a env. non-vide: ")
1130.                         (write name)
1131.                         (newline)
1132.                         (cons 'clos 0))))))))))
1133.
1134. (define find-inits
1135.   (lambda ()
1136.     (let loop ((no 0))
1137.       (if (< no glob-counter)
1138.           (let ((desc (vector-ref glob-v no)))
```

```
1139.            (if (get-lib? desc)
1140.                (let* ((name (get-symbol desc))
1141.                       (init (find-an-init name)))
1142.                  (set-init! desc init)))
1143.            (loop (+ no 1)))))))
1144.
1145.
1146.
1147.
1148. ; Parcours des noeuds pour:
1149. ;     Resoudre a priori certaines references
1150. ;     "Inliner" lorsque possible
1151.
1152. (define reduce-list
1153.   '((append      2 append2)
1154.     (=           2 math=2)
1155.     (<           2 math<2)
1156.     (>           2 math>2)
1157.     (<=          2 math<=2)
1158.     (>=          2 math>=2)
1159.     (max         2 max2)
1160.     (min         2 min2)
1161.     (+           2 math+2)
1162.     (*           2 math*2)
1163.     (-           2 math-2)
1164.     (/           2 quotient)
1165.     (gcd         2 mathgcd2)
1166.     (lcm         2 mathlcm2)
1167.     (make-string 1 make-string1)
1168.     (make-vector 1 make-vector1)
1169.     (apply       2 apply1)
1170.     (map         2 map1)))
1171.
1172. (define reduced-function
1173.   (lambda (name nbargs)
1174.     (let ((rule (assq name reduce-list)))
1175.       (if (not rule)
1176.           #f
1177.           (if (= nbargs (cadr rule))
1178.               (caddr rule)
1179.               #f)))))
1180.
1181. (define optimize-call
1182.   (lambda (node match)
1183.     (let* ((symbol-field match)
1184.            (glob?-field  #t)
1185.            (loc-field    (glob-var-no match #t))
1186.            (var-desc     (vector-ref glob-v loc-field))
1187.            (val-field    (get-init var-desc))
1188.            (new-op       (make-ref-node-full symbol-field
1189.                                              loc-field
1190.                                              val-field
1191.                                              glob?-field)))
1192.       (set-op! node new-op))))
1193.
1194. (define reduce-call
1195.   (lambda (node)
1196.     (let* ((op (get-op node))
1197.            (name (get-symbol op))
1198.            (nbargs (length (get-larg node)))
1199.            (match (reduced-function name nbargs)))
1200.       (if match (optimize-call node match)))))
1201.
1202. (define traverse2-cte-node
1203.   (lambda (node lib?)
1204.     #t))
1205.
1206. (define traverse2-ref-node
1207.   (lambda (node lib?)
1208.     (if (get-glob? node)
1209.         (set-val! node
1210.                   (let* ((vardesc (vector-ref glob-v (get-loc node)))
1211.                          (varinit (get-init vardesc)))
1212.                     (if lib?
1213.                         varinit
1214.                         (if (not (get-lib? vardesc))
1215.                             #f
1216.                             (if (> (get-nbaff vardesc) 1)
1217.                                 #f
1218.                                 varinit))))))
1219.
1220. (define traverse2-lambda-node
1221.   (lambda (node lib?)
1222.     (traverse2-node (get-body node) lib?)))
1223.
1224. (define traverse2-if-node
1225.   (lambda (node lib?)
1226.     (traverse2-node (get-test node) lib?)
1227.     (traverse2-node (get-conseq node) lib?)
1228.     (traverse2-node (get-altern node) lib?)))
1229.
1230. (define traverse2-set!-node
1231.   (lambda (node lib?)
1232.     (traverse2-node (get-exp node) lib?)))
1233.
1234. (define traverse2-begin-node
1235.   (lambda (node lib?)
1236.     (for-each (lambda (node) (traverse2-node node lib?))
1237.               (get-lexp node))))
1238.
1239. (define traverse2-def-node
1240.   (lambda (node lib?)
1241.     (traverse2-node (get-exp node) lib?)))
1242.
1243. (define traverse2-call-node
1244.   (lambda (node lib?)
1245.     (let ((op (get-op node)))
1246.       (traverse2-node op lib?)
1247.       (for-each (lambda (node) (traverse2-node node lib?))
1248.                 (get-larg node))
1249.       (if (and (ref-node? op) (get-glob? op) (get-val op))
1250.           (reduce-call node)))))
1251.
