Master d'Informatique
Parcours AIGLE

HMIN 104 Compilation et Interprétation

Compilation et Interprétation des Langages

24 novembre 2016

Roland Ducournau ducournau@lirmm.fr
http://www.lirmm.fr/~ducour
Mathieu Lafourcade lafourcade@lirmm.fr
http://www.lirmm.fr/~lafourcade

Département d'Informatique
—
Faculté des Sciences
—
Université de Montpellier

Avertissement Ce polycopié a couvert les versions successives du module de compilation de Maîtrise Informatique (module intitulé *Langage, Evaluation, Compilation* LEC, jusqu'en 2004) puis de première année de Master (modules intitulés *Evaluation des Langages Applicatifs* ELA et *Génération de Code* GDC, jusqu'en 2007, *Compilation* de 2008 à 2010, enfin *Compilation et Interprétation* à partir de la rentrée 2011).

C'est un chantier perpétuel qui doit être complété par le polycopié sur le langage LISP [Duc13a] et par les pages web des auteurs :

`http://www.lirmm.fr/~ducour/` → `Enseignement`.
`http://www.lirmm.fr/~lafourcade/` → `Enseignements`.

1

Introduction

Ce polycopié couvre une grande partie du cours de compilation du Master Informatique de l'Université Montpellier 2.

Bibliographie

Il est complété :
— par un polycopié de LISP, langage qui sert de support au cours [Duc13a] ;
— par le polycopié en ligne de Mathieu Lafourcade sur la partie *Génération de code* : http://www.lirmm.fr/~lafourcade/ML-enseign/Compilation/compil-GenCode.HTML

Il y a de nombreux ouvrages disponibles sur la compilation. [ASU89] est l'un des plus classiques. On trouve aussi de nombreux ouvrages plus récents en ligne, par exemple un ouvrage sur la compilation et la génération de compilateurs avec application à C++ [Ter96] http://scifac.ru.ac.za/compilers/.

Plan du polycopié

Le plan de ce polycopié est le suivant :
— le chapitre 2 introduit de façon générale la problématique de l'interprétation et de la compilation des langages ;
— le chapitre 3 présente l'exemple simple de la compilation et de l'interprétation des automates et des expressions régulières ;
— le chapitre 4 développe une première version naïve d'un évaluateur du langage COMMON LISP ;
— le chapitre 5 en fait une deuxième version, plus sophistiquée, qui mixe les deux approches de la compilation et de l'interprétation, en passant par un langage intermédiaire ;
— le chapitre 6 reprend le principe général de l'analyse par cas qui sous-tend les chapitres précédents ;
— le chapitre 7 décrit le langage de la machine virtuelle à registres ;
— le chapitre 8 décrit l'implémentation en LISP de la machine virtuelle à registres (qui est traitée en détails dans le polycopié de Mathieu Lafourcade) qui constitue le langage cible de la génération de code ;
— le chapitre 9 décrit le principe d'une machine virtuelle à pile, et la façon de transformer le code LISP (ou du langage intermédiaire du Chapitre 5) dans le langage de la machine à pile ;
— le chapitre 10 présente enfin les grandes lignes du projet à réaliser (dont le sujet précis varie suivant les années).

2

Evaluation et compilation

2.1 Interprétation et compilation

L'exécution d'un programme peut être mise en œuvre de façons variées, entre les deux extrêmes que constituent les notions d'*interprète* et de *compilateur* dans une première définition classique mais provisoire.

Définition 1 *Un* compilateur *est un programme qui transforme un programme écrit dans un langage évolué en un programme directement exécutable sur une machine.*

Définition 2 *Un* interprète *est un programme qui exécute des programmes.*

Les figures 2.1, 2.2 et 2.3 décrivent successivement un système général de traitement de langages compilés, un compilateur et un interprète.

Ces définitions strictes peuvent être assouplies de plusieurs façons :

— d'une part, on peut interpréter ou compiler bien d'autres choses que des programmes écrits dans un langage de programmation : par exemple, des automates, des règles, un traitement de texte ou de graphiques, une machine virtuelle, etc.

— d'autre part, les schémas d'interprétation et de compilation peuvent s'appliquer beucoup plus localement que ne le laissent entendre ces définitions globales : en particulier, la cible de la compilation peut ne pas être un langage machine « concret », mais un langage machine « virtuel » (ou « abstrait »), voire un langage évolué, mais moins que le langage source [1].

On pourra donc donner des notions d'*interprétation* et de *compilation* les définitions plus générales suivantes :

Définition 3 *L'*interprétation *est un mode d'exécution des programmes qui repose sur l'analyse* dynamique *des instructions (ou expressions) du programme en parallèle avec les données sur lesquelles ces instructions s'exécutent.*

Définition 4 *La* compilation *est une transformation des programmes qui permet d'effectuer* statiquement *une partie des calculs sur les instructions du programme, indépendamment des données, au lieu d'attendre l'exécution.*

De façon générale, on peut voir un programme comme une *analyse par cas* (voir les chapitres suivants, en particulier le Chapitre 6) sur une donnée qui est en quelque sorte interprétée. Dans un interprète, on a deux programmes, l'interpréteur et l'interprété, et une double analyse par cas, sur les instructions du programme interprété d'un côté, et sur les données de l'autre. En revanche, dans l'exécution d'un programme

1. On se gardera néanmoins de traiter de compilation toute opération de traduction entre deux langages. En particulier, l'opération inverse de la traduction, consistant à produire du code évolué — en général humainement illisible — à partir de langage d'assemblage, s'appelle une *décompilation*.

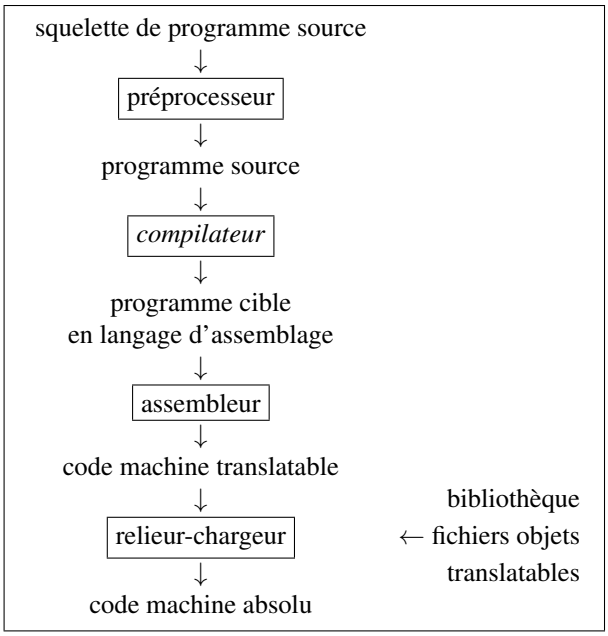


FIGURE 2.1 – Un système de traitement de langage, d'après [ASU89]

compilé, l'analyse par cas du programme a été effectuée à la compilation, sans les données, et il ne reste plus qu'à faire l'analyse par cas des données.

Dans ce sens très général, la compilation constitue un investissement, qui peut être onéreux, mais qui peut rapporter gros, le travail statique n'étant fait qu'une seule fois pour de potentiellement très nombreuses exécutions différentes. Inversement, l'interprétation est un mode d'exécution très rapide à mettre en œuvre, mais souvent peu efficace, trop de choses étant repoussées à l'exécution.

En conséquence, la compilation est idéale pour les programmes en phase d'exploitation, alors que l'interprétation est à préférer lors du développement et de la mise au point.

2.1.1 La compilation

De façon classique, la compilation est constituée de 3 grandes étapes :

— l'analyse lexicale permet de décomposer le texte du programme en une suite d'unités lexicales (des « mots » et des « ponctuations » : on parle souvent de *tokens*) ;

— l'analyse syntaxique produit la structure du programme sous forme d'un arbre dont les feuilles sont les unités lexicales ;

— la génération de code réalise la traduction dans le langage cible.

Se greffent à ce schéma ternaire, des phases plus secondaires comme l'analyse sémantique (réduite essentiellement au typage) et l'optimisation. Cette dernière peut se faire, d'abord par des transformations source à source, ensuite par des transformations du code produit. Plus le langage est évolué, plus il peut donner lieu à des optimisations par des transformations source à source. En revanche, les optimisations du code assembleur sont relativement communes à tous les langages mais très dépendantes des processeurs.

Transversalement, toutes ces phases ont en commun de gérer des symboles (noms) et des erreurs : chacun sait, qu'avant de produire du code cible, un compilateur produit surtout beaucoup de messages d'erreur !

2.1.2 L'interprétation

L'examen des figures 2.2 et 2.3 montre que l'interprétation et la compilation débutent par les mêmes phases d'analyse lexicale et syntaxique. Ces deux phases produisent une forme structurée du programme qui est la base des étapes ultérieures, *génération de code* pour la compilation, et *évaluation* pour l'interprétation.

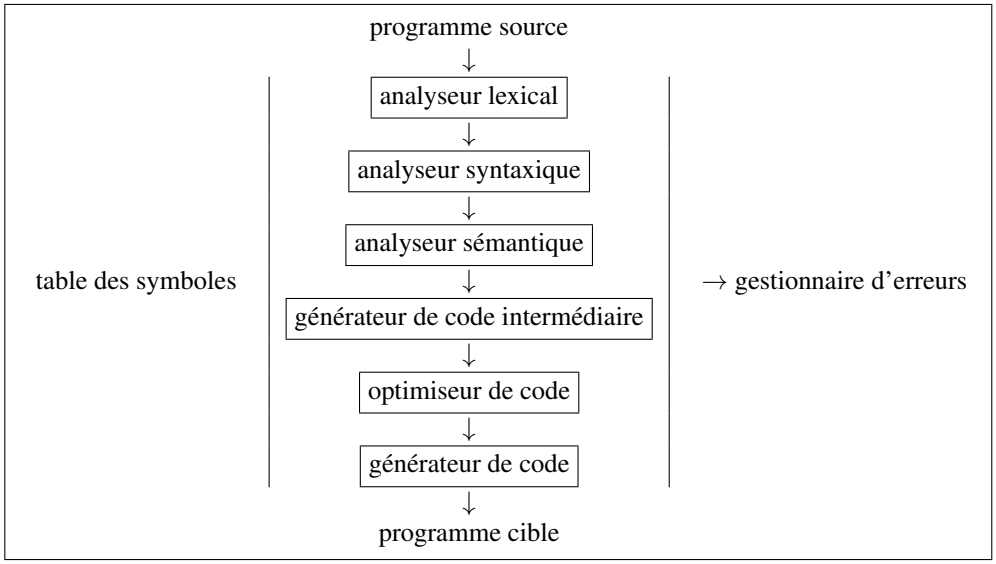


FIGURE 2.2 – Phases d'un compilateur, d'après [ASU89]

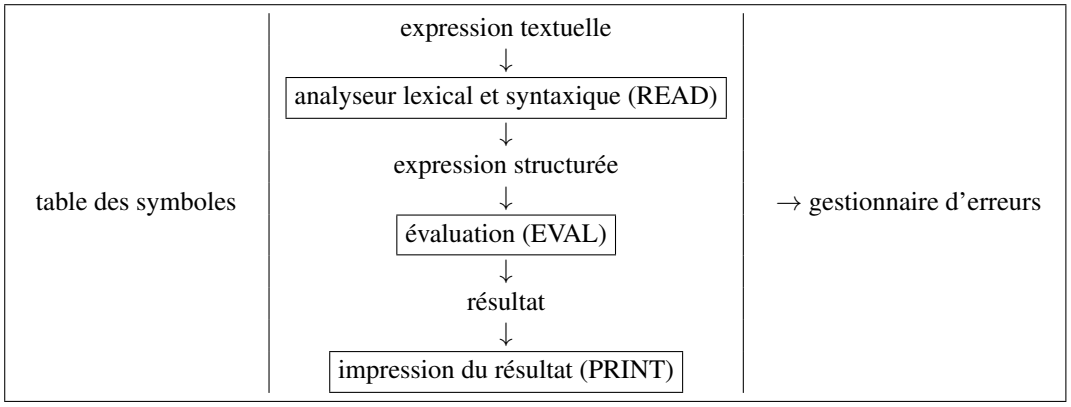


FIGURE 2.3 – Phases d'un interprète : la boucle READ-EVAL-PRINT

Définition 5 *L'*évaluateur *est la partie de l'interprète qui prend en entrée le programme (ou expression) sous une forme structurée et l'exécute effectivement pour retourner un résultat, sa* valeur*, après avoir fait d'éventuels effets de bord.*

L'interprétation est basée sur l'itération sans fin de ces trois phases :
— lecture et analyse d'une expression ;
— évaluation de cette expression ;
— impression du résultat.
C'est la boucle classique `READ-EVAL-PRINT`, dite du *toplevel* de l'interprète.

Il est particulièrement important de distinguer, aussi bien en pratique qu'en théorie, ce qui relève de `READ` et ce qui relève d'`EVAL`. Ainsi, initialiser une variable par l'expression `'(1 . 2)` ou par l'expression `(cons 1 2)` revient au même d'un point de vue purement « fonctionnel », mais ce n'est plus le cas dès que la gestion mémoire ou des effets de bord sont considérés : en effet, pour `'(1 . 2)`, une cellule unique est allouée par `READ`, alors que pour `(cons 1 2)`, un cellule différente est allouée par `EVAL` chaque fois que le flot de contrôle passe par l'expression (voir polycopié de LISP [Duc13a], Chapitre 2).

Face à l'expression `(cons 1 2)`, un compilateur pourrait juger utile de la simplifier en `'(1 . 2)`, mais ce ne serait pas forcément malin à cause de possibles effets de bord.

2.1.3 Points communs

Interprétation et compilation ont deux points communs : les phases d'analyse syntaxique et de transformation source à source.

Analyses lexicale et syntaxique. A partir d'expression purement textuelles, elles produisent un arbre syntaxique qui sert de donnée aux étapes ultérieures. C'est très exactement le rôle de la fonction `READ`. Il n'y a pas de nécessité à ce que l'arbre syntaxique diffère entre la compilation et l'interprétation.

Cependant, il n'y a pas de raison non plus que l'évaluation s'effectue, en LISP, directement sur l'arbre syntaxique LISP naturel, c'est-à-dire la liste telle qu'elle est lue par la fonction `READ`. Un arbre syntaxique étiqueté peut rendre l'évaluation plus rapide : les différents cas sont différenciés par des étiquettes (petit entier), les noms des variables lexicales sont remplacés par des indices, l'environnement est alloué dans la pile, etc.

Transformations source à source. Une transformation source à source prend un arbre syntaxique correct et retourne un arbre syntaxique correct, en général dans un sous-ensemble du langage. La transformation source à source a 2 objetifs : réduire le nombre de cas particuliers à traiter et effectuer des optimisations à plus haut niveau.

Plutôt que de traiter de façon distincte des formes syntaxiques voisines (par exemple les multiples formes de conditionnelles, en LISP `if`, `unless`, `when`, `cond`, `case`, etc.), ce qui entraînerait de nombreuses redondances avec des risques d'erreur, l'objectif des transformations source à source est de se ramener à une unique forme primitive (poar exemple `if`).

Les transformations source à source peuvent être implémentées dans l'interprète ou le compilateur, mais elles peuvent aussi être intégrées au langage lui-même. C'est le cas lorsqu'il dispose d'un préprocesseur, ou avec les *macros* LISP.

Dans ce dernier cas, le langage peut distinguer les macros usuelles, expansées à l'évaluation (ainsi qu'à la compilation) des macros réservées au compilateur (voir `compiler-macro` dans [Ste90]).

2.1.4 Généralisations

Les problématiques abordées ici peuvent être généralisées de très nombreuses façons.
— L'interprétation est une phase ultime puisqu'elle réalise une exécution, en modifiant l'environnement du programme ou retournant une valeur. En revanche, la compilation peut se comprendre comme n'importe quelle transformation de code d'un langage source à un langage cible.

— Au niveau du langage source, qui peut être un langage de programmation de haut niveau, au sens habituel du terme, mais aussi un langage de bas niveau (par exemple le langage d'une machine virtuelle, *bytecode* JAVA ou .NET), ou des objets plus mathématiques comme les automates [2] ;
— Au niveau du langage cible de la compilation, qui peut ne pas être un langage machine. On exige en général que le langage source soit plus « évolué » que le langage cible. On compile souvent d'un langage évolué vers C, qui peut être considéré comme un assembleur évolué. Cependant, il existe des compilateurs de *bytecode* vers C, et vice et versa.
— Les langages considérés sont en général des langages « de programmation », mais les transformations de documents XML sont relativement du même ordre, bien qu'elles soient généralement très superficielles. De façon générale, on peut aussi transformer les programmes (ou documents) pour changer leur syntaxe superficielle : on peut ainsi passer du style de LISP (complètement parenthésé), aux *markup languages* à la HTML ou XML (balises avec chevrons), ou à la syntaxe TEXet LATEX(macros préfixées par "\" et accolades) sans changer aucunement la structure : la transformation est complètement réversible, ce qui n'est en général pas le cas de la compilation.

2.1.5 Choisir entre compilation et interprétation

Compilation et interprétation représentent les deux termes d'une alternative que l'on rencontre très souvent. On est donc amené à choisir.

Choix du programmeur

Ce choix peut s'adresser à l'utilisateur, c'est-à-dire au programmeur, mais c'est peu courant. En effet, on dispose rarement d'un interprète et d'un compilateur pour le même langage. Si c'est le cas, par exemple dans l'implémentation CLISP de COMMON LISP utilisée dans ce cours, on se contentera de l'interprète pour le code en phase de développement. La compilation n'est alors utile que pour les parties du code qui ont atteint un état stable et dont l'efficacité est critique.

On notera néanmoins le cas des *récursions terminales* (cf. polycopié de LISP), qui sont optimisées en compilé mais pas en interprété, et peuvent nécessiter une compilation précoce.

Choix du concepteur ou implémenteur

Le choix compilation-interprétation s'impose au concepteur de langage (au sens large) ou à l'implémenteur d'un langage sur une plate-forme particulière.

Dans ce choix, plusieurs éléments interviennent :
— les coûts de développement : un interprète est beaucoup plus facile et rapide à réaliser qu'un compilateur ;
— l'efficacité des programmes à l'exécution : un compilateur produit un code dont l'exécution est beaucoup plus efficace que l'interprétation du code source ;
— le nombre de programmes concernés ;
— le caractère dynamique des programmes concernés : si de nouveaux programmes peuvent arriver de façon inattendue au cours de l'exécution d'un programme principal, l'interprétation est de très loin le plus facile ; il faudrait sinon avoir recours à une compilation dynamique autrement plus difficile ;
— le nombre d'exécution de ces programmes.

Si l'on vise un unique programme qui doit être exécuté une unique fois (par exemple pour calculer les n premières décimales de π), les besoins en optimisation ne dépassent pas le seuil nécessaire à l'obtention d'un résultat en un temps raisonnable.

2.2 Langages considérés

Interprètes et compilateurs sont des programmes, écrits dans un certain langage de programmation, destinés à un langage de programmation source, pour produire (pour le compilateur seulement) un programme

2. Rappelons que les automates sont équivalents à des langages, au sens où ils permettent de décider si un mot appartient à un langage ou pas. Cependant, le langage est ici du côté de la donnée : le programme lui-même est constitué par un automate.

dans un langage de programmation cible.

2.2.1 Formalisation

Soit I, J et K trois langages de programmation. On pourra noter $Int_I(J)$ un interprète du langage J écrit dans le langage I. On notera de même $Comp_I(J, K)$ un compilateur, écrit dans le langage I, du langage J vers le langage K. Une contrainte naturelle est que $J \neq K$, car $Comp_I(J, J)$ serait une implémentation de la fonction identité. On peut néanmoins considérer le cas où $K \subset J$: c'est celui des transformations source à source, par exemple les macros LISP.

Composition

Les compilateurs sont des transformations qui peuvent se composer, comme des fonctions :

$$Comp_I(J, K) = Comp_I(K', K) \circ Comp_I(J, K')$$

On obtient des compositions similaires pour les interprètes :

$$Int_I(J) = Int_I(K) \circ Comp_I(J, K)$$

On a traité ici le cas où les deux compilateurs composés sont écrits dans le même langage : ce n'est bien sûr pas une nécessité. Par exemple, la compilation du langage PRM se décrit comme

$$Comp(\text{PRM}, \texttt{x86}) = Comp_{\text{C}}(\text{C}, \texttt{x86}) \circ Comp_{\text{PRM}}(\text{PRM}, \text{C})$$

où $Comp_{\text{PRM}}(\text{PRM}, \text{C})$ est un compilateur de PRM vers C écrit en PRM, et $Comp_{\text{C}}(\text{C}, \texttt{x86})$ est `gcc`. Mais notre notation n'est pas capable d'indicer le terme composé.

De son côté, l'exécution de JAVA repose, à l'origine, sur la composition

$$Int(\text{JVM}) \circ Comp(\text{JAVA}, \text{JVM})$$

d'un compilateur vers le bytecode d'une machine virtuelle et d'un interprète de cette machine virtuelle, les deux pouvant être écrits dans des langages quelconques.

Critère d'opérationnalité

Pour qu'un programme $Prog_I$ écrit dans le langage I soit opérationnel, il faut que le langage I soit opérationnel, c'est-à-dire qu'il existe un interprète $Int_{I'}(I)$ ou un compilateur $Comp_{I'}(I, K)$ qui soient opérationnels, et dans ce dernier cas, que K soit aussi opérationnel.

Il n'y a pas de contrainte particulière entre I, J et K pour que $Int_I(J)$ ou $Comp_I(J, K)$ soient opérationnels. Cependant, le cas de $I = J$ présente un problème de circularité redoutable : pour que $J = I$ soit opérationnel, il faut que I le soit. C'est un poblème classique connu sous le nom d'*amorçage* (ou *bootstrap*), qui se résout par le fait que I peut avoir été rendu opérationnel par un autre moyen, c'est-à-dire au moyen d'un interprète ou compilateur écrit dans un autre langage I'. Un compilateur $Comp_I(I, K)$ est dit *autogène* (ou *bootstrappé*).

De plus, pour qu'un compilateur $Comp_I(J, K)$ produise du code opérationnel, il faut que K soit un langage machine ou qu'il existe un interprète $Int_{I'}(K)$, ou un compilateur $Comp_{I'}(K, K')$, du langage K, qui soient opérationnels. De même, pour qu'un interprète $Int_I(J)$ soit opérationnel, il faut qu'il existe un interprète $Int_{I'}(I)$ ou un compilateur $Comp_{I'}(I, K)$ qui soient opérationnels.

2.2.2 Choix pratiques

L'objectif de ce cours est d'arriver à des programmes $Int_I(J)$ et $Comp_I(J, K)$ qui soient opérationnels.

Pour ce faire, on élimine le cas où K est un langage machine existant, pour sa trop grande complexité. On va donc considérer que $K = \texttt{VM}$ est le langage d'une machine virtuelle pour laquelle on développera un interprète $Int_I(\texttt{VM})$. Pour simplifier les étapes d'analyse lexico-syntaxiques, on prendra LISP comme

langage source et langage de développement : $I = J =$ LISP. Il nous faut donc développer $Int_{\text{LISP}}(\texttt{VM})$ ainsi que $Comp_{\text{LISP}}(\text{LISP}, \texttt{VM})$ et $Int_{\text{LISP}}(\text{LISP})$.

Enfin, comme LISP reste encore un peu compliqué malgré sa grande simplicité, ou pourra aussi développer un langage intermédiaire, LI, très proche de LISP mais permettant une certaine factorisation. Une alternative reviendra donc à développer $Int_{\text{LISP}}(\texttt{VM})$, $Comp_{\text{LISP}}(\text{LISP}, \texttt{LI})$, $Comp_{\text{LISP}}(\texttt{LI}, \texttt{VM})$ et $Int_{\text{LISP}}(\texttt{LI})$.

$$Comp_{\text{LISP}}(\text{LISP}, \texttt{VM}) = Comp_{\text{LISP}}(\texttt{LI}, \texttt{VM}) \circ Comp_{\text{LISP}}(\text{LISP}, \texttt{LI})$$
$$Int_{\text{LISP}}(\text{LISP}) = Int_{\text{LISP}}(\texttt{LI}) \circ Comp_{\text{LISP}}(\text{LISP}, \texttt{LI})$$

2.2.3 Langages imbriqués vs langages linéaires

Un critère simple permet de partionner les différents langages considérés en deux classes :
— dans les *langages imbriqués*, comme LISP ou LI, les expressions ou instructions peuvent être composées : ces langages ont une structure d'arbre ;
— en revanche, dans les *langages linéaires* comme ceux des machines virtuelles, les instructions ne sont jamais composées : ces langages ont une structure de liste.

Sauf rare exception, les langages évolués sont tous imbriqués, et les langages de machine virtuelle sont tous linéaires. Cela n'empêche pas, néanmoins, de compiler des langages évolués vers des langages imbriqués, C par exemple, mais l'utilisation qui en est faite est souvent très linéaire.

2.3 Plan du cours

Dans ce cours, on va s'intéresser, de façon variée, aux notions :
— de compilation dans sa phase plus spécifique de génération de code,
— d'interprétation dans sa phase plus spécifique d'évaluation,
— de transformation de programmes entre langages de structure relativement similaire.

La génération de code et l'interprétation vont se faire dans différents contextes :
— compilation d'un langage évolué dans une machine abstraite ;
— évaluation d'un langage évolué dans un autre langage évolué ;
— évaluation d'une machine abstraite dans un langage évolué ;
— évaluation d'automates et compilation dans un langage évolué ;

Les analyses lexicale et syntaxique qui représentent un élément-clé de la problématique (souvent les cours de compilation ne traitent de rien d'autre) ne seront pourtant pas considérées. En effet, ces analyses sont abordées dans les cours de Licence sur la théorie des langages et des automates et cette UE de Master ne peut pas tout traiter. Par ailleurs, nous travaillerons avec un langage qui nous offre gratuitement ces analyses.

Le langage évolué qui servira de support de cours est en effet le langage COMMON LISP, dialecte de LISP et cousin de SCHEME. Ce choix s'explique par les spécificités de LISP :
— c'est un langage interprété qui a l'originalité de représenter son code dans ses propres structures de données, ce qui rend aisé l'écriture d'un évaluateur LISP en LISP (que l'on appelle un méta-évaluateur) ;
— en tant que langage interprété, il dispose d'une fonction READ qui retourne la structure syntaxique de ses programmes, ce qui permet de faire, momentanément du moins, abstraction des problèmes d'analyse lexicale et syntaxique.

2.4 Bibliographie commentée

Bibliographie générale

Quelques références classiques :
— sur la compilation : [ASU89],
— sur l'initiation à la programmation par le langage SCHEME (cousin de LISP) : [ASS89] ;

— de haut niveau, sur LISP et SCHEME, et les problématiques d'évaluation et de compilation : [SJ93, Que94] ;
— sur LEX et YACC [LMB94] ;
— enfin, [Kar97] est un polycopié intéressant sur la compilation et l'interprétation (dont l'approche très différente du présent cours pourra déconcerter certains).

Bibliographie sur les langages

langage	référence	autres	`Web`
COMMON LISP	[Ste90]		`file:/net/local/doc/cltl/clm/clm.html` `http://clisp.cons.org/`

On se reportera aussi au polycopié de LISP distribué en cours [Duc13a].

3

Interprétation et compilation d'automates

Un automate est assimilable à un programme qui prend en entrée un mot et retourne vrai ou faux suivant que l'automate reconnaît le mot ou non. La compilation et l'interprétation s'appliquent donc parfaitement. Deux versions sont à prévoir suivant que l'automate est déterministe ou pas.

On peut aussi mettre en correspondance les étapes générales de l'interprétation ou de la compilation avec le cas particulier des automates. Ainsi,

— la phase d'analyse lexicale et syntaxique se traduit ici par la construction d'une structure de données automate,

— des transformations source à source peuvent être réalisées, par exemple pour déterminiser un automate, ou pour éliminer les ϵ-transitions.

La problématique est complètement générale mais on se placera dans la suite dans un contexte où le langage d'interprétation ou de cible de compilation est LISP.

3.1 Structure de données automate

Dans tous les cas, il faut définir une structure de données automate qui peut être spécifiée par son interface fonctionnelle :

(AUTO-STATES `auto`) → ***`liste d'états`***
: Retourne la liste des états de l'automate argument : un état sera par exemple un entier, ou un symbole.

(AUTO-TRANSITIONS `auto state`) → ***`liste de transitions`***
: Retourne la listes des transitions issues d'un état, par exemple sous forme de *liste d'association*, où une transition est une paire caractère-état ; dans le cas d'un automate non-déterministe, la transition sera une paire caractère-liste d'états. Pour simplifier, on traitera les caractères comme des symboles.

(AUTO-INIT `auto`) → ***`état`***
: Retourne l'état initial de l'automate argument.

(AUTO-FINAL-P `auto etat`) → **`t / nil`**
: Retourne vrai si l'état est final.

Cette structure de données peut servir aussi bien pour l'interprétation que pour la compilation. Bien entendu, elle ne doit plus être utilisée dans le code généré par la compilation d'un automate.

3.2 Interprétation d'un automate

Il s'agit de définir un programme — ou une fonction `eval-auto` — qui prend en paramètres un automate et un mot et qui retourne vrai ou faux suivant que le mot est reconnu par l'automate ou pas. Le principe de cette fonction repose sur une *analyse par cas* (voir aussi Chapitre 6) de l'automate, chaque

cas se décomposant dans une analyse par cas du mot. (Noter que l'inverse serait vraisemblablement faux.) On retrouve donc ici la double analyse par cas dont il a été question dans les définitions générales de la problématique (Section 2.1).

EXERCICE 3.1. Définir la fonction `eval-auto-d` qui prend en argument un automate *déterministe* et un mot et retourne vrai/faux suivant que l'automate reconnaît le mot ou pas. □

EXERCICE 3.2. Définir la fonction `eval-auto-nd`, similaire à la précédente, sauf qu'elle accepte des automates *non déterministes*. □

3.3 Compilation d'un automate

Il s'agit dans ce cas de générer, à partir d'un automate donné, un programme — par exemple une fonction LISP — qui prend en paramètre un mot et qui retourne vrai ou faux suivant que le mot est reconnu par l'automate ou pas.

3.3.1 Dans un langage évolué

Il faut donc définir une fonction `compile-auto` à un paramètre, l'automate, qui retourne une fonction à un paramètre, le mot, telle que, pour tout automate et tout mot. En LISP, on aura alors l'équivalence :

```
(eval-auto auto mot) ≡ (apply (compile-auto auto) mot ())
```

Cette équivalence est juste la formulation LISP de la composition telle qu'elle est définie au chapitre précédent :

$$Int_{\text{LISP}}(\texttt{Auto}) = Int(\text{LISP}) \circ Comp_{\text{LISP}}(\texttt{Auto}, \text{LISP})$$

où `apply` représente l'interprète "natif" de LISP, $Int(\text{LISP})$, et `Auto` représente le langage des automates.