1252. (define traverse2-node
1253.   (let ((action-v
1254.          (vector
1255.           traverse2-cte-node
1256.           traverse2-ref-node
1257.           traverse2-lambda-node
1258.           traverse2-if-node
1259.           traverse2-set!-node
1260.           traverse2-begin-node
1261.           traverse2-def-node
1262.           traverse2-call-node)))
1263.     (lambda (node lib?)
1264.       ((vector-ref action-v (node-type node)) node lib?))))
1265.
1266. (define traverse2
```

```
1267.   (lambda ()
1268.     (for-each (lambda (node) (traverse2-node node #t))
1269.               (map cdr req-clos-nodes))
1270.     (for-each (lambda (node) (traverse2-node node #f))
1271.               lnode)))
1272.
1273.
1274.
1275.
1276. ; Assigner des numeros physiques aux variables
1277.
1278. (define phys-glob-no 0)
1279.
1280. (define gen-phys-no
1281.   (lambda ()
1282.     (set! phys-glob-no (+ phys-glob-no 1))
1283.     (- phys-glob-no 1)))
1284.
1285. (define assign-phys-no
1286.   (lambda ()
1287.     (let loop ((no 0))
1288.       (if (< no glob-counter)
1289.           (let* ((desc (vector-ref glob-v no))
1290.                  (libvar? (get-lib? desc))
1291.                  (mutvar? (> (get-nbaff desc) 1)))
1292.             (cond ((not libvar?)
1293.                    (let ((phys-no (gen-phys-no)))
1294.                      (set-libno! desc phys-no)
1295.                      (set-srcno! desc phys-no)))
1296.                   (mutvar?
1297.                    (set-libno! desc (gen-phys-no))
1298.                    (set-srcno! desc (gen-phys-no)))
1299.                   (else
1300.                    (let ((phys-no (gen-phys-no)))
1301.                      (set-libno! desc phys-no)
1302.                      (set-srcno! desc phys-no))))
1303.             (loop (+ no 1)))))))
1304.
1305.
1306.
1307.
1308. ; Gestion des labels
1309.
1310. (define label-counter 0)
1311. (define label-v '#())
1312. (define label-v-len 0)
1313.
1314. (define make-label
1315.   (lambda ()
1316.     (if (= label-counter label-v-len)
1317.         (let* ((newlen (+ (* label-v-len 2) 1))
1318.                (newv (make-vector newlen)))
1319.           (let loop ((pos 0))
1320.             (if (< pos label-counter)
1321.                 (begin
1322.                   (vector-set! newv pos (vector-ref label-v pos))
1323.                   (loop (+ pos 1)))))
1324.           (set! label-v newv)
1325.           (set! label-v-len newlen)))
1326.     (set! label-counter (+ label-counter 1))
1327.     (- label-counter 1)))
1328.
1329.
1330.
1331.
1332. ; Generation du byte-code
1333.
1334. (define program-bytecode #f)
1335. (define flat-program-bytecode #f)
1336. (define final-program-bytecode #f)
1337.
1338. (define bcompile-no
1339.   (lambda (no)
1340.     (let ((msb (quotient no 256))
1341.           (lsb (modulo no 256)))
1342.       (list msb lsb))))
1343.
1344. (define bcompile-cte-null     ; 27 pour ()
1345.   (lambda ()
1346.     '(27)))
1347.
1348. (define bcompile-cte-boolean  ; 28 pour #f, 29 pour #t
1349.   (lambda (b)
1350.     (if b '(29) '(28))))
1351.
1352. (define bcompile-cte-char     ; 30 pour char
1353.   (lambda (c)
1354.     (list 30 (char->integer c))))
1355.
1356. (define bcompile-cte-number   ; courts: + 31 - 32, longs: + 33 - 34
1357.   (lambda (n)
1358.     (if (>= n 0)
1359.         (if (< n 256)
1360.             (list 31 n)
1361.             (list 33 (bcompile-no n)))
1362.         (if (< (- n) 256)
1363.             (list 32 (- n))
1364.             (list 34 (bcompile-no (- n)))))))
1365.
1366. (define bcompile-cte-imm
1367.   (lambda (cte)
1368.     (cond ((null? cte)
1369.            (bcompile-cte-null))
1370.           ((boolean? cte)
1371.            (bcompile-cte-boolean cte))
1372.           ((char? cte)
1373.            (bcompile-cte-char cte))
1374.           (else
1375.            (bcompile-cte-number cte)))))
1376.