EXERCICE 3.3. Définir un petit automate simple, par exemple celui qui correspond à l'expression régulière `ab*a`, et écrire à la main la fonction LISP qui prend en paramètre un mot et qui retourne vrai ou faux suivant que le mot est reconnu par l'automate ou pas. □

EXERCICE 3.4. Définir la fonction `compile-auto-d` qui prend en argument un automate déterministe et retourne une fonction d'interprétation de mot, qui prend en argument un mot et retourne vrai/faux suivant que l'automate reconnaît le mot ou pas. □

EXERCICE 3.5. Définir la fonction `compile-auto-nd`, similaire à la précédente, sauf qu'elle accepte des automates *non déterministes*. □

EXERCICE 3.6. Redéfinir la fonction `compile-auto-d` sous la forme d'une macro. □

Remarque 3.1. La démarche de compilation est générale : avant d'écrire la fonction qui génère le code cible, il faut être capable d'écrire à la main le code cible.

Pour cela il est d'abord nécessaire de répondre à une première question : comment peut-on représenter un état en LISP ? Une fois cette question résolue, le reste n'est plus qu'affaire de technique.

Remarque 3.2. Pour bien juger de la réussite de l'exercice il faut vérifier deux conditions :

— la sémantique est vérifiée, donc les mots reconnus sont exactement les mots du langage ; mais cette première condition est vérifiée aussi par `eval-auto` ;

— la compilation doit être complète, donc il ne reste plus aucune analyse par cas basée sur la syntaxe de l'automate : les tests restant dans la fonction obtenue ne doivent porter que sa donnée, le mot.

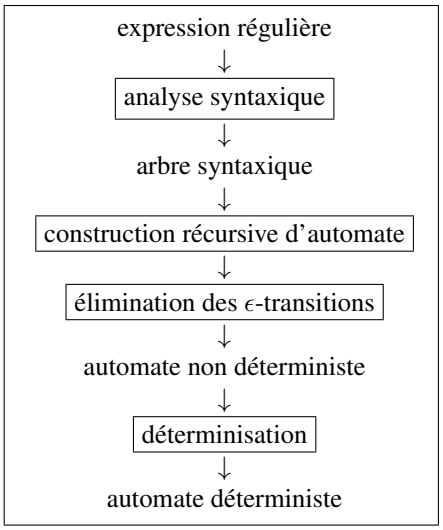


FIGURE 3.1 – Construction d'automate à partir d'une expression régulière

3.3.2 Dans un langage machine

On peut directement compiler dans un langage machine, par exemple dans une machine virtuelle. On aurait alors une équivalence approximative :

```
(vm-compile-auto auto) ≈ (vm-compile (compile-auto auto))
```

qui correspond exactement à la composition

$$Comp_{\text{LISP}}(\texttt{Auto}, \texttt{VM}) = Comp_{\text{LISP}}(\text{LISP}, \texttt{VM}) \circ Comp_{\text{LISP}}(\texttt{Auto}, \text{LISP})$$

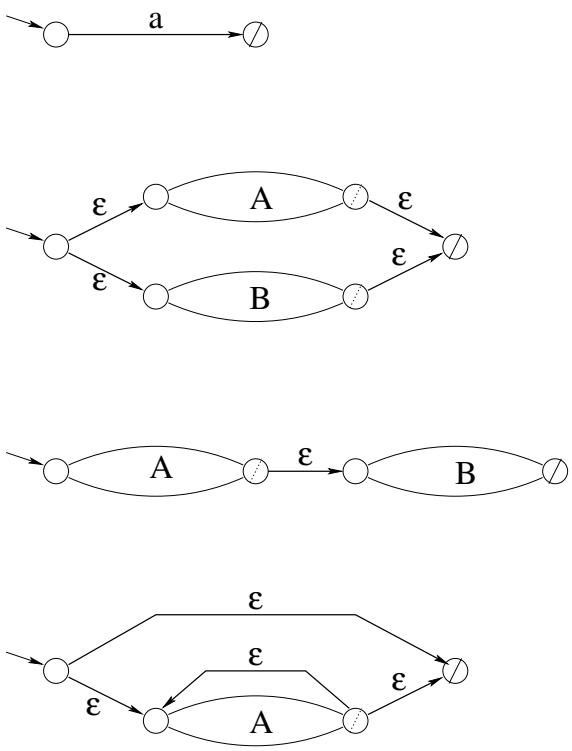
On répondra donc aux questions de la section précédente en considérant comme langage cible le langage de la machine virtuelle (Chapitre 8). Une fois encore, la question principale sera de décider de la façon de représenter un état dans la machine virtuelle.

3.4 Choix entre interprétation et compilation

Dans le chapitre précédent, on a discuté de façon abstraite du choix entre compilation et interprétation. Le cas des automates permet de reprendre cette discussion de façon beaucoup plus concrète. Plusiezurs cas peuvent se présenter :

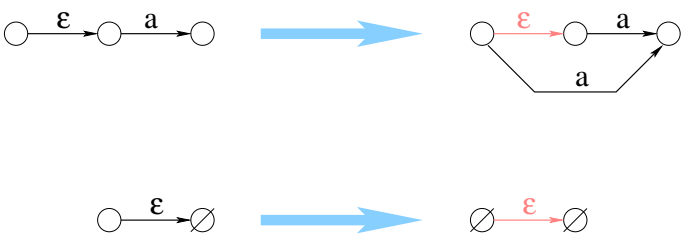
1. à une extrême, on a besoin d'un unique automate pour une application particulière : le plus simple est certainement de le compiler "à la main", c'est-à-dire de l'écrire directement dans le langage cible, ou dans un langage plus évolué qui sera compilé vers le langage cible ;
2. à l'autre extrême, l'application considérée peut charger à la volée des automates : le plus raisonnable est de développer un interprète d'automates ;
3. l'application considérée repose sur un ensemble prédéfini de plusieurs automates et nécessite une certaine efficacité : un compilateur s'impose.

De façon générale, on peut évaluer les coûts de chacune des approches :
— coût de compilation "à la main" de chaque automate ;
— coût de développement d'un interprète ou d'un compilateur ;
— coût de l'inefficacité du programme utilisant un interprète (ce dernier coût n'étant pas comparable, il risque d'être incertain).



Les quatre automates font l'analyse par cas des expressions régulières, de haut en bas : symbole (`a`), addition (`A+B`), concaténation (`AB`) et répétition (`A*`). Un invariant de la construction est que l'état final est unique.

FIGURE 3.2 – La construction récursive d'automates



Les motifs de gauche se réécrivent comme motifs de droite, où les transitions en grisé sont éliminées, ainsi que les états qui deviendraient inaccessibles.

FIGURE 3.3 – Elimination des ϵ-transitions

3.5 Construction d'un automate à partir d'une expression régulière

Si l'automate est donné sous forme d'expression régulière, il faut commencer par générer la structure de données d'un automate équivalent à l'expression.

La figure 3.1 décrit les étapes de constructions d'un automate à partir d'une expression régulière.

La figure 3.2 décrit les quatre cas de construction récursive d'un automate suivant l'expression régulière. L'invariant de la construction est que

— chaque automate a exactement un état initial
— et un état final
— et que l'état initial n'a pas de transition entrante,
— et que l'état final n'a pas de transition sortante.

L'élimination des ϵ-transitions consiste en la fermeture del'automate par les deux opérations d'augmentation et de propagation des terminaux (figure 3.3), suivies de l'enlèvement des ϵ-transitions et des parties inaccessibles.

3.6 En savoir plus

Les automates ont beaucoup à voir avec la compilation et l'interprétation. C'est d'abord la base de l'analyse lexicale. La construction des environnements nécessaires au passage de paramètres (Exercice 4.5) et le filtrage (Section 4.7.1), qui en est une généralisation, sont aussi des formes d'automates.

4

Un méta-évaluateur naïf pour COMMON LISP

Ce chapitre a pour objectif de définir, en COMMON LISP, un évaluateur du langage COMMON LISP écrit en COMMON LISP : on appelle cela un *méta-évaluateur*. Les motivations d'un tel méta-évaluateur sont multiples :

1. c'est d'abord un exercice en vraie grandeur de programmation en LISP ;
2. c'est ensuite une façon formelle de spécifier la "sémantique" du langage LISP (même si c'est surtout la sémantique d'**un** dialecte LISP, pas forcément celui qui est visé) ;
3. c'est donc un bon moyen pour comprendre le langage ;
4. c'est aussi un outil pour comparer divers dialectes ;
5. c'est un moyen pratique de mettre en œuvre des constructions qui n'existent pas dans le langage (ou le dialecte considéré) LISP ;
6. c'est enfin une première étape dans la réalisation d'un véritable interprète LISP : il "suffit" de compiler ce méta-évaluateur dans le langage cible choisi (par exemple la JVM) pour obtenir 80% d'un véritable interprète.

Les références [Que94, SJ93] sont recommandées *pour un approfondissement ultérieur* mais le poly-copié de LISP [Duc13a] doit être consulté en parallèle avec ce chapitre.

4.1 Sur le sens de méta

La notion de « méta » (méta-niveau, méta-modèle, méta-programmation) est devenu essentielle en informatique, en particulier en génie logiciel et en modélisation et programmation par objets (de UML à CLOS en passant par JAVA). Cette notion sera développée dans l'UE GMIN310, *Méta-programmation et réflexivité*, de S3. Nous en présentons juste l'origine et une utilisation très commune.

Aristote et la métaphysique.

> Rappelons que *meta* est une préposition grecque ($\mu\epsilon\tau\grave{\alpha}$) qui signifie « avec » ou « après » selon qu'elle gouverne le génitif ou l'accusatif.
> Le sens qu'elle prend dans « métaphysique » vient de ce que, dans l'ordre traditionnel des œuvres d'Aristote, les livres qui traitaient de la « philosophie première » venaient après les livres de physique — $\mu\epsilon\tau\grave{\alpha}\ \tau\grave{\alpha}\ \phi\upsilon\sigma\iota\kappa\acute{\alpha}\ \beta\iota\beta\lambda\iota\acute{\alpha}$.
>
> J. F. Perrot, in [DEMN98]

Le sens et l'origine du mot méta-physique convergent ainsi vers un au-delà de la physique qui n'est qu'une coïncidence car elle résulte d'un contre-sens.

Méta-langage. Le terme de « méta-langage » désigne de façon usuelle, aussi bien en linguistique qu'en logique, un langage dont l'objet est un autre langage [1].

Ainsi, la logique distingue classiquement (voir [Kle71]) :

— son langage, par exemple celui de la logique des propositions dont les formules « $A \wedge B$ » et « $A \rightarrow B$ » font partie ;

— les méta-langages qui permettent de parler de ces formules, qu'ils soient informels, par exemple « la formule $A \wedge B$ est vraie », ou formels, par exemple la formule du *modus ponens* dans le formalisme de la déduction naturelle :

$$\frac{A \qquad A \rightarrow B}{B} \tag{4.1}$$

qui signifie, dans ce formalisme, que si l'on a A et $A \rightarrow B$, alors on a B.

Réflexivité et *bootstrap*.

> Car, pour forger, il faut un marteau, et pour avoir un marteau, il faut le fabriquer.
> Ce pourquoi on a besoin d'un autre marteau et d'autres outils,
> et pour les posséder, il faut encore d'autres instruments, et ainsi infiniment.
> Spinoza, *De la réforme de l'entendement*, 1661.

Le méta mène à la notion de *bootstrap* (ou *amorçage*), omniprésente en informatique (chaque fois que vous allumez votre ordinateur), en particulier dans le cadre de la compilation. Car la question se pose : comment compiler un compilateur ou interpréter un interpréteur ?

4.2 Principe de la méta-évaluation

L'interprète LISP repose sur la boucle `read-eval-print`. L'évaluateur est la fonction `eval`, qui

1. prend en paramètre une expression LISP qui est censée être évaluable,
2. évalue (ou essaie d'évaluer) cette expression,
3. retourne une valeur, à moins qu'une erreur ne provoque une exception.

Pour ne pas interférer avec le système LISP sous-jacent, la fonction qui implémente le méta-évaluateur s'appellera `meval`.

Méta-évaluateur trivial On dispose donc déjà naturellement d'un méta-évaluateur trivial :

```
(defun MEVAL (expr)
   (eval expr))
```

Ce n'est pas simplement une plaisanterie : un méta-évaluateur peut et même doit se reposer sur l'évaluateur sous-jacent. En particulier, le méta-évaluateur ne prétend pas traiter directement des types de base du langage et entend bien se reposer sur l'évaluateur. **Le méta-évaluateur ne se préoccupe que d'évaluation, pas d'implémentation.**

Ce n'est pas non plus toujours possible, par exemple si l'on souhaitait développer un interprète à partir de rien. En pratique, cela signifie aussi que l'on peut toujours comparer ce que retourne `meval` avec ce que retourne `eval`.

Analyse par cas Comme pour tout ce qui repose sur une expression syntaxique, un méta-évaluateur consiste en une *analyse par cas*, de toutes les différentes formes syntaxiques possibles de l'expression en argument. Le corps de la fonction `meval` consiste donc en une structure conditionnelle (`if` imbriqués, `cond`, voire `case`) qui va énumérer tous les cas possibles et traiter spécifiquement chacun. Pour simplifier la lecture, on utilisera un `cond` mais une structure de `if` imbriqués pourrait être plus équilibrée.

Idéalement, le squelette du méta-évaluateur pourrait donc avoir la structure suivante :

1. Le lecteur avisé aura remarqué que l'ensemble de ce paragraphe relève d'un méta-méta-langage ! Je rajouterais bien que cette note relève, elle, d'un méta-méta-méta... si le vertige ne me prenait.

```
(defun MEVAL (expr)
   (cond (..)                                                    ;  cas 1
         (..)                                                    ;  cas 2
         ...
         (..)                                                    ;  cas n
1        (t (eval expr)))                                  ;  ramasse miettes
```

On traiterait les cas non prévus par appel à `eval`.

La problématique de l'analyse par cas est reprise de façon plus transversale dans le chapitre 6.

Définition incrémentale Avec cette structure, on peut envisager simplement une définition incrémentale de `meval` en rajoutant les cas un par un, et en les **testant au fur et à mesure**.

En pratique, il va souvent s'agir de traiter plusieurs cas simultanément car ils sont très étroitement corrélés : par exemple, l'application d'une fonction et l'évaluation d'une variable, ou encore `function` et `apply`.

La seule difficulté est de placer les clauses successives du `cond` **dans le bon ordre** car les conditions sont rarement disjointes.

Remarque 4.1. Dans l'analyse par cas, les cas seront numérotés et les numéros seront rappelés au fur et à mesure, comme repère pour l'ordre des cas ou pour marquer leurs modifications ultérieures.

Définition récursive La définition de `meval` est enfin forcément récursive, puisque dans les cas les plus généraux, l'évaluation d'une expression va reposer sur l'évaluation récursive de ses sous-expressions. C'est particulièrement le cas pour l'application d'une 'vraie' fonction, qui commence par l'évaluation des arguments.

Principe et limites Le principe d'un méta-évaluateur est donc le suivant. Par une *analyse par cas*, il décompose l'expression à évaluer en sous-expressions qu'il évalue *récursivement*, jusqu'à tomber sur les cas primitifs — formes spéciales et fonctions prédéfinies — pour lesquels il fait *appel à l'évaluateur sous-jacent*.

Le méta-évaluateur s'intéresse donc à la syntaxe des expressions, dont il exprime la sémantique en les traduisant en une évaluation récursive. Il ne s'intéresse en aucun cas à la *création* des *valeurs* manipulées, qui reste du ressort de la plate-forme d'exécution sous-jacente. Les deux évaluateurs se partagent donc les mêmes valeurs, qui sont créées par l'évaluateur sous-jacent et dont le méta-évaluateur se contente d'organiser la circulation [2].

4.3 Le méta-évaluateur de base

On va donc procéder pas à pas, en traitant d'abord les constantes, les variables et l'application des fonctions, avant de considérer les formes syntaxiques de base. L'objectif est d'arriver à méta-évaluer des fonctions comme Fibonacci.

4.3.1 Évaluation des constantes

L'évaluation la plus simple concerne les constantes, dont on peut distinguer deux cas : les atomes qui ne sont pas des variables, et les expressions *quotées*. COMMON LISP fournit le prédicat `constantp` qui dit si une expression est une constante ou pas, mais les deux cas de constante sont à traiter différemment :

```
(defun MEVAL (expr)
   (cond
2        ((and (atom expr) (constantp expr))              ;  constante atomique
         expr)
3        ((and (consp expr) (eq 'quote (car expr)))                  ;  quote
         (cadr expr))
1        (t (eval expr)))                                  ;  ramasse miettes
```

2. « Puisque ces mystères nous dépassent, feignons d'en être l'organisateur. », Jean Cocteau, les Mariés de la Tour Eiffel.

On peut maintenant faire les évaluation suivantes (colonne de gauche), qui correspondent aux valeurs du paramètre `expr` de la colonne centrale, et qui retournent le résultat de la colonne de droite :

```
(meval 3)      3                    →  3
(meval '3)     3                    →  3
(meval ''3)    (quote 3)            →  3
(meval '''3)   (quote (quote 3))    →  '3
```

Remarque 4.2. Attention aux « ' » ! `read` transforme le caractère « ' » en la liste du symbole `quote` et de l'expression qui suit. La première « ' » est consommée par `eval`, la seconde par `meval` et la troisième est restituée par `print` de façon symétrique à `read`. S'il n'y a aucun « ' », c'est `eval` qui fait tout le boulot !

4.3.2 Évaluation des variables et application des fonctions

Le cœur de l'évaluation LISP repose sur la capacité à appliquer des fonctions et à évaluer des variables. Que faut-il faire pour pouvoir évaluer une expression aussi simple que `((lambda (x) x) 3)` ? Bien que triviale, cette évaluation exprime l'essence du langage.

Pour la traiter, il faut introduire 2 nouveaux cas syntaxiques, pour les variables et les λ-fonctions. On peut aussi ajouter un cas supplémentaire pour éviter qu'une liste évaluable commence par autre chose qu'un symbole ou une λ-fonction :

```
(defun MEVAL (expr)
   (cond
2        ((and (atom expr) (constantp expr))      ;   atome constant = littéral
          expr)
4        ((atom expr)                          ;  atome non constant, donc variable
          ...)
         ;; plus d'atome à partir d'ici
5        ((and (consp (car expr)) (eq 'lambda (caar expr)))       ;  λ-fonction
          ...)
6        ((or (not (symbolp (car expr))) (constantp (car expr)))
                                  ;  une fonction est un symbole non constant
          (error "~ ne peut être une fonction" (car expr)))
3        ((eq 'quote (car expr))                                      ;  quote
          (cadr expr))
1        (t (eval expr)))                                       ;  ramasse miettes
```

Noter que la clause de `quote` (3) se simplifie car l'ajout du test sur les variables a éliminé la possibilité d'avoir un atome à ce stade de l'analyse par cas. L'ordre des premiers cas est impératif et tous les ajouts devront se faire après le traitement de l'erreur.

EXERCICE 4.1. Vérifier ce que fait la fonction `error` dans le poly ou le manuel en ligne. □

Environnement L'évaluation d'une variable et l'application d'une fonction s'articulent autour de la notion d'*environnement*. Pour évaluer `((lambda (x) x) 3)`, on commence par 'construire' l'environnement $\{X \to 3\}$, avant d'évaluer le corps de la λ-fonction dans cet environnement. L'évaluation n'a de sens que *dans un environnement donné*, ce qui signifie que `meval` doit prendre un second paramètre, pour l'environnement. Ce paramètre peut être optionnel (par défaut l'environnement est vide). Il faut ensuite *implémenter* cet environnement, par exemple par une *liste d'association*, c'est-à-dire une liste de *liaisons* variable-valeur, ce qui donnerait `((X . 3))`.

Le squelette de `meval` a alors la structure suivante (on ne rappelle pas les différents cas)

```
(defun MEVAL (expr &optional env)
   (cond ...
1        (t (eval expr))))                                      ;  ramasse miettes
```

et on remarque une dissymétrie : `meval` reçoit un environnement mais ne le passe pas à `eval`. En réalité, `eval` n'a aucun moyen de connaître l'environnement du méta-évaluateur donc il n'est possible de faire appel à `eval` que si l'environnement courant `env` est vide.

```
(defun MEVAL (expr &optional env)
   (cond ...
1        ((null env) (eval expr))                              ;  ramasse miettes
7        (t (error "impossible d'évaluer ~s dans l'environnement ~s"
                   expr env))))
```

Évaluation des variables Une fois que l'environnement est défini, l'évaluation des variables est simple. Si le symbole figure dans l'environnement, l'évaluation doit retourner la valeur qui lui est associée. Sinon, il s'agit d'une *variable libre* dont la tentative d'évaluation doit lever une exception :

```
(defun MEVAL (expr &optional env)
   (cond ...
4        ((atom expr)                  ;  constante non atomique, donc variable
          (let ((cell (assoc expr env)))          ;  cell représente la liaison
             (if cell
                 (cdr cell)
               (error "~s n'est pas une variable" expr))))
         ...))
```

EXERCICE 4.2. Vérifier ce que fait la fonction `assoc` dans le poly ou le manuel en ligne. □

λ-expression L'évaluation d'une λ-expression consiste à appliquer la λ-fonction aux arguments. Cette application n'est pas beaucoup plus compliquée mais elle mais en jeu des fonctions annexes. Le principe est d'évaluer le corps de la fonction, un `progn` implicite, dans l'environnement créé par l'appariement des paramètres et des arguments, préalablement évalués.

```
(defun MEVAL (expr &optional env)
   (cond ...
5        ((and (consp (car expr)) (eq 'lambda (caar expr)))       ;  λ-fonction
          (meval-body (cddar expr)
                      (make-env (cadar expr)
                                (meval-args (cdr expr) env)
                                env)))
         ...))
```

EXERCICE 4.3. Définir la fonction `meval-body` qui prend en paramètre une liste d'expressions évaluables et un environnement, qui les évalue en séquence et retourne la valeur retournée par la dernière. □

EXERCICE 4.4. Définir la fonction `meval-args` qui prend en paramètre une liste d'expressions évaluables et un environnement, qui les évalue en séquence et retourne la liste de leurs valeurs. □

EXERCICE 4.5. Définir la fonction `make-env` qui prend en paramètre une liste de symboles, une liste de valeurs et un environnement : construit l'environnement (une liste d'association) en appariant les paramètres aux valeurs correspondantes et signale une exception si paramètres et arguments ne concordent pas. On ne traitera d'abord que le cas des paramètres obligatoires. Si l'environnement passé en paramètre n'est pas vide, le nouvel environnement doit l'inclure. □

Définition et application de fonctions globales Il faut ensuite traiter le cas des fonctions globales, où il faut distinguer les fonctions définies par `defun` pour le méta-évaluateur des fonctions définies par `defun` pour l'évaluateur de base. Il est nécessaire de distinguer ces deux cas parce que les deux évaluateurs ne peuvent pas se partager ces définitions, de même qu'ils ne savent pas se partager les environnements. Les premières seront dites *méta-définies* et les secondes *prédéfinies*.

Les fonctions globales imposent de traiter le cas de `defun` ainsi que celui de l'application d'une fonction définie par `defun`. Le rôle de `defun` est d'associer une valeur fonctionnelle, en première approximation une λ-fonction, au nom de la fonction. L'application consiste alors à appliquer cette λ-fonction, d'une façon similaire au cas de la λ-expression.

On supposera donc que l'on dispose d'une fonction `get-defun` permettant d'accéder à la λ-fonction associée à un symbole, et que cette fonction est `setf`-*able*.

```
(defun MEVAL (expr &optional env)
   (cond ...
6        ...                              ; une fonction est un symbole non constant
8        ((get-defun (car expr))                                    ;  fonction globale
          (let ((fun (get-defun (car expr))))
               (meval-body (cddr fun)
                           (make-env (cadr fun)
                                     (meval-args (cdr expr) env)
                                     ()))))
9        ((eq 'defun (car expr))                                                ;  defun
          (setf (get-defun (cadr expr))
                `(lambda ,@(cddr expr))))
3                                                                               ; quote
         ...))
```

On placera l'application de la fonction au premier endroit où l'on est assuré d'avoir un symbole en position fonctionnelle, donc juste après (6), et on regroupera `defun` avec les autres formes syntaxiques, par exemple `quote` (3).

Remarque 4.3. L'application de la fonction globale est identique à celle de la λ-fonction dans la λ-expression, à un détail crucial près : dans le premier cas, l'environnement construit n'inclut pas l'environnement d'appel, contrairement au premier : cela se traduit par l'argument `nil` ou `env` dans l'appel de `make-env`.

On peut abstraire ces deux applications de fonctions par une fonction `meval-lambda` qui applique une λ-fonction quelconque à des valeurs d'arguments dans un certain environnement. Cette fonction servira aussi pour les autres cas d'application de fonction, par exemple pour les macros. On obtient alors :

```
(defun MEVAL (expr &optional env)
   (cond ...
5        ((and (consp (car expr)) (eq 'lambda (caar expr)))       ;  λ-fonction
          (meval-lambda (car expr) (meval-args (cdr expr) env) env))
6        ...                              ; une fonction est un symbole non constant
8        ((get-defun (car expr))                                    ;  fonction globale
          (meval-lambda (get-defun (car expr)) (meval-args (cdr expr) env) ()))
         ...))
```

L'analogie et les différences apparaissent ainsi clairement.

EXERCICE 4.6. Définir cette fonction `meval-lambda`. □

Il reste enfin à définir la fonction `get-defun`. On pourrait construire un environnement spécial — il s'agit bien d'ailleurs d'un environnement spécial, réservé aux fonctions et global — en réutilisant des listes d'association, mais cela poserait divers problèmes techniques et le plus simple est d'utiliser les *propriétés* des *symboles*, et la fonction `get`. On définira alors `get-defun` ainsi :

```
(defun GET-DEFUN (symb)
    (get symb :defun))
```

où `symb` est le symbole, c'est-à-dire le nom de fonction, concerné et `:defun` est un *keyword*, c'est-à-dire un symbole constant arbitraire. Cependant, si `get` est bien `setf`-able, ce n'est plus le cas de `get-defun`.

EXERCICE 4.7. Vérifier ce que fait la fonction `get` dans le poly ou le manuel en ligne. □

EXERCICE 4.8. Écrire `get-defun` sous forme de macro et vérifier que cette nouvelle version est bien `setf`-able. □

Fonctions prédéfinies La fonction `meval` est déjà assez sophistiquée mais elle ne permet pourtant pas d'effectuer un calcul non trivial. En effet, on ne sait toujours pas appeler les fonctions prédéfinies, qui sont nécessaires à la manipulation de tous les types du langage.

Il faut donc prévoir un cas supplémentaire pour traiter toutes ces fonctions et la structure de l'analyse de cas va être légèrement modifiée, car il faut distinguer les vraies fonctions des formes spéciales. On va donc commencer par vérifier que le symbole a bien une définition fonctionnelle, par le prédicat `fboundp`, puis éliminer les formes spéciales, par `special-form-p` :

```
(defun MEVAL (expr &optional env)
   (cond ...
3        ...                                                    ; quote
10       ((not (fboundp (car expr)))
          (error "~s symbole sans définiton fonctionnelle" (car expr)))
         ((special-form-p (car expr))                ; forme spéciale non traitée
          (if (null env)
1             (eval expr)
7           (error "~s forme spéciale NYI" (car expr))))
11       (t                                                  ; fonction globale
          (apply (symbol-function (car expr)) (meval-args (cdr expr) env)))
         ))
```

La fonction `symbol-function` de COMMON LISP retourne la valeur fonctionnelle associée à un symbole, qu'il suffit d'appliquer, par `apply`, à la liste des valeurs d'arguments. Une exception est levée si le symbole n'a aucune définition fonctionnelle. Il faudra ensuite traiter le cas des macros, mais on verra cela plus tard.

```
(meval 'x '((x . 3)))          → 3
(meval 'x '((y . 4)))          → X n'est pas une variable
(meval '(+ 1 2) ())            → 3
(meval '(+ x 2) '((x . 2)))    → 4
(meval '(1 2 3) ()             → 1 ne peut être une fonction
```

EXERCICE 4.9. Vérifier ce que fait la fonction `symbol-function` dans le poly ou le manuel en ligne. □

EXERCICE 4.10. Vérifier ce que fait la fonction `special-form-p` dans le poly ou le manuel en ligne. □

EXERCICE 4.11. Vérifier ce que fait la fonction `fboundp` dans le poly ou le manuel en ligne. □

EXERCICE 4.12. Tester ces 3 fonctions sur des arguments de différents types : autre que `symbol`, sur des symboles avec ou sans définition fonctionnelle, et enfin avec des définitions fonctionnelles de différents types (formes syntaxiques, macros, fonctions globales ou locales). □

4.3.3 Structures de contrôle

Il manque encore les structures de contrôle pour pouvoir faire de vrais calculs. La définition de la conditionnelle est immédiate, mais elle paraît assez mystérieuse au premier abord :

```
(defun MEVAL (expr &optional env)
   (cond ...
3        ...                                                    ; quote
12       ((eq 'if (car expr))                                     ; if
          (if (meval (cadr expr) env)            ; si la condition est vérifiée
              (meval (caddr expr) env)             ; évaluer le second argument
            (meval (cadddr expr) env)))            ; sinon évaluer le troisième
10       ...
         ))
```

La conditionnelle se méta-évalue en LISP aussi simplement qu'elle se décrit en français, mais il est légitime de s'interroger. En fait, le `if` de `meval` s'implémente en faisant appel à celui de `eval`. Si l'on imagine une chaîne d'évaluateurs, $\texttt{meval}_i$, avec $\texttt{meval}_0$ =`eval` et une expression de la forme `(`$\texttt{meval}_k$ `'(`$\texttt{meval}_{k-1}$ `... '(`$\texttt{meval}_1$ `'(if ..)))))`, l'évaluateur $\texttt{meval}_k$ va implémenter `if` en faisant appel à meval_{k-1}, qui fera de même jusqu'à ce que $\texttt{meval}_0$ =`eval` fasse le test effectif et résolve la conditionnelle.

4.3.4 Mise en œuvre et premiers tests

Il est maintenant possible de méta-évaluer des calculs non triviaux, comme les fonctions factorielle et Fibonacci.

Remarque 4.4. D'un point de vue méthodologique, il est très difficile de distinguer les calculs effectués par le méta-évaluateur de ceux que l'évaluateur sous-jacent a effectués. Lorsque qu'une fonction `foo` a été définie pour `eval`, sa méta-évaluation va marcher trivialement si la fonction n'a pas été aussi définie pour `meval`. Il est donc conseillé de ne jamais définir les fonctions de test pour `eval`, mais ce n'est pas toujours possible.

Le seul critère net qui permette de distinguer `eval` de `meval` est l'efficacité : les temps d'exécution doivent être dans un rapport de 5 à 10. Aussi, il est recommandé de toujours mesurer le temps et l'espace d'une méta-évaluation avec la macro `time` : tout appel du méta-évaluateur doit être de la forme `(time (meval '<expr>))`, dont il faut comparer le résultat à `(time <expr>)`. Attention à ne pas oublier la « ' » (cf. Remarque 4.2) ! Il suffit de l'oublier pour tester `eval` et non `meval`.

EXERCICE 4.13. Méta-définir les fonctions `fact` et `fibo`. Les tester. □

4.4 Méta-évaluation du méta-évaluateur

L'étape suivante consiste à couvrir "suffisamment" le langage COMMON LISP. Comment formaliser ce "suffisamment" ? Par le fait que le méta-évaluateur soit capable d'évaluer un programme LISP d'une complexité significative. Lorsque l'on se pose ce genre de questions, on dispose en général d'un seul programme, celui que l'on veut tester.