1377. (define bcompile-cte-built
1378.   (lambda (no)
1379.     (if (< no 256)
1380.         (list 0 no)
1381.         (list 1 (bcompile-no no)))))
1382.
1383. (define bcompile-cte
1384.   (lambda (node tail? lib?)
1385.     (let* ((const-no (get-no node))
1386.            (get-cte-bc (if const-no
1387.                            (bcompile-cte-built const-no)
1388.                            (bcompile-cte-imm (get-cte node)))))
1389.       (if tail? (list get-cte-bc 14) get-cte-bc))))
1390.
1391. (define special-lex-pos
1392.   (lambda (frame offset)
1393.     (case frame
1394.       ((0)  (case offset ((0) 0) ((1) 1) ((2) 2) (else #f)))
```

```
1395.       ((1)  (case offset ((0) 3) ((1) 4) (else #f)))
1396.       ((2)  (case offset ((0) 5) (else #f)))
1397.       (else #f))))
1398.
1399. (define bcompile-ref-lex
1400.   (lambda (node)
1401.     (let* ((loc (get-loc node))
1402.            (frame (car loc))
1403.            (offset (cdr loc))
1404.            (spec (special-lex-pos frame (if offset offset 0))))
1405.       (cond (spec
1406.              (list (+ spec 36)))
1407.             (offset
1408.              (if (and (< frame 256) (< offset 256))
1409.                  (list 2 frame offset)
1410.                  (list 3 (bcompile-no frame) (bcompile-no offset))))
1411.             (else
1412.              (if (< frame 256)
1413.                  (list 2 frame)
1414.                  (list 3 (bcompile-no frame))))))))
1415.
1416. (define bcompile-ref-glob
1417.   (lambda (node lib?)
1418.     (let* ((loc (get-loc node))
1419.            (vardesc (vector-ref glob-v loc))
1420.            (phys-no (if lib? (get-libno vardesc) (get-srcno vardesc))))
1421.       (if (< phys-no 256)
1422.           (list 4 phys-no)
1423.           (list 5 (bcompile-no phys-no))))))
1424.
1425. (define bcompile-ref
1426.   (lambda (node tail? lib?)
1427.     (let ((result (if (get-glob? node)
1428.                       (bcompile-ref-glob node lib?)
1429.                       (bcompile-ref-lex node))))
1430.       (if tail? (list result 14) result))))
1431.
1432. (define bcompile-set!-lex
1433.   (lambda (node)
1434.     (let* ((loc (get-loc node))
1435.            (frame (car loc))
1436.            (offset (cdr loc)))
1437.       (if offset
1438.           (if (and (< frame 256) (< offset 256))
1439.               (list 6 frame offset)
1440.               (list 7 (bcompile-no frame) (bcompile-no offset)))
1441.           (if (< frame 256)
1442.               (list 6 frame)
1443.               (list 7 (bcompile-no frame)))))))
1444.
1445. (define bcompile-set!-glob
1446.   (lambda (node lib?)
1447.     (let* ((loc (get-loc node))
1448.            (vardesc (vector-ref glob-v loc))
1449.            (phys-no (if lib? (get-libno vardesc) (get-srcno vardesc))))
1450.       (if (< phys-no 256)
1451.           (list 8 phys-no)
1452.           (list 9 (bcompile-no phys-no))))))
1453.
1454. (define bcompile-set!
1455.   (lambda (node tail? lib?)
1456.     (let* ((exp (get-exp node))
1457.            (exp-bc (bcompile exp #f lib?))
1458.            (aff-bc (if (get-glob? node)
1459.                        (bcompile-set!-glob node lib?)
1460.                        (bcompile-set!-lex node)))
1461.            (set!-bc (list exp-bc aff-bc)))
1462.       (if tail? (list set!-bc 14) set!-bc))))
1463.
1464. (define bcompile-def
1465.   (lambda (node tail? lib?)
1466.     (let* ((exp (get-exp node))
1467.            (exp-bc (bcompile exp #f lib?))
1468.            (aff-bc (bcompile-set!-glob node lib?))
1469.            (def-bc (list exp-bc aff-bc)))
1470.       (if tail? (list def-bc 14) def-bc))))
1471.
1472. (define bcompile-pop-n
1473.   (lambda (n)
1474.     (cond ((= n 1)
1475.            '(51))
1476.           ((< n 256)
1477.            (list 49 n))
1478.           (else
1479.            (list 50 (bcompile-no n))))))
1480.
1481. (define bcompile-begin
1482.   (lambda (node tail? lib?)
1483.     (let loop ((lexp (get-lexp node)) (nb-prev 0))
1484.       (if (null? (cdr lexp))
1485.           (list (bcompile-pop-n nb-prev)
1486.                 (bcompile (car lexp) tail? lib?))
1487.           (list (bcompile (car lexp) #f lib?)
1488.                 (loop (cdr lexp) (+ nb-prev 1)))))))
1489.
1490. (define bcompile-label-def
1491.   (lambda (no)
1492.     (list 'def no)))
1493.
1494. (define bcompile-label-ref
1495.   (lambda (no)
1496.     (list 'ref no)))
1497.
1498. (define bcompile-if
1499.   (lambda (node tail? lib?)
1500.     (let* ((debut-altern    (make-label))
1501.            (fin-altern      (if tail? #f (make-label)))
1502.            (test-bc         (bcompile (get-test node) #f lib?))
1503.            (cjump-bc        (list 11 (bcompile-label-ref debut-altern)))
1504.            (conseq-bc       (bcompile (get-conseq node) tail? lib?))
1505.            (ujump-bc        (if tail?
1506.                                 '()
1507.                                 (list 12 (bcompile-label-ref fin-altern))))
1508.            (debut-altern-bc (bcompile-label-def debut-altern))
1509.            (altern-bc       (bcompile (get-altern node) tail? lib?))
1510.            (fin-altern-bc   (if tail?
1511.                                 '()
1512.                                 (bcompile-label-def fin-altern))))
1513.       (list test-bc cjump-bc conseq-bc ujump-bc
1514.             debut-altern-bc altern-bc fin-altern-bc))))
1515.
1516. (define bcompile-make-frame
1517.   (lambda (fdesc)
1518.     (let* ((nbreq (car fdesc))
1519.            (fac? (cdr fdesc))
1520.            (frame-size (+ nbreq (if fac? 1 0))))
1521.       (cond ((= frame-size 0)
1522.              '()))
```

```
1523.            ((and (= frame-size 1) (not fac?))
1524.             '(42))
1525.            ((and (= frame-size 2) (not fac?))
1526.             '(43))
1527.            ((and (= frame-size 1) fac?)
1528.             '(44))
1529.            (fac?
1530.             (if (< frame-size 256)
1531.                 (list 22 frame-size)
1532.                 (list 23 (bcompile-no frame-size))))
1533.            (else
1534.             (if (< frame-size 256)
1535.                 (list 20 frame-size)
1536.                 (list 21 (bcompile-no frame-size))))))))
1537.
1538. (define bcompile-closure
1539.   (lambda (node lib?)
1540.     (let* ((fdesc (get-fdesc node))
1541.            (make-frame-bc (bcompile-make-frame fdesc))
1542.            (body-bc (bcompile (get-body node) #t lib?)))
1543.       (list make-frame-bc body-bc))))
1544.
1545. (define bcompile-lambda
1546.   (lambda (node tail? lib?)
1547.     (let ((clos-bc (bcompile-closure node lib?)))
1548.       (if tail?
1549.           (list 10 clos-bc)
1550.           (let* ((suite (make-label))
1551.                  (make-clos-bc (list 48 (bcompile-label-ref suite)))
1552.                  (suite-bc (bcompile-label-def suite)))
1553.             (list make-clos-bc clos-bc suite-bc))))))
1554.
1555. (define bcompile-calc-args
1556.   (lambda (larg lib?)
1557.     (let loop ((larg larg) (prev-args-bc '()))
1558.       (if (null? larg)
1559.           prev-args-bc
1560.           (let* ((arg (car larg))
1561.                  (reste (cdr larg))
1562.                  (calc-arg-bc (bcompile arg #f lib?)))
1563.             (loop reste (list calc-arg-bc prev-args-bc)))))))
1564.
1565. (define bcompile-call-C
1566.   (lambda (node tail? lib?)
1567.     (let ((cprim-no (cdr (get-val (get-op node)))))
1568.       (if (= cprim-no apply1-cprim-no)
1569.           (bcompile-call-I node tail? lib?)     ; Le cas apply
1570.           (let* ((larg (get-larg node))
1571.                  (calc-args-bc (bcompile-calc-args larg lib?))