4.4.1 `Meval` de `meval`

L'objectif est donc maintenant de couvrir ce qu'il faut du langage pour pouvoir méta-évaluer de façon non triviale le méta-évaluateur : par exemple, faire `(meval '(meval '(fibo 10)))` ou `(meval '(meval '(meval '(fibo 10))))`.

Pour réaliser cela, il est indispensable de traiter toutes les constructions syntaxiques qui figurent dans le code de la fonction `meval` ou des fonctions annexes. Une façon de faire consiste à essayer d'évaluer `(meval '(meval '(fibo 10)))` : cela va provoquer une erreur, il faudra diagnostiquer sa cause, implémenter la construction qui l'a provoquée, et recommencer.

EXERCICE 4.14. Considérer l'expression `(meval '(meval '(fibo 10)))` et en déduire quelle va être la première erreur produite par son évaluation. Vérifier par un test. Si ça marche du premier coup, c'est mauvais signe : vérifier que `meval` a bien été méta-définie (par `(meval '(defun meval ...))`) ! □

Remarque 4.5. Dans la lignée des remarques précédentes, on n'oubliera pas les « ' » qui doivent être aussi nombreuses que les appels à `meval`.

Une autre façon de faire consiste à parcourir le code de `meval` et des fonctions annexes et à déterminer les constructions qui ne sont pas encore traitées. En parcourant le code dans l'ordre, on rencontre successivement le 'mot-clé' `&optional` — l'exercice 4.5 ne traitait que les paramètres obligatoires —, et les formes syntaxiques ou macros `cond`, `and`, `setf` et `let`.

Remarque 4.6. Si le méta-évaluateur est un critère de couverture du langage, on évitera d'utiliser de trop nombreuses formes spéciales pour ne pas avoir à les implémenter prématurément. En particulier, ne pas utiliser `labels` et les fonctions locales (Section4.5.3).

EXERCICE 4.15. Étendre la définition de `make-env` aux mots-clés `&optional` et `&rest`. On se basera sur le fait que la spécification de ces mots-clés repose sur un automate implicite. Expliciter l'automate et l'implémenter par des fonctions adéquates (voir aussi Chapitre 3, en particulier la section 3.3). □

4.4.2 Fonction de méta-chargement

Avant de chercher à étendre le méta-évaluateur, il faut songer à se doter d'outils pour charger le code. En effet, `(meval '(meval '(fibo 10)))` n'a d'intérêt que si `meval` a été méta-définie. Pour les tests de `fact` ou `fibo`, il suffit de faire `(meval '(defun fact (n) ...))` mais cela devient moins agréable lorsqu'il faut méta-définir tout un ensemble de fonctions : cela ne sert à rien de faire `(meval '(defun meval ...))` si on ne le fait pas aussi sur toutes les fonctions annexes, d'autant que toutes ces fonctions doivent être connues à la fois par `eval` et `meval` (cf. aussi Remarque 4.4).

Il est logique de définir les fonctions du méta-évaluateur dans un unique fichier que l'on charge par `load` pour qu'elles soient connues de `eval`. La solution passe donc par une fonction de *méta-chargement*, `mload`, qui fait l'équivalent de `load` en remplaçant `eval` par `meval`. On rappelle que `load` est une variante sans `print` de la boucle de *toplevel*, dont le squelette a la forme

```
(loop while t
      do
      (eval (read)))
```

mais qui doit aussi faire ce qu'il faut pour gérer les fichiers, canaux d'entrée et fin de fichier.

EXERCICE 4.16. Définir cette fonction `mload` : on regardera dans le manuel le chapitre sur les entrées-sorties, en particulier les fonctions `open`, `read` et `close`, ainsi que le traitement de la fin de fichier. □

Remarque 4.7. Dans la lignée des remarques précédentes, on n'oubliera pas de méta-charger le méta-évaluateur avant de tester `(meval '(meval ...))`.

4.4.3 Les macros

Le principe de l'évaluation des macros est parfaitement analogue à celui des fonctions. Voir la section sur les macros dans le chapitre 3 du polycopié LISP [Duc13a]. On distinguera d'abord le cas des macros *méta-définies*, connues par `meval` des macros prédéfinies qui ne sont connues que par `eval`.

Macros méta-définies Les macros méta-définies sont parfaitement analogues aux fonctions méta-définies, et on placera les deux cas côte-à-côte. On définira d'abord `get-defmacro` comme `get-defun`. Pour l'évaluation, on se rappelle que l'évaluation des macros consiste à faire commuter l'application (`meval-lambda`) et l'évaluation, celle des arguments (`meval-args`) étant remplacée par celle de la valeur retournée (`meval`) : `meval` ∘ `meval-lambda` au lieu de `meval-lambda` ∘ `meval-args`. Cela donne :

```
(defun MEVAL (expr &optional env)
   (cond ...
8        ...                                  ;  fonction globale  méta-définie
13       ((get-defmacro (car expr))                          ;  macro méta-définie
          (meval (meval-lambda (get-defmacro (car expr)) (cdr expr) ())
                 env))
         ...
9                                                                  ; defun
14       ((eq 'defmacro (car expr))                                ;  defmacro
          (setf (get-defmacro (cadr expr))
                `(lambda ,@(cddr expr))))
         ...))
```

Le code qui précède marche bien mais il lui manque le petit rien qui fait tout ; le résultat de l'expansion (`meval-lambda`) doit se substituer à l'expression d'origine. Il faut pour cela intercaler entre `meval-lambda` et `meval` l'appel à une petite fonction qui effectue cette substitution. Cela donne maintenant : `(meval (displace expr (meval-lambda ...)))`.

EXERCICE 4.17. Définir `get-defmacro` comme une macro. □

EXERCICE 4.18. Définir la fonction `displace` qui prend en argument 2 cellules, met dans la première le contenu de la seconde et retourne la première. Rajouter le cas où le résultat de la macro-expansion est un atome. □

Macros prédéfinies Le cas des macros prédéfinies est lui aussi analogue à celui des fonctions prédéfinies mais l'interface fonctionnelle est un peu *ad hoc*. Il faut d'abord se rapporter à la séquence de code qui termine l'analyse par cas, à partir du cas (11), page 25. Pour les macros, le prédicat `fboundp` retourne vrai, `special-form-p` retourne faux et `symbol-function` retourne une valeur fonctionnelle. Donc, si l'on ne fait rien, le code des fonctions prédéfinis s'applique, mais il ne fera pas ce qui est attendu. Le traitement des macros reste particulier et il est nécessaire d'identifier le fait qu'il s'agit d'une macro : cela s'obtient par la fonction `macro-function`, qui a un rôle ambigu, puisqu'elle sert de prédicat et retourne une valeur fonctionnelle.

```
(defun MEVAL (expr &optional env)
   (cond ...
3        ...                                                    ; quote
10       ((not(fboundp (car expr)))
15       ((macro-function (car expr))                          ; macro prédéfinie
          (meval (displace expr
                           (apply (macro-function (car expr)) expr () ()))
                 env))
1/7      ((special-form-p (car expr)) ...)          ; forme spéciale non traitée
11       (t ...)                                               ; fonction globale
         ))
```

On constatera dans le manuel que `macro-function` retourne une valeur fonctionnelle qui prend 2 paramètres : le premier est l'expression d'origine et le deuxième est un environnement dont la forme et le rôle ne sont pas spécifiées (on utilisera donc `nil`). Une forme exactement équivalente mais moins tributaire de ces détails de spécification consiste à faire appel à la fonction de macro-expansion, `macroexpand-1`. On obtient alors l'expression `(meval (displace expr (macroexpand-1 expr)) env)`.

Remarque 4.8. Attention à ne pas confondre `macroexpand` et `macroexpand-1` : la seconde ne fait qu'un pas de macro-expansion alors que la première ré-expanse le résultat. La bonne fonction est donc `macroexpand-1`, qui repassera la main à `meval` avant une éventuelle ré-expansion. Il peut y avoir une différence si la macro suivante est connue aussi bien par `eval` que par `meval`.

EXERCICE 4.19. Définir la fonction `m-macroexpand-1` qui expanse une fois une macro méta-définie, par analogie avec `macroexpand-1`. □

EXERCICE 4.20. Définir la fonction `m-macroexpand` qui expanse complètement une macro méta-définie, par analogie avec `macroexpand`. Le principe de `macroexpand` est d'appliquer `macroexpand-1` tant que le résultat de l'expansion est toujours une macro. On traitera dans cette fonction aussi bien les macros méta-définies que les prédéfinies. □

4.4.4 Let et cond

Il reste encore à traiter `let` et `cond` qui ont un statut particulier car les deux sont des formes syntaxiques (`special-form-p` retourne vrai), mais les deux peuvent être définies comme des macros (cf. Chapitre 3 du polycopié LISP [Duc13a]). De plus `cond` est effectivement défini comme une macro, mais ce n'est pas le cas de `let`.

Dans la séquence de code précédente, on constate que le cas des macros (16) a été placé avant le ramasse-miettes des formes syntaxiques (1/7). La forme syntaxique `cond` va donc être traitée comme une macro. En revanche, le cas de `let` reste entier. On peut le traiter comme une forme syntaxique ou comme une macro qui ne doit être définie que pour `meval` (mais surtout pas pour `eval`) : on mettra ce genre de fonction dans un fichier à part, qui doit être chargé exclusivement par `mload`.

EXERCICE 4.21. Définir la fonction `meval-let` qui méta-évalue une expression `let`. □

EXERCICE 4.22. Définir la fonction `meval-cond` qui méta-évalue une expression `cond`, comme si c'était une forme syntaxique. □

4.4.5 `Setf`

Une couverture minimale du langage nécessite de traiter `setf`. Comme pour la conditionnelle, l'affectation par `meval` ne peut se résoudre que par une affectation par `eval`. On ne traitera ici que le cas binaire, la généralisation est proposée en exercice. Comme le cas est assez compliqué, on passe par une fonction annexe :

```
(defun MEVAL (expr &optional env)
   (cond ...
3        ...                                                    ; quote
16       ((eq 'setf (car expr))                                ;  affectation
          (msetf (cadr expr) (caddr expr) env))
         ...))
```

Sur les variables Dans le cas des variables, on prendra une définition très stricte (plus stricte que celle de COMMON LISP) de l'affectation : elle n'est valide que si l'évaluation de la variable est valide, c'est-à-dire si la variable figure bien dans l'environnement courant (cas 4). L'affectation se traduit par la modification de la liaison existante :

```
(defun MSETF (place val env)
   (if (atom place)
       (let ((cell (assoc place env)))
          (if cell
              (setf (cdr cell)
                    (meval val env))
            (error "~s n'est pas une variable" place)))
     ...))
```

Sur les champs des objets construits Si l'affectation se fait sur un champ d'objet construit, par exemple `car` ou `cdr`, `place` est de la forme `(foo <arg>)`. Il faut alors évaluer `<arg>` et on peut énumérer tous les `foo` possibles, en traitant chacun par le `setf` approprié :

```
(defun MSETF (place val env)
   (if (atom place)
       ...
     (cond ((eq (car place) 'car)
            (setf (car (meval (cadr place) env))
                  (meval val env)))
           ((eq (car place) 'cdr)
            (setf (cdr (meval (cadr place) env))
                  (meval val env)))
           ...)))
```

mais c'est assez fastidieux...
C'est par ailleurs insuffisant car dans ce cas, `place` doit d'abord être macro-expansé.

EXERCICE 4.23. Étendre `msetf` à l'affectation d'arité quelconque. □

EXERCICE 4.24. Intégrer la macro-expansion de `place` dans `msetf` : traiter les deux cas de macros méta-définies et prédéfinies. □

EXERCICE 4.25. Au lieu d'énumérer dans `msetf` toutes les fonctions `setf`-able, le mieux est d'évaluer tous les arguments de `place`, de reconstruire l'expression `place` en remplaçant les arguments par leur

valeur *quotée*, d'évaluer `val`, puis de reconstruire l'expression `expr` avec `place` transformée et la valeur de `val` *quotée*. On peut alors évaluer `expr` pour effectuer l'affectation : tous les arguments étant *quotés*, cette évaluation peut se faire dans un environnement vide. □

Remarque 4.9. Toute expression « `val` » est équivalente à « `',val` » et à « `'(eval ',val)` » : on peut toujours rajouter des « `eval` » et des « `'` » en parallèle. Voir Chapitre 3 du polycopié LISP [Duc13a], Section sur les macros et *backquote*.

4.5 Fermetures

Voir la section correspondante dans le chapitre 3 du polycopié de LISP [Duc13a]. Une valeur fonctionnelle est en fait une fermeture (en anglais *closure*), c'est-à-dire une λ-fonction couplée à un environnement. C'est donc une structure de donnée particulière : le plus simple est d'utiliser un simple cellule LISP, que l'on peut éventuellement préfixée par un mot-clé pour aider au débugguage.

D'où la fonction `make-closure` pour fabriquer une fermeture :

```
(defun MAKE-CLOSURE (lmbd env)
   `(:closure ,lmbd . ,env))
```

Bien entendu, comme pour les environnements, cette structure de donnée est propre à `meval` et n'a rien à voir avec celle d'`eval`.

Appliquer une fermeture revient alors à appliquer la λ-fonction associée dans l'environnement capturé, d'où la fonction `meval-closure` :

```
(defun MEVAL-CLOSURE (clos args)
    (meval-lambda (cadr clos) args (cddr clos)))
```

4.5.1 Fonctions globales

On transforme alors très simplement le cas des fonctions globales méta-définies, en remplaçant les simples λ-fonction par des fermetures :

```
(defun MEVAL (expr &optional env)
   (cond ...
6        ...                          ; une fonction est un symbole non constant
8        ((get-defun (car expr))                              ;  fonction globale
          (meval-closure (get-defun (car expr)) (meval-args (cdr expr) env)))
9        ((eq 'defun (car expr))                                          ;  defun
          (setf (get-defun (cadr expr))
                (make-closure `(lambda ,@(cddr expr)) env)))
3                                                                         ; quote
         ...))
```

EXERCICE 4.26. Définir les 3 fonctions de l'exemple du compteur dans le polycopié LISP [Duc13a], Section 3.4.3, et tester `meval`. □

4.5.2 `function` et `apply`

Le traitement de `function` et `apply` est un peu plus compliqué car il faut différencier les fonctions prédéfinies des méta-définies. Pour `function`, il faut distinguer les trois cas d'une λ-fonction, d'une fonction globale méta-définie et d'une fonction prédéfinie :

```
(defun MEVAL (expr &optional env)
   (cond ...
3         ...                                                    ; quote
17        ((eq 'function (car expr))                          ;  function
          (cond ((consp (cadr expr))              ;  on suppose une λ-fonction
                 (make-closure (cadr expr) env))
                ((get-defun (cadr expr)))      ;  fonction globale méta-définie
                (t (symbol-function (cadr expr)))       ;  fonction prédéfinie
          ))
         ...))
```

Le cas de `apply` est à peu près symétrique, avec deux cas seulement, suivant que la valeur fonctionnelle est de `meval` ou d'`eval` :

```
(defun MEVAL (expr &optional env)
   (cond ...
3         ...                                                    ; function
18        ((eq 'apply (car expr))
          (let ((clos (meval (cadr expr) env))
                (args (meval-args* (cddr expr) env)))
             (cond ((consp clos)                    ; on suppose une fermeture
                    (meval-closure clos args))
                   ((functionp clos)          ; valeur fonctionnelle prédéfinie
                    (apply clos args)))))
         ...))
```

Une petite difficulté apparaît : le dernier argument de `apply` est la fameuse *liste des arguments restants* de la fonction passée en argument. Il n'est donc pas possible d'utiliser la fonction `meval-args` et il faut une fonction spéciale, `meval-args*`.

EXERCICE 4.27. Définir cette fonction `meval-args*` qui est à `meval-args` ce que `list*` est à `list` (voir ces fonctions dans le polycopié de LISP). □

EXERCICE 4.28. Tester les fermetures sur le schéma de terminalisation des récursions enveloppées par passage de continuation. Voir polycopié de LISP. □

4.5.3 Fonctions locales

Une fois que les fermetures sont implémentées, on peut envisager de s'attaquer aux fonctions locales, `flet` et `labels`.