1572.                  (apply-bc (list (- 255 cprim-no)))
1573.                  (call-bc (list calc-args-bc apply-bc)))
1574.             (if tail?
1575.                 (list 15 call-bc 14)
1576.                 (list 25 call-bc 26)))))))
1577.
1578. (define bcompile-call-Fi
1579.   (lambda (node tail? lib?)
1580.     (let* ((op (get-op node))
1581.            (larg (get-larg node))
1582.            (calc-args-bc (bcompile-calc-args larg lib?))
1583.            (fdesc (get-fdesc op))
1584.            (make-frame-bc (bcompile-make-frame fdesc))
1585.            (body (get-body op))
1586.            (body-bc (bcompile body tail? lib?)))
1587.       (if tail?
1588.           (list 15 calc-args-bc make-frame-bc body-bc)
1589.           (list 25 calc-args-bc make-frame-bc body-bc 26 35)))))
1590.
1591. (define bcompile-call-I
1592.   (lambda (node tail? lib?)
1593.     (let ((op (get-op node)))
1594.       (if (and (ref-node? op) (get-glob? op))
1595.           (let* ((larg (get-larg node))
1596.                  (calc-args-bc (bcompile-calc-args larg lib?))
1597.                  (var-desc (vector-ref glob-v (get-loc op)))
1598.                  (phys-no (if lib? (get-libno var-desc) (get-srcno var-desc))))
1599.             (if (< phys-no 256)
1600.                 (if tail?
1601.                     (list 15 calc-args-bc 52 phys-no)
1602.                     (list 45 calc-args-bc 54 phys-no))
1603.                 (if tail?
1604.                     (list 15 calc-args-bc 53 (bcompile-no phys-no))
1605.                     (list 45 calc-args-bc 55 (bcompile-no phys-no)))))
1606.           (let* ((larg (get-larg node))
1607.                  (allarg (cons op larg))
1608.                  (allarg-bc (bcompile-calc-args allarg lib?)))
1609.             (if tail?
1610.                 (list 15 allarg-bc 17)
1611.                 (list 45 allarg-bc 46)))))))
1612.
1613. (define bcompile-call
1614.   (lambda (node tail? lib?)
1615.     (let ((op (get-op node)))
1616.       (cond ((ref-node? op)
1617.              (let ((val (get-val op)))
1618.                (if (and val (eq? (car val) 'cprim))
1619.                    (bcompile-call-C node tail? lib?)
1620.                    (bcompile-call-I node tail? lib?))))
1621.             ((lambda-node? op)
1622.              (bcompile-call-Fi node tail? lib?))
1623.             (else
1624.              (bcompile-call-I node tail? lib?))))))
1625.
1626. (define bcompile
1627.   (let ((action-v
1628.          (vector
1629.           bcompile-cte
1630.           bcompile-ref
1631.           bcompile-lambda
1632.           bcompile-if
1633.           bcompile-set!
1634.           bcompile-begin
1635.           bcompile-def
1636.           bcompile-call)))
1637.     (lambda (node tail? lib?)
1638.       ((vector-ref action-v (node-type node)) node tail? lib?))))
1639.
1640. (define bcompile-program
1641.   (lambda ()
1642.     (map (lambda (ass) (set-label! (cdr ass) (make-label)))
1643.          req-clos-nodes)
1644.     (let* ((source-bc (map (lambda (node) (list (bcompile node #f #f) 51))
1645.                            lnode))
1646.            (fin-bc (list 24))
1647.            (lib-bc (map (lambda (ass)
1648.                           (let ((node (cdr ass)))
1649.                             (list (bcompile-label-def (get-label node))
1650.                                   (bcompile-closure node #t))))
```

```
1651.                        req-clos-nodes))
1652.            (apply-hook-bc (list 17)))
1653.       (list source-bc fin-bc lib-bc apply-hook-bc))))
1654.
1655. (define flatten-bytecode
1656.   (lambda (hierarcode)
1657.     (let loop ((h hierarcode) (rest '()))
1658.       (if (pair? h)
1659.           (loop (car h) (loop (cdr h) rest))
1660.           (if (null? h)
1661.               rest
1662.               (cons h rest))))))
1663.
1664. (define link-bytecode
1665.   (lambda (flat-reloc-bc)
1666.     (let loop ((bc flat-reloc-bc) (pos 0))
1667.       (if (not (null? bc))
1668.           (let ((head (car bc)))
1669.             (cond ((number? head)
1670.                    (loop (cdr bc) (+ pos 1)))
1671.                   ((eq? head 'ref)
1672.                    (loop (cddr bc) (+ pos 2)))
1673.                   (else ; (eq? head 'def)
1674.                    (let ((label-no (cadr bc)))
1675.                      (vector-set! label-v label-no pos)
1676.                      (loop (cddr bc) pos))))))))
1677.     (let loop ((bc flat-reloc-bc))
1678.       (if (null? bc)
1679.           '()
1680.           (let ((head (car bc)))
1681.             (cond ((number? head)
1682.                    (cons head (loop (cdr bc))))
1683.                   ((eq? head 'ref)
1684.                    (let* ((label-no (cadr bc))
1685.                           (pos (vector-ref label-v label-no)))
1686.                      (append (bcompile-no pos) (loop (cddr bc)))))
1687.                   (else ; (eq? head 'def)
1688.                    (loop (cddr bc))))))))
1689.
1690.
1691.
1692.
1693. ; Codage de la valeur initiale des variables globales
1694.
1695. (define glob-var-init-codes #f)
1696.
1697. (define code-init
1698.   (lambda (init)
1699.     (cond ((not init)
1700.            -1)
1701.           ((eq? (car init) 'cprim)
1702.            (- -2 (cdr init)))
1703.           (else ; (eq? (car init) 'clos)
1704.            (let* ((lambda-node (cdr init))
1705.                   (label-no (get-label lambda-node)))
1706.              (vector-ref label-v label-no))))))
1707.
1708. (define code-glob-inits
1709.   (lambda ()
1710.     (let loop ((var-no (- glob-counter 1)) (codes '()))
1711.       (if (< var-no 0)
1712.           codes
1713.           (let* ((var-desc (vector-ref glob-v var-no))
1714.                  (var-init (get-init var-desc))
1715.                  (init-code (code-init var-init))
1716.                  (newcodes (if (= (get-libno var-desc)
1717.                                   (get-srcno var-desc))
1718.                                (cons init-code codes)
1719.                                (cons init-code (cons init-code codes)))))
1720.             (loop (- var-no 1) newcodes))))))
1721.
1722.
1723.
1724.
1725. ; Impression des resultats
1726.
1727. (define byte-strings
1728.   (vector
1729.    "   0" "   1" "   2" "   3" "   4" "   5" "   6" "   7" "   8" "   9"
1730.    "  10" "  11" "  12" "  13" "  14" "  15" "  16" "  17" "  18" "  19"
1731.    "  20" "  21" "  22" "  23" "  24" "  25" "  26" "  27" "  28" "  29"
1732.    "  30" "  31" "  32" "  33" "  34" "  35" "  36" "  37" "  38" "  39"
1733.    "  40" "  41" "  42" "  43" "  44" "  45" "  46" "  47" "  48" "  49"
1734.    "  50" "  51" "  52" "  53" "  54" "  55" "  56" "  57" "  58" "  59"
1735.    "  60" "  61" "  62" "  63" "  64" "  65" "  66" "  67" "  68" "  69"
1736.    "  70" "  71" "  72" "  73" "  74" "  75" "  76" "  77" "  78" "  79"
1737.    "  80" "  81" "  82" "  83" "  84" "  85" "  86" "  87" "  88" "  89"
1738.    "  90" "  91" "  92" "  93" "  94" "  95" "  96" "  97" "  98" "  99"
1739.    " 100" " 101" " 102" " 103" " 104" " 105" " 106" " 107" " 108" " 109"
1740.    " 110" " 111" " 112" " 113" " 114" " 115" " 116" " 117" " 118" " 119"
1741.    " 120" " 121" " 122" " 123" " 124" " 125" " 126" " 127" " 128" " 129"
1742.    " 130" " 131" " 132" " 133" " 134" " 135" " 136" " 137" " 138" " 139"
1743.    " 140" " 141" " 142" " 143" " 144" " 145" " 146" " 147" " 148" " 149"
1744.    " 150" " 151" " 152" " 153" " 154" " 155" " 156" " 157" " 158" " 159"
1745.    " 160" " 161" " 162" " 163" " 164" " 165" " 166" " 167" " 168" " 169"
1746.    " 170" " 171" " 172" " 173" " 174" " 175" " 176" " 177" " 178" " 179"
1747.    " 180" " 181" " 182" " 183" " 184" " 185" " 186" " 187" " 188" " 189"
1748.    " 190" " 191" " 192" " 193" " 194" " 195" " 196" " 197" " 198" " 199"
1749.    " 200" " 201" " 202" " 203" " 204" " 205" " 206" " 207" " 208" " 209"
1750.    " 210" " 211" " 212" " 213" " 214" " 215" " 216" " 217" " 218" " 219"
1751.    " 220" " 221" " 222" " 223" " 224" " 225" " 226" " 227" " 228" " 229"
1752.    " 230" " 231" " 232" " 233" " 234" " 235" " 236" " 237" " 238" " 239"
1753.    " 240" " 241" " 242" " 243" " 244" " 245" " 246" " 247" " 248" " 249"
1754.    " 250" " 251" " 252" " 253" " 254" " 255"))
1755.