Environnement fonctionnel lexical Les fonctions locales font appel à un environnement lexical dédié. Cet environnement peut prendre la même forme de *liste d'association* que l'environnement lexical des variables, mais la fonction `meval` doit prendre un paramètre supplémentaire :

```
(defun MEVAL (expr &optional env fenv)
   (cond ...
         ))
```

Il faut donc commencer par rajouter à tous les appels récursifs de `meval` l'argument supplémentaire `fenv`, et le rajouter comme paramètre à toutes les fonctions auxiliaires (`meval-args`, `meval-lambda`, `meval-body`, etc.) qui appellent récursivement `meval`.

Remarque 4.10. La traitement des fonctions locales est une étape importante, qui nécessite des changements nombreux dans le code du méta-évaluateur. Avant d'y procéder, il est indispensable de s'assurer que la version courante du méta-évaluateur fonctionne parfaitement sur tous les cas traités jusqu'ici, y compris méta-évaluation d'ordre supérieur et fermetures. On procède alors à un archivage soigneux de la version courante et on peut commencer les modifications sur un clone.

Application d'une fonction locale Le principe de l'application d'une fonction locale est simple :

```
(defun MEVAL (expr &optional env fenv)
   (cond ...
6        ...                          ; une fonction est un symbole non constant
19       ((assoc (car expr) fenv)                              ; fonction locale
          (meval-closure (cdr (assoc (car expr) fenv))
                         (meval-args (cdr expr) env fenv)))
8        ((get-defun (car expr)) ...)          ;  fonction globale méta-définie
         ...))
```

Comme une fonction locale a priorité sur toute autre fonction, il faut placer ce cas au tout début du traitement des symboles.

Remarque 4.11. Il n'y a pas à traiter les fonctions locales prédéfinies, qui ne sont pas accessibles au méta-évaluateur puisqu'elles sont encapsulées dans une fonction globale prédéfinies, donc invisibles.

Fonctions locales et fermetures L'ajout de paramètre pour l'environnement fonctionnel doit s'effectuer aussi sur les fermetures qui doivent capturer l'environnement fonctionnel. Par ailleurs, les fonctions locales sont des fermetures et `function` s'applique, ce qui entraîne un 4ième cas.

EXERCICE 4.29. Redéfinir les fonctions `make-closure` et `meval-closure` pour tenir compte de l'environnement fonctionnel. □

EXERCICE 4.30. Étendre le traitement de `function` aux fonctions locales. □

Fonctions locales non récursives : `flet` La définition des fonctions locales est plus délicate. Le langage propose deux constructions suivant qu'il s'agit de fonctions non récursives ou de fonctions mutuellement récursives.

Le principe de `flet` consiste à méta-évaluer le corps du `flet` dans l'environnement fonctionnel construit à cet effet :

```
(defun MEVAL (expr &optional env fenv)
   (cond ...
3        ...                                                        ; quote
20       ((eq 'flet (car expr))                  ; fonction locale non récursive
          (meval-body (cddr expr)
                      env
                      (make-flet-fenv (cadr expr) env fenv)))
         ...))
```

Le principal réside dans cette construction : pour chaque fonction définie dans le `flet`, une fermeture doit être créée, contenant la λ-fonction associée et capturant les environnements lexicaux de variables et de fonctions.

EXERCICE 4.31. Définir la fonction `make-flet-fenv` qui construit cet environnement fonctionnel. □

Fonctions locales récursives : `labels` Le cas de `labels` est similaire mais il diffère par la construction de l'environnement fonctionnel.

```
(defun MEVAL (expr &optional env fenv)
   (cond ...
20       ...                                                        ; flet
21       ((eq 'labels (car expr))                  ; fonction locale récursive
          (meval-body (cddr expr)
                      env
                      (make-labels-fenv (cadr expr) env fenv)))
         ...))
```

La différence, et la difficulté, est que les fonctions locales ainsi définies doivent pouvoir s'appeler mutuellement : l'environnement fonctionnel qu'elles capturent n'est donc pas l'environnement `fenv` initial mais l'environnement résultant de l'appel de `make-labels-fenv`. La structure résultante est donc éminemment circulaire.

EXERCICE 4.32. Définir la fonction `make-labels-fenv` qui construit cet environnement fonctionnel circulaire. On appellera `make-flet-fenv` en lui passant un environnement fonctionnel "vide" qu'il s'agira ensuite de remplacer par son résultat même, par exemple par appel de `displace`. □

Remarque 4.12. Les structures circulaires sont délicates à manipuler (cf. Section sur la chirurgie de listes dans le polycopié LISP). Avant d'en faire, il est impératif de donner aux variables suivantes une valeur entière positive, par exemple :

```
(setf *print-length* 1000 *print-level* 100)
```

Sinon, la fonction `print` partira en boucle infinie à chaque tentative d'impression.

4.6 Constructions non traitées

Plusieurs parties non essentielles de COMMON LISP ne sont pas traitées dans ce méta-évaluateur. Il s'agit des environnements dynamique et global, des échappements, des blocs, des macros locales et des macros symboles.

4.7 Extensions à COMMON LISP

Il y a de nombreuses façon d'étendre le langage.

4.7.1 De la déstructuration au filtrage

Le filtrage est une structure de contrôle et un mécanisme de passage de paramètres basés sur l'appariement d'arbres. Christian Queinnec lui a consacré un joli petit livre [Que90], malheureusement épuisé mais disponible sur sa page Web.

Déstructuration

La déstructuration est un mode alternatif de passage de paramètres, qui généralise le passage de paramètres obligatoires de COMMON LISP ainsi que le mot-clé `&rest`. Elle est basée sur l'appariement d'un arbre de paramètres et d'un arbre de valeurs, le premier déstructurant le second.

L'arbre des paramètres est n'importe quel arbre binaire dont les feuilles sont exclusivement des symboles non constants ou `nil`.

<param-tree>	:=	<symbol-var> | « **nil** » |
		« **(** » <param-tree>+ « **.** » <param-tree> « **)** »

C'est l'arbre de paramètre qui guide la déstructuration, suivant l'analyse par cas résumée dans le tableau ci-dessous :

paramètres	arguments	→	action
`consp`	`consp`	→	double récursion parallèle
`null`	`null`	→	OK
`symbolp`	quelconque	→	liaison
autre		→	`error`

Lorsque la liste des paramètres est une *liste plate et propre*, par exemple `(x y z)`, la déstructuration équivaut au passage de paramètres obligatoires. Lorsque la liste des paramètres est une *liste plate* mais pas *propre*, par exemple `(x y z . w)`, la déstructuration équivaut au passage de paramètres obligatoires

avec un paramètre `&rest`, c'est-à-dire à `(x y z &rest w)`. Lorsque l'arbre des paramètres n'est plus une liste plate, la déstructuration permet de déstructurer un argument, en le considérant comme un arbre et en nommant ses sous-arbres.

En COMMON LISP, la déstructuration est une forme alternative de passage de paramètres pour les macros et elle est aussi réalisée par la forme spéciale `destructuring-bind` qui est une variante de `let` (voir le manuel COMMON LISP).

EXERCICE 4.33. Définir la fonction `destruct` qui construit un environnement de façon similaire à `make-env` mais avec la déstructuration. □

EXERCICE 4.34. Traiter la forme syntaxique `destructuring-bind` dans `meval`. □

EXERCICE 4.35. Étendre la fonction `make-env` pour qu'elle inclue la déstructuration sur les paramètres obligatoires, tout en conservant la possibilité des mots-clés `&optional`, `&key` et `&rest` avec leur syntaxe habituelle. □

EXERCICE 4.36. Étendre la fonction `destruct` pour qu'elle interdise la double occurrence d'un paramètre dans l'arbre. □

L'appariement ou "*matching*"

L'étape suivante consiste à lever les contraintes portant sur l'arbre de paramètres, que l'on appellera maintenant un *motif* (*pattern*) : des atomes constants autres que `nil` sont autorisés, et la même variable peut apparaître plusieurs fois, mais ces situations contraignent l'arbre des arguments. De plus, il est possible d'utiliser une variable anonyme, « _ », pour les éléments de l'arbre d'arguments qui ne sont ni mémorisés ni contraints.

<pattern>	:=	<symbol-var> | <constant> |
		« (» <pattern>+ « **.** » <pattern> «) »

C'est toujours l'arbre des paramètres qui mène la danse.

paramètres	arguments	→	action
`consp`	`consp`	→	double récursion parallèle
`constantp`	`constantp`	→	OK si `eql`
		→	sinon :`fail`
_	quelconque	→	rien
`symbolp`	quelconque	→	liaison si pas dans l'environnement
		→	sinon, OK si lié à une valeur `equal`
		→	sinon :`fail`

EXERCICE 4.37. Définir la fonction `match` qui apparie un motif et une valeur dans un environnement qu'elle étend et retourne. En cas d'échec, retourne le mot-clé :`fail`. La fonction est similaire à `destruct` mais elle intègre ces nouvelles contraintes. □

Le filtrage

Le filtrage consiste à se servir du *matching* comme structure de contrôle. On définit donc une forme syntaxique `case-match`, similaire à un `case`, sauf que la comparaison se fait par *matching*.

<case-match>	:=	« (**case-match** » <expr-eval> <clause-match>+ «) »
<clause-match>	:=	« (» <pattern> <expr-eval>* «) »

L'évaluation du corps de la clause s'effectue ensuite dans l'environnement résultant du *matching*.

Remarque 4.13. Il y a deux interprétations possibles de `case-match` dans le cas où une variable d'un motif appartient à l'environnement englobant, par exemple :

```
(let ((x ..)) (case-match .. ((.. x ..) ...) ...))
```

Le `x` du motif peut désigner le `x` du `let` ou, au contraire, une nouvelle variable. La première interprétation permet d'utiliser une variable de l'environnement pour contraindre la donnée. Comme la portée est lexicale, le programmeur peut choisir un autre symbole s'il veut une nouvelle variable : en conséquence, il est inutile d'avoir deux constructions différentes et la première interprétation doit être préférée.

Le corps d'une fonction peut se définir par filtrage. Il suffit de lui donner la structure suivante :

```
(defun foo (&rest tree)
    (case-match tree
          (<motif1> ...)
          (<motif2> ...)
          ...
          (<symbol-var> ...)))                                    ; cas terminal
```

Une forme alternative consisterait à définir la fonction par des règles successives, un peu comme en PROLOG, avec la forme syntaxique `defil` (« `fil` » pour *filtre*) :

```
(defil foo <motif1> ...)
(defil foo <motif2> ...)
...
(defil foo <symbol_var> ...)                                      ; cas terminal
```

L'ordre des filtres est donné par l'ordre de définition, et il est impossible de définir un nouveau filtre après avoir défini le filtre terminal dont le motif est un `<symbol-var>` ou _. Pour le cas terminal, on utilisera une variable si une action est associée ou _ s'il n'y a aucune action :

```
(defil foo _)                                                     ; cas terminal
```

EXERCICE 4.38. Définir la fonction `meval-case-match` qui implémente la forme syntaxique `case-match` dans le méta-évaluateur. □

EXERCICE 4.39. Définir la macro `defil` qui construit progressivement le `case-match` qui fait office de corps de la fonction. □

Le filtrage s'applique particulièrement bien aux macros, les différents motifs correspondant à l'analyse par cas à faire sur la syntaxe de l'expression.

EXERCICE 4.40. Définir la macro `defil-macro` qui construit progressivement le `case-match` qui fait office de corps d'une macro. □

EXERCICE 4.41. Définir la macro `or` par filtrage. Faire de même pour les différents exemples de macros du polycopié de LISP. □

Règles de réécriture

Une autre application du filtrage est la réécriture. Une règle de réécriture est constituée d'un *motif* et d'une *production* qui sont tous deux des arbres de paramètres. Le principe consiste à apparier le motif avec une donnée : si l'appariement réussit, on remplace dans la production chaque paramètre par la donnée à laquelle elle est associée.

En pratique, on dispose d'un ensemble ordonné de règles de réécriture et on applique à la donnée la première règle qui s'applique, jusqu'à ce qu'il n'y en ait plus. L'arrêt dépend bien entendu de la structure des règles. Des règles "simplificatrices", où la taille de la production est toujours strictement inférieure à celle du motif, s'arrêteront toujours.

La réécriture peut être indéterministe, en particulier avec les variables segments.

EXERCICE 4.42. Définir la fonction `rewrite-1` qui prend en entrée une donnée, un motif et une production et réécrit la donnée suivant la règle de réécriture donnée par le motif et la production. Retourne `:fail` si l'appariement ne réussit pas. □

EXERCICE 4.43. Définir la fonction `rewrite` qui prend en entrée une donnée et une liste de règles de réécriture et réécrit la donnée tant qu'une règle s'applique. □

EXERCICE 4.44. Définir la macro `defrewrite-macro` qui définit une macro par des règles de réécriture, comme les `let-syntax` et `syntax-rules` de SCHEME. Cela revient à remplacer la construction explicite de l'expansion, avec *backquote*, par une construction implicite où l'action associée à chaque motif est implicitement *backquotée* et où chaque variable figurant dans le motif y est implicitement *virgulée*. □

EXERCICE 4.45. Définir la macro `or` par règles de réécriture et faire de même pour les différents exemples de macros du polycopié de LISP. Comment pourrait-on éviter avec `or` les problèmes de capture de variable ? (cf. Section sur les macros dans le chapitre 3 du polycopié LISP.) □

Variables "segment"

Une dernière extension consiste à introduire des *variables "segment"* qui peuvent s'apparier à une sous-liste de la donnée. On utilisera pour cela des symboles particuliers, par exemple commençant par « * ».

<pattern>	:=	<pattern-simple> | <pattern-seg>
<pattern-seg>	:=	<symbol-seg> | <pattern-simple>
<pattern-simple>	:=	<symbol-var> | <constant> |
		« **(** » <pattern-seg>+ « **.** » <pattern-simple> « **)** »
<symbol-seg>	:=	« * »<symbol-var>

La notion de "segment" n'est cependant qu'un point de vue sur une variable : `x` et `*x` sont 2 occurrences de la même variable `x`, sous des points de vue différents. Il n'y a donc plus de « * » dans le résultat :

```
(*x a x)   (1 2 a (1 2))   →   ((x . (1 2)))
```

Le principe des variables segment conduit à un certain indéterminisme. En effet, il peut y avoir maintenant plusieurs appariements possibles :

```
(*x a *y)   (1 2 a 3 a 4)   →   ((x . (1 2))(y . (3 a 4)))
                            →   ((x . (1 2 a 3))(y . (4)))
```

Mais le flot de contrôle du filtrage ne suivra que l'un d'eux.

EXERCICE 4.46. Étendre la fonction `match` aux variables segments. On définira deux fonctions auxiliaires pour tester si une variable est segment et pour en extraire la variable simple correspondante. Pour simplifier ce traitement, on peut utiliser une forme parenthésée pour les segments, par exemple `(* x)`, avec le risque de limiter les squelettes possibles : il faut en tout cas bien placer la clause sur les segments. □

Remarque 4.14. Le *matching* peut être interprété comme l'inverse de *backquote* (voir la section sur les macros dans le chapitre 3 du polycopié LISP) : dans les deux fonctionnalités se retrouve la même notion de *squelette*, constitué de constantes et de l'imbrication des « `(..)` », avec des parties variables. *Backquote* sert à générer une expression en insérant ces parties variables, alors que le *matching* sert à les extraire d'une donnée. De plus, variables simples et variables segment sont exactement analogues à « `,` » et « `,@` ». On pourra donc s'inspirer de `backquotify` pour définir `match`.

4.7.2 Retardement et flots

Un flot est une liste plate dont l'évaluation de la queue (CDR) est *retardée*. Les flots permettent ainsi de manipuler des listes virtuellement infinies.