1756. (define write-bytecode
1757.   (lambda (bc port)
1758.     (display "int bytecode_len = " port)
1759.     (let ((len (length bc)))
1760.       (write len port)
1761.       (if (> len 32768)
1762.           (begin
1763.             (display "Warning: bytecode too long.")
1764.             (newline))))
1765.     (display ";" port)
1766.     (newline port)
1767.     (display "unsigned char bytecode[] = {" port)
1768.     (let ((virgule ""))
1769.       (let loop ((bc bc) (mod 0))
1770.         (if (not (null? bc))
1771.             (begin
1772.               (display virgule port)
1773.               (set! virgule ",")
1774.               (if (= mod 0)
1775.                   (begin
1776.                     (newline port)
1777.                     (display "       " port)
1778.                     (set! mod -12)))
```

```
1779.             (display (vector-ref byte-strings (car bc)) port)
1780.             (loop (cdr bc) (+ mod 1))))))
1781.     (display "};" port)
1782.     (newline port)))
1783.
1784. (define write-const-desc
1785.   (lambda (cd port)
1786.     (let ((cd (map char->integer (string->list cd))))
1787.       (display "int const_desc_len = " port)
1788.       (write (length cd) port)
1789.       (display ";" port)
1790.       (newline port)
1791.       (display "unsigned char const_desc[] = {" port)
1792.       (let ((virgule ""))
1793.         (let loop ((cd cd) (mod 0))
1794.           (if (not (null? cd))
1795.               (begin
1796.                 (display virgule port)
1797.                 (set! virgule ",")
1798.                 (if (= mod 0)
1799.                     (begin
1800.                       (newline port)
1801.                       (display "        " port)
1802.                       (set! mod -12)))
1803.                 (display (vector-ref byte-strings (car cd)) port)
1804.                 (loop (cdr cd) (+ mod 1))))))
1805.       (display "};" port)
1806.       (newline port))))
1807.
1808. (define pretty-signed-int
1809.   (lambda (int)
1810.     (let* ((sint (number->string int))
1811.            (lpadding (- 6 (string-length sint)))
1812.            (padding (substring "        " 0 lpadding)))
1813.       (string-append padding sint))))
1814.
1815. (define write-glob-init-codes
1816.   (lambda (glob-var-init-codes port)
1817.     (display "int nb_scm_globs = " port)
1818.     (write (length glob-var-init-codes) port)
1819.     (display ";" port)
1820.     (newline port)
1821.     (display "int scm_globs[] = {" port)
1822.     (let ((virgule ""))
1823.       (let loop ((gi glob-var-init-codes) (mod 0))
1824.         (if (not (null? gi))
1825.             (begin
1826.               (display virgule port)
1827.               (set! virgule ",")
1828.               (if (= mod 0)
1829.                   (begin
1830.                     (newline port)
1831.                     (display "        " port)
1832.                     (set! mod -8)))
1833.               (display (pretty-signed-int (car gi)) port)
1834.               (loop (cdr gi) (+ mod 1))))))
1835.     (display "};" port)
1836.     (newline port)))
1837.
1838. (define write-output
1839.   (lambda (final-program-bytecode
1840.            const-desc-string
1841.            glob-var-init-codes
1842.            nameout)
1843.     (let ((port (open-output-file nameout)))
1844.       (write-bytecode       final-program-bytecode port) (newline port)
1845.       (write-const-desc     const-desc-string      port) (newline port)
1846.       (write-glob-init-codes glob-var-init-codes   port)
1847.       (close-output-port port))))
1848.
1849.
1850.
1851.
1852. ; Programme principal
1853.
1854. (define byte-compile
1855.   (lambda (namein nameout)
1856.     (init-glob-vars)
1857.     (let* ((source (read-source namein))
1858.            (source-symbols (find-all-symbols source))
1859.            (uniq-pref (find-uniq-prefix source-symbols)))
1860.       (set! gen-sym-pref uniq-pref)
1861.       (set! safe-name-memv        (gen-sym))
1862.       (set! safe-name-make-promise (gen-sym))
1863.       (set! safe-name-list->vector (gen-sym))
1864.       (set! safe-name-list        (gen-sym))
1865.       (set! safe-name-append2     (gen-sym))
1866.       (set! safe-name-cons        (gen-sym))
1867.       (let* ((source++
1868.               (append
1869.                (list
1870.                 (list 'define safe-name-memv        'memv)
1871.                 (list 'define safe-name-make-promise 'make-promise)
1872.                 (list 'define safe-name-list->vector 'list->vector)
1873.                 (list 'define safe-name-list        'list)
1874.                 (list 'define safe-name-append2     'append2)
1875.                 (list 'define safe-name-cons        'cons)
1876.                 '(install-const))
1877.                source))
1878.              (simple (map trans-sub source++)))
1879.         (set! lnode (map exp->node simple))
1880.         (let* ((llglob (map extract-glob-names lnode))
1881.                (lglob (foldr symbol-set-union '() llglob)))
1882.           (load-lib)
1883.           (set! dirreq (symbol-set-intersection lglob libpublics))
1884.           (grab-lib dirreq)
1885.           (traverse1)
1886.           (find-inits)
1887.           (traverse2)
1888.           (assign-phys-no)
1889.           (set! program-bytecode (bcompile-program))
1890.           (set! flat-program-bytecode (flatten-bytecode program-bytecode))
1891.           (set! final-program-bytecode (link-bytecode flat-program-bytecode))
1892.           (set! const-desc-string (code-const))
1893.           (set! glob-var-init-codes (code-glob-inits))
1894.           (write-output final-program-bytecode const-desc-string
1895.                         glob-var-init-codes nameout))))))
```

A.4 Exemple de compilation vers du code-octets

Supposons que l'on compile le petit programme suivant:

```
(display "Hello world!")
(newline)
```

Le fichier produit par le compilateur vers du code-octets est le suivant (pour plus de clarté, plusieurs lignes sont élidées):

```
int bytecode_len = 2594;
unsigned char bytecode[] = {
          4,  93,   8,  94,  51,   4,  75,   8,  95,  51,   4,  74,
          8,  96,  51,   4,  55,   8,  97,  51,   4,  54,   8,  98,
         51,   4,  61,   8,  99,  51,  45,  54,  52,  51,  45,   0,
          1,  54,  33,  51,  45,  54,   0,  51,  24,  15,  30,  10,
                                     ...
         54,  91,  11,  10,   3,  28,  14,  45,  37,  25,  38, 249,
         26,  36,  46,  11,  10,  16,  38,  14,  15,  25,  38, 248,
         26,  37,  36,  52,  92,  43,  15,  37,  36,   4,  87,  52,
         92,  17};

int const_desc_len = 27;
unsigned char const_desc[] = {
          0,   2,  52,   0,   1,  48,  52,   0,  12,  72, 101, 108,
        108, 111,  32, 119, 111, 114, 108, 100,  33,   0,   2,   0,
          0,   0,   1};

int nb_scm_globs = 100;
int scm_globs[] = {
           45,   -24,    50,    57,   -11,    71,   136,   150,
          -36,   -36,   212,   290,   -16,   316,   -18,   -17,
          485,   836,   -13,   891,    -7,   954,   963,   -25,
          970,   998,  1044,   -38,  1074,  1106,  1152,  1159,
          -21,  1258,   -26,   -30,  1333,   -35,   -35,  1357,
         1366,  1401,   -37,   -19,   -14,   -15,  1470,  1524,
          -41,  1557,  1857,  1866,  1873,   -29,  1966,  2041,
         2044,   -22,   -23,   -34,   -34,   -27,  2085,  2122,
         2148,  2154,  2161,  2215,   -33,   -33,  2248,  2275,
          -42,  2284,  2293,  2362,   -12,  2397,   -10,   -32,
          -32,  2409,  2425,  2433,  2441,  2487,   -20,  2494,
           -9,    -8,   -39,  2546,  2552,  2585,    -1,    -1,
           -1,    -1,    -1,    -1};
```