Pour définir le retardement d'évaluation il est possible d'introduire de nouvelles formes syntaxiques `delay` et `force` :

— `delay` fabrique une fermeture sans paramètre dont le corps est l'argument non évalué de `delay` ;

— `force` prend en argument (évalué) une fermeture sans paramètre — fabriquée par `delay` — et provoque son évaluation.

EXERCICE 4.47. Définir dans `meval` les deux formes syntaxiques `delay` et `force`. □

Une fois ces formes syntaxiques définies, on peut définir des fonctions de manipulations de flots, en particulier `vide-f`, `tete-f` et `queue-f` qui sont les analogues de `null`, `car` et `cdr` pour les flots :

```
(defun vide-f (flot) (null flot))
(defun tete-f (flot) (car flot))
(defun queue-f (flot) (force (cdr flot)))
```

Cette mécanique de base une fois définie, il est possible de définir des flots. Par exemple, la fonction suivante définit le flot des entiers à partir de son argument `n` :

```
(defun enumerer-f (n) (cons n (delay (enumerer-f (1+ n)))))
```

Remarque 4.15. Il s'agit bien entendu de *méta-définitions* : le code de ces différentes fonctions doit forcément être méta-évalué, puisque les flots ne sont pas connus par `eval`.

EXERCICE 4.48. Il n'est en fait pas nécessaire de passer par des formes syntaxiques pour définir les retardements. Définir `delay` comme une macro et `force` comme une fonction. □

EXERCICE 4.49. Définir le flot `enum-fibo` qui énumère la suite de Fibonacci. Voir aussi la fonction `next-fibo` dans le polycopié de LISP. □

EXERCICE 4.50. Définir le flot `enum-prime` qui énumère les nombres premiers. Voir aussi la fonction `next-prime` dans le polycopié de LISP. □

Remarque 4.16. Faire de `delay` une macro souligne son rôle de sucre syntaxique : en SCHEME, qui ne dispose pas de macros globales, ce sucre syntaxique nécessite une forme spéciale. Mais le programmeur peut aussi se passer de sucre.

4.8 Alternative à COMMON LISP : Méta-évaluateur SCHEME

On se reportera au chapitre 2 de LISP pour une comparaison de COMMON LISP et SCHEME. La différence principale est que le premier élément d'une liste évaluable est une expression évaluable, dont l'évaluation doit retourner une fonction. En conséquence, il n'y a pas un environnement global particulier — géré par `get-defun` dans `meval` — pour les fonctions globales mais elles sont dans l'environnement lexical courant.

SCHEME peut intervenir ici de 2 façons différentes, suivant que l'on veut méta-évaluer SCHEME ou en SCHEME. Au total, cela fait 3 cas : on peut faire un méta-évaluateur SCHEME en COMMON LISP, un méta-évaluateur SCHEME en SCHEME ou enfin un méta-évaluateur COMMON LISP en SCHEME. Seul le deuxième cas permet une méta-évaluation d'ordre supérieur sans limitation. Cependant, si on note $\texttt{meval}_i^j$ le méta-évaluateur du langage j écrit dans le langage i, on peut toujours faire $(\texttt{meval}_i^j\ \texttt{'}(\texttt{meval}_j^k\ \texttt{'<expr>}_k))$ où $\texttt{<expr>}_k$ est une expression du langage k.

4.8.1 Méta-évaluateur en SCHEME

Pour les fonctions prédéfinies, on peut traiter de façon spécifique le cas de l'évaluation des symboles en position fonctionnelle : si le symbole n'a pas de valeur dans l'environnement lexical on cherchera simplement à l'évaluer par appel à `eval` ; on peut alors remplacer la fonction `symbol-function` à un paramètre par une fonction à 2 paramètres :

```
(defun SCHEME-SYMBOL-FUNCTION (symb env)
    (or (meval symb env) (eval symb)))
```

Il n'est en fait pas possible d'évaluer cette expression sans lever une exception si la fonction n'est pas dans l'environnement `env`. Il faut donc remplacer l'appel à `meval` par le code du traitement du cas des variables (4), en remplaçant l'appel à `error` par l'appel à `eval`.

EXERCICE 4.51. Définir la fonction `scheme-symbol-function`. □

4.8.2 Méta-évaluateur de SCHEME

La modification principale à effectuer consiste à évaluer le premier élément de `expr` quand c'est une liste. L'environnement global des fonctions prédéfinies (la fonction `symbol-function`) doit être remplacée. On peut construire à la main un environnement avec les principales fonctions prédéfinies, par exemple

```
(mapcar 'cons '(+ - car cons cdr ...) (list + - car cons cdr ...))
```

à utiliser comme paramètre par défaut de `meval` (on remplace alors simplement `symbol-function` par un appel à `meval`).

Par ailleurs, `defun`, `function` et `flet` n'existent pas en SCHEME car `define` y joue le triple rôle de `defun`, `let` et `flet`. Et `labels` est remplacé par `letrec`. Les macros globales (`defmacro`) sont aussi remplacées en SCHEME par des transformations syntaxiques à portée lexicales (`let-syntax`, `letrec-syntax`) qui ont la particularité supplémentaire d'être définies par règles de réécriture (Section 4.7.1).

Les modifications à faire sont donc nombreuses mais ne posent pas de problème particulier.

5

Un méta-évaluateur moins naïf

(chapitre passablement remanié et étendu en décembre 2013)

Le méta-évaluateur du chapitre précédent est naïf. Il se contente de parcourir le code LISP tel qu'il est, en refaisant les mêmes vérifications à chaque évaluation. C'est doublement inefficace, à cause du trop grand nombre de cas considérés et à cause de l'absence de mémorisation des tests déjà effectués. Une façon de faire plus efficace consiste à transformer superficiellement le code, dans un langage intermédiaire très proche de LISP, pour simplifier l'analyse par cas, tout en effectuant toutes les analyses statiques possibles.

Au lieu d'avoir un unique programme (`meval`), on se retrouve donc avec 2 programmes distincts : le traducteur (`lisp2li`) et l'évaluateur (`eval-li`).

Avec cette approche, l'analyse par cas de l'évaluation repose, à l'exécution, sur des tests immédiats, chaque cas pouvant même être codé par un octet (*bytecode*). Le code peut ensuite être organisé de telle sorte que l'évaluation de chaque expression saute directement au traitement de la suivante (le terme anglais est *threaded* [1] *interpreter*) mais c'est une autre histoire, qui est effleurée dans le chapitre 8...

De façon générale, cette approche illustre la continuité entre interprétation et compilation : la présente transformation est une compilation dans un langage cible presqu'aussi évolué que le langage source et qui en conserve les grandes structures, alors que la compilation concerne généralement un langage cible moins évolué et aux structures beaucoup plus pauvres que le langage source.

5.1 Principe du langage intermédiaire (LI) et de la transformation

La transformation est effectuée lors de la méta-évaluation d'une expression de *top-level*, avant l'évaluation proprement dite, donc statiquement. Le plus gros de la transformation est donc fait en définissant les fonctions (`defun`).

5.1.1 Syntaxe du langage intermédiaire

La syntaxe du langage intermédiaire est la suivante :

<expr-li>	:=	« (» <keyword> « . » <expr> «) »

Une expression évaluable du langage intermédiaire est une expression LISP préfixée par un mot-clé, c'est-à-dire un `keyword`, symbole LISP préfixé par « : ». Ces mots-clés sont en nombre limité et la syntaxe peut-être décrite plus précisément comme suit :

1. Le terme était utilisé de façon impropre dans des versions antérieures de ce cours. On trouvera dans le chapitre sur l'implémentation de la machine virtuelle son sens original (Section 8.4.2).

<expr-li>	:=	« (:**const .** » <expr> «) » |
		« (:**var .** » <int> «) » |
		« (:**if** » <expr-li> <expr-li> « **.** » <expr-li> «) » |
		« (:**progn** » <expr-li> <expr-li>+ «) » |
		« (:**set-var** » <int> « **.** » <expr-li> «) » |
		« (:**mcall** » <symbol> <expr-li>* «) » |
		« (:**call** » <symbol> <expr-li>* «) » |
		« (:**unknown** » <expr-eval> « **.** » <environ> «) » |
		« (:**let** » <int> <expr-li> <expr-li>+ «) » |
		« (:**lclosure .** » <lambda-fun> «) » |
		« (:**set-fun** » <symbol> <expr-li>+ «) » |
		« (:**apply** » <expr-li>* «) » |
		« (:**cvar** » <int> . <int> «) » |
		« (:**set-cvar** » <int> <int> « **.** » <expr-li> «) » |
		« (:**lcall** » <int> <int> <expr-li>* «) »

Cœur du LI

La syntaxe précédente a été décomposée en deux groupes de mots-clés. Le premier groupe représente le cœur du langage, qu'il faut traiter en priorité, et qui permet par exemple de traiter la fonction `fibo`.

- les constantes (atomiques ou *quotée*) sont regroupées sous le mot-clé `:lit`,
- les variables ont le mot-clé `:var`, et le nom de la variable est remplacé par sa position dans l'environnement,
- tous les `progn` implicites sont explicités avec le mot-clé `:progn`, mais uniquement s'il y a plus d'une expression ;
- les appels de fonctions prédéfinies sont traitées avec le mot clé `:call`,
- les appels de fonctions méta-définies sont préfixés par le mot-clé `:mcall`, et la liste d'arguments est comparée *statiquement*, lors de la transformation, avec la liste de paramètres pour éliminer les erreurs éventuelles ;
- les différentes formes syntaxiques sont en nombre très réduites : le mot-clé `:if` pour `if`, etc.), en ainsi que `:set-var` pour `setf` sur les variables.
- enfin, le mot-clé `:unknown` traitera le cas des fonctions non définies : lors de l'évaluation soit une macro ou une fonction a été définie entre temps, et la transformation est effectuée dynamiquement, soit il y a une erreur.

On note que les macros prédéfinies ou méta-définies sont expansées — par les différentes fonctions `macroexpand` — *statiquement*, donc il n'y a plus de macros dans la syntaxe du langage intermédiaire.

Dans ce contexte, `<lambda-fun>` représente la partie informative d'une valeur fonctionnelle qui est, dans un premier temps, constituée d'une expression du langage intermédiaire et d'une taille d'environnement.

<lambda-fun>	:=	« (» <int> « **.** » <expr-li> «) »

On pourra aussi étiqueter le tout, par exemple par `:lambda`, pour reconnaître la structure de donnée « valeur fonctionnelle » (`<fun-value>`) en débugant :

<fun-value>	:=	« (:**lambda .** » <lambda-fun> «) »

Attention ! `:lambda` n'est pas une instruction du langage intermédiaire mais une étiquette de structure de donnée (un type). Noter que ces valeurs fonctionnelles ne font pas partie du langage, ce sont des données nécessaires pour définir les fonctions (voir le cas de `defun`, plus loin).

Extension du LI

Autour de ce cœur, on rajoute différentes constructions pour traiter les variables locales et les fermetures :

— le cas des `let` et des λ-expressions passe par un mot-clé `:let` avec un entier indiquant la taille de l'environnement à allouer, suivi de `:set-var` et du corps, donc avec un `progn` implicite. Il est en général possible de faire disparaître ce `:let` explicite en augmentant d'autant la taille de l'environnement englobant (voir ci-dessous) ;
— `setf` sur les autres expressions avec `:set-fun`,

(setf (<fun> <args>+) <val>) → (:set-fun <fun> <val> <args>+)

— `:lclosure` permet de créer une fermeture en capturant l'environnement courant, et `:apply` permet d'appliquer une fermeture (voir plus loin) ; bien que `:lclosure` et `:lambda` aient la même syntaxe, la différence est grande puisque le premier mot-clé est une inscruction du langage, alors que le second est un type de donnée ;
— `:cvar` et `:set-cvar` sont les analogues de `:var` et `:set-var` pour les environnements imbriqués.
— enfin, `:lcall` permet d'appeler une fonction locale dont la valeur fonctionnelle est dans l'environnement.

5.1.2 Variables et environnements.

La seule vraie difficulté de la transformation concerne les environnements.
— On pourra implémenter un environnement non vide avec un tableau (voir le manuel CLtL sur les tableaux, fonctions `make-array` et `aref`). Il faut donc associer à chaque variable un indice et calculer la taille de l'environnement nécessaire. Pour un environnement vide, on utilisera `nil`, qui ne coûte rien.
— le traitement de `:var` et `:set-var` est parfaitement trivial puisqu'il s'agit juste d'accéder à l'environnement (`aref`) à la bonne position sans rien vérifier.
— Dans les cas simples d'imbrication d'environnements par `let` ou λ-expression, un seul environnement est nécessaire, dont la taille sera la taille maximale des environnements possibles (cas de flux de contrôle multiples) : l'initialisation des variables se fera ensuite par `:set-var`.
— Avec les fermetures et la capture d'environnement, cela se complique : dans le cas général, l'environnement capturé peut l'être par une fonction récursive. Un seul environnement n'est plus suffisant, et il faut les chaîner, chaque environnement pointant sur l'environnement capturé ou englobant. Si l'on considère que l'environnement capturé est toujours à la même position (0), on identifie une variable par le nombre d'indirections dans la chaîne : on aura donc 2 mots clés pour les variables (à doubler pour l'affectation)
  — `(:var . <indice>)` pour les variables de l'environnement courant,
  — `(:cvar <nb> . <indice>)` pour les variables d'un environnement capturé à distance `<nb>`.
  Un mot-clé `:set-cvar` est aussi nécessaire. On économise deux mots-clés en uniformisant la deuxième syntaxe, mais on peut aussi couper la poire en deux : comme les affectations sont moins nombreuses, on peut avoir un mot-clé en écriture et deux en lecture.

Pour simplifier, on pourra considérer que tout environnement non vide capture un environnement, éventuellement vide : l'indice 0 est toujours occupé par l'environnement capturé et les paramètres des fonctions commencent à l'indice 1. On pourra du coup traiter les `let` et λ-expressions, avec `:let` et comme des fermetures, au moins dans un premier temps.

Dans tous les cas, la transformation d'une fonction produira une expression ayant la syntaxe `(<mot-clé> . <lambda-fun>)`, constituée d'un entier (la taille de l'environnement à créer) ou plus, et d'une unique expression évaluable, le corps. Quant au `<mot-clé>`, ce sera `:lambda` si les fermetures ne sont pas traitées, et c'est alors un litéral, ou bien ce sera `:lclosure` et c'est une instruction dont l'évaluation provoque la capture de l'environnement courant.

5.1.3 Définition et appel de fonctions

— pour les fonctions prédéfinies, il n'y a pas grand chose à faire car il n'est pas possible de vérifier statiquement la correction du nombre d'arguments. Il faut juste vérifier qu'une méta-définition n'a pas été faite.
— pour les fonctions méta-définies,

- La différence importante réside dans le traitement de `defun`, qui doit effectuer la transformation et associer au symbole (nom de la fonction), 2 informations, la taille de l'environnement et le corps (qui est une expression et non plus une liste d'expressions)
- l'appel se fait de façon habituelle, sauf que l'environnement est construit (`make-array`) avant d'être rempli par la valeur des arguments.
- pour les fermetures, il faut traiter spécialement `apply`, comme si c'était une forme syntaxique, donc le remplacer par un mot-clé `:apply`. C'est la même chose pour `function`, qui doit distinguer le cas des λ-fonctions car il provoque une capture effective d'environnement.

— les macros sont expansées dans la transformation et n'apparaissent plus à l'évaluation (sauf avec `:unknown`, qui ne donne lieu à aucun traitement spécifique mais se contente d'appliquer la transformation).

Traitement de `:unknown`

Lors de la transformation d'une expression *a priori* évaluable dont le symbole de fonction n'a pas de définition fonctionnelle (vraie ou fausse, méta- ou prédéfinie), l'expression est juste préfixée par le mot-clé `:unknown`, sans subir aucune transformation.

Lors de l'évaluation de `:unknown`, on regarde si une fonction ou une macro a été définie ou méta-définie entre temps : en cas de réponse positive il faut alors effectuer la transformation correspondante et refaire l'évaluation. Sinon, il s'agit d'une erreur. On remarque que cette seconde tentative de transformation nécessite l'environnement de transformation qui n'est plus disponible lors de l'évaluation : cet environnement doit donc être stocké dans l'instruction. D'où la syntaxe de `:unknown`.

Remarque 5.1. Dans ce méta-évaluateur, la transformation dans le langage intermédiaire est une forme de compilation, basée sur l'état courant du programme. Les modifications ultérieures ne sont pas prises en compte : ainsi en cas de traitement des fermetures par une constante (ci-dessus). Le cas le plus clair représente les fonctions pré- et méta-définies : un appel à une fonction prédéfinie sera transformé en `:call` et cette transformation est irréversible : une méta-définition ultérieure n'y changerait rien, sauf si on "recompile" tout.

Dans cette optique, `:unknown` est le seul cas complètement dynamique, au moins jusqu'à sa première évaluation.

5.2 Mise en œuvre progressive

Les 2 fonctions peuvent se faire en parallèle, chacune de façon incrémentale.

5.2.1 Mise en œuvre progressive de la transformation (lisp2li)

La fonction de transformation `lisp2li` est une fonction récursive très semblable à `meval` — elle a la même structure récursive et présente la même analyse par cas — sauf qu'elle n'évalue pas mais transforme.

Transformation des fonctions et création d'environnement

— la première chose à faire est de construire un *environnement statique*, où les variables ne sont plus associées à des *valeurs* mais à des *positions*, comme en génération de code, suivant la spécification suivante :

```
(x y z)     →  ((x . 1)(y . 2)(z . 3))
(a b c d)   →  ((a . 1)(b . 2)(c . 3)(d . 4))
```

— dans le traitement des fonctions (λ-fonction, `defun`, `labels`) `make-stat-env` sera appliquée aux listes de paramètres ;

— le corps des fonctions sera considéré comme une expression unique, en explicitant s'il le faut le `progn` implicite par un `:progn` ;

— pour les `let`, qu'on traitera d'abord avec `:let`, il faut aussi construire un environnement imbriqué, puis générer les affectations des variables, par `:set-var`, et enfin transformer le corps.

EXERCICE 5.1. Définir cette fonction `make-stat-env` dans une version simple, comme dans l'exemple ci-dessus : les paramètres sont obligatoires et il n'y a pas d'environnement imbriqué. □

EXERCICE 5.2. Étendre la fonction `make-stat-env` pour traiter les environnements imbriqués. On peut adopter l'une des deux spécifications suivantes :

```
(a b) ((x 0 . 1)(y 0 . 2)) → ((a 0 . 1)(b 0 . 2)(x 1 . 1)(y 1 . 2)))
(a b) ((x . 1)(y . 2))     → ((a . 1)(b . 2)(x 1 . 1)(y 1 . 2)))
```

suivant que l'on adopte une syntaxe des environnements uniformément à la `:cvar` ou en distinguant `:var` et `:cvar`. □

Cas de `&optional` et `&rest`

Le traitement des paramètres `&optional` et `&rest` compliquerait un peu inutilement les choses, et il est conseillé de ne pas s'en occuper.

— Pour `&optional` il n'y a rien à faire de particulier si la valeur par défaut est `nil`. Dans le cas contraire, il faudrait introduire un paramètre supplémentaire et rajouter l'argument `t` à l'appel lorsque le paramètre optionnel est fourni ; à l'entrée de la fonction, on teste le paramètre et s'il est `nil`, on affecte la valeur par défaut. Tout ceci se fait par une transformation source-à-source.
Une alternative consiste à passer les valeurs par défaut comme arguments explicites dans l'expression appelante : mais ce ne serait pas possible en compilation séparée.

— pour `&rest`, il faut indiquer le rang du paramètre `&rest` qui doit donc être indiqué par un champ supplémentaire dans la `<lambda-fun>`.

<lambda-fun>	:=	« (» <int> <int> « . » <expr-li> «) »

On peut se poser la question de l'opportunité de faire un mot-clé supplémentaire pour l'appel de fonctions méta-définies avec paramètre `&rest`.

EXERCICE 5.3. Etendre la fonction `make-stat-env` pour traiter les mots-clés `&optional` et `&rest`, toujours sans environnement imbriqué, suivant les spécifications suivantes :

```
(x y &optional (z t))       →  ((x . 1)(y . 2)(z . 3)(z-p . 4))
(a b &optional c &rest d)   →  ((a . 1)(b . 2)(c . 3)(d . 4))
```

On constate que les mots-clés sont sans effet pour l'environnement : ils ne sont traités qu'au passage de paramètres. □

Alternative plus simple pour les environnements imbriqués

La solution précédente consiste à expliciter la position des variables dans l'environnement. Une alternative plus simple consiste à utiliser de simples listes de variables en se basant sur la fonction LISP `position` pour en calculer la position (qu'il faudra juste penser à incrémenter de 1).

```
position y (x y z)     →  1
         c (a b c d)   →  2
```

Pour les environnements imbriqués (cas du `let`), on manipulera alors une liste d'environnements, au lieu d'un environnement unique : l'environnement `((a b)(x y))` correspond ainsi à l'exemple d'environnement imbriqué cité plus haut.

Il faut alors définir une fonction de recherche `search-multi-env` dans les environnements imbriqués qui retourne la paire constituée de la position de l'environnement et de la position de la variable dans l'environnement :

```
search-multi-env x ((a b)(x y)) → (1 . 1)
                 y ((a b)(x y)) → (1 . 2)
                 a ((a b)(x y)) → (0 . 1)
                 b ((a b)(x y)) → (0 . 2)
```

Si la position de l'environnement est 0, on génère un `:var` ou `:set-var`. Si cette position est strictement positive, on génère un `:cvar` ou `:set-cvar`.

EXERCICE 5.4. Définir la fonction `search-multi-env` qui retourne la double position d'une variable dans une liste d'environnements, ou `nil` si la variable est inconnue. □

Le cas de `defun`

La forme syntaxique `defun` est un cas particulier intéressant. Outre que sa transformation doit passer par la création d'un environnement, il est nécessaire de bien considérer les rôles respectifs de `defun` et de `lisp2li`.

La forme syntaxique `defun` sert à définir une fonction, c'est-à-dire à associer une valeur fonctionnelle (λ-fonction) à un symbole (le nom de la fonction) : il s'agit d'une effet de bord. D'un autre côté, le rôle de `lisp2li` est de transformer du code LISP dans du code du langage intermédiaire, sans faire aucun effet de bord. La transformation d'une expression `defun` par `lisp2li` doit donc produire l'expression du langage intermédiaire qui provoquerait la définition de la fonction si on l'évalue avec `eval-li`.

On suppose que l'on dispose d'une fonction LISP `set-defun` qui associe à un symbole sa définition de fonction (que l'on retrouve ensuite par `get-defun`), et qui est une donnée de type `:lambda`.

```
(get-defun <symbol>)                              ⇒ (:lambda <lambda-fun>)
(set-defun <symbol> (:lambda <lambda-fun>)) ⇒
```

Dans ces conditions, la transformation doit être, dans un premier temps, la suivante :

```
(defun fibo (n) (if ...))  ()
            → (:call set-defun (:lit . fibo) (:lit :lambda 1 (:if ...)))
```

Par la suite, on pourra être amené à complexifier la valeur fonctionnelle des fonctions (c'est-à-dire la syntaxe de `<lambda-fun>` ou le deuxième argument de `set-defun`), en y dédoublant l'argument de taille de l'environnement pour différencier la taille de l'environnement proprement dite du nombre de paramètres : ce dernier sert à la vérification statique, alors que la taille de l'environnement peut inclure des variables locales (voir traitement du `let`, plus loin).

Dans un deuxième temps, pour traiter les *fermetures*, on ne traitera plus la valeur fonctionnelle comme un littéral (avec `:lit`), mais on fabriquera à la place une fermeture, avec `:lclosure`, qui remplace alors à la fois `:lit` et `:lambda` (voir plus loin).

Transformation des autres expressions

— expanser les macros avant toute chose : pour les macros méta-définies, on peut se contenter de la macro-expansion par `meval` (mais il faut alors que le code des 2 méta-évaluateurs soit compatible), car ce n'est pas un objectif majeur ici ;
— il est alors possible de transformer une expression dans un environnement, par exemple :

```
lisp2li x ((x . 1) (y . 2))  →  (:var . 1)
        y ())                →  ERROR
```

ou bien, avec la seconde alternative :

```
lisp2li x ((x y))        →  (:var . 1)
        y ((a b) (x y))  →  (:cvar 2 . 1)
        y ())            →  ERROR
```

— la transformation des constantes atomiques ou *quotées* est élémentaire

```
lisp2li 3 ()          →  (:const . 3)
        'x ()         →  (:const . x)
        '(1 2 3) ()   →  (:const 1 2 3)
```

— transformation du `if` et du `progn` : trivial mais à simplifier (voir ci-dessous) ;
— la transformation des appels de fonctions prédéfinies est très simple.
— la transformation des appels de fonctions méta-définies est aussi simple. Il faut juste ajouter une vérification du nombre des arguments (le plus simple est d'essayer de créer un environnement, à la `meval`).
— pour les fonctions locales, il faut rajouter à `lisp2li` et à toutes les fonctions annexes un paramètre supplémentaire pour l'environnement fonctionnel ; cet environnement fonctionnel a la même structure que l'environnement des variables et peut être calculé par la même fonction `make-stat-env`, mais c'est la façon de l'utiliser qui fera la différence entre `flet` et `labels` : la transformation du corps des fonctions locales de `flet` utilise l'ancien environnement, alors que celle de `labels` utilise le nouveau.

Simplification des expressions

Le résultat de la transformation peut souvent être simplifié. C'est le cas par exemple pour :`progn`, :`let` ou :`if` :

```
(:progn <expr>)                                  →  <expr>
(:progn .. (:progn ...) ....)                    →  (:progn .. ... ....)
(:let .. (:progn ...) ....)                      →  (:let .. ... ....)
(:progn .. (:let <i> ...) ....)                  →  (:let <i> .. ... ....)
(:let <i> .. (:let <j> ...) ....)                →  (:let <i+j> .. ... ....)
(:if (:const . nil) <alors> <sinon>)             →  <sinon>
(:if (:const . <non-nil>) <alors> <sinon>)       →  <alors>
```

Dans le cas des :`let` imbriqués, il faut procéder à la renumérotation des variables du :`let` intérieur. Certaines règles peuvent s'appliquer plusieurs fois.

On peut aussi traiter la disparition des :`let` par une simplification :

```
(<j> . (:let <i> ..)) → (<i+j> . (:progn ..))
```

EXERCICE 5.5. Définir la fonction `simplify` qui simplifie les expressions dans les cas simples. □

EXERCICE 5.6. Etendre `simplify` au cas des :`let` imbriqués. Il faut alors traiter les cas suivants sur l'indiçage des variables dans le :`let` interne :

```
(:let <i>..(:let <j> (:var . <k>)...))       → (:let <i+j>..(:var . <i+k>)...)
(:let <i>..(:let <j> (:cvar 1 . <k>)...))    → (:let <i+j>..(:var . <k>)...)
(:let <i>..(:let <j> (:cvar <l> . <k>)...)) → (:let <i+j>..(:cvar <l-1> . <k>)...)
```

et la même chose pour :`set-var`. Dans le cas de la remontée d'un :`let` au :`progn` englobant, il faut aussi renuméroter les variables entre le :`progn` et le :`let`. □

Remarque 5.2. Pour simplifier les expressions on peut aussi se servir de *règles de réécriture* (Voir Section 6.3 du polycopié LISP et Section 4.7.1 du présent polycopié). Plus généralement, toutes les transformations effectuées par `lisp2li` peuvent se faire avec des règles de réécriture accompagnées d'environnements.

Simplification du `let` dans un `defun`

Beaucoup de fonctions ont la forme suivante :

```
(defun foo (x y z)
    (let ((a ..)
          (b ..))
       ...))
```

Il est possible de les traiter avec des environnements imbriqués et une combinaison de `:lambda` et `:let`, ce qui donnerait la valeur fonctionnelle suivante :

```
(:lambda 3 (:let 2 (:set-var 1 ..) (:set-var 2 ..) ...))
```

Mais il est possible de se passer complètement du `:let`, en générant lors de l'appel de `foo` un environnement assez grand pour les variables locales :

```
(:lambda 5 (:set-var 4 ..) (:set-var 5 ..) ...)
```

Comme il est nécessaire de pouvoir vérifier le nombre d'arguments à l'appel de `foo`, il faut alors étendre la syntaxe de `<lambda-fun>` :

<lambda-fun>	:=	« (:**lambda** » <int> <int> « **.** » <expr-li> « **)** »

où les deux entiers représenteront le nombre de paramètres et la taille de l'environnement.

Cette extension de la syntaxe de `:lambda` vaudra aussi pour les mots-clés `:closure` (type fermeture) et `:lclosure` (fabrication d'une fermeture).

5.2.2 Mise en œuvre progressive de l'évaluation (eval-li)

La fonction d'évaluation `eval-li` est une fonction très semblable à `meval` — même structure récursive, même présence d'environnements — sauf qu'elle traite un ensemble de cas beaucoup plus réduit, et de façon beaucoup plus simple.

— Sa structure est celle d'un `case` et non d'un `cond` : l'ordre des clauses n'importe pas, sauf pour des questions d'efficacité.

— La plupart des cas sont simples, voire très simples : le code de `eval-li` est très petit.

— On peut tester au fur et à mesure sans attendre `lisp2li`, en écrivant du code en langage intermédiaire à la main, par exemple :

```
eval-li (:const . 3)                        ()                        →  3
        (:var . 1)                          #(() 4 5 6)               →  5
        (:cvar 1 . 2)                       #(#(() 7 8)) 4 5 6))      →  8
        (:call + (:const . 3) (:var . 1)) #(() 4 5 6))                →  8
```

Bien entendu, définir `fibo` en langage intermédiaire serait un peut plus fastidieux.

— pour les appels de fonctions méta-définies, il faut récupérer la `<lambda-fun>` associée, créer l'environnement, c'est-à-dire un tableau de la bonne taille, le remplir à partir de la liste d'arguments.

— Pour `&rest` il faut indiquer à partir de quel rang il faut conserver la liste des arguments et où il faut l'affecter.

Remarque 5.3. « `#(...)` » est la syntaxe COMMON LISP pour les constantes (ou littéraux) tableaux (`array`), donc l'équivalent de « `'(...)` » pour les listes (plates et propres).

Exercices

EXERCICE 5.7. Une technique classique d'optimisation des programmes consiste à les "profiler" pour déterminer la fréquence d'usage de leurs composants dans le but de réorganiser le code généré de façon optimale. On peut appliquer cette technique à l'analyse par cas — en général mais ce sera particulièrement

simple dans le cas du langage intermédiaire — en comptant le nombre d'occurrences de chacun des cas, de façon à en optimiser l'ordre.

Faites ce profilage et en déduire 3 versions différentes de l'évaluateur. La première prend un ordre quelconque, la seconde suit l'ordre optimal, et la troisième suit l'ordre inverse. Mesurer les performances des 3 versions. □

EXERCICE 5.8. Une optimisation alternative consiste à remplacer les mots-clés par des entiers, et à remplacer le `case` par un arbre binaire équilibré, de profondeur 4. Comparer l'efficacité avec le `case` profilé de l'exercice précédent. □

EXERCICE 5.9. Pour mesurer le gain apporté par ces nouvelles versions du méta-évaluateur, on fera des tests avec les différentes versions, mesurés par la macro `time`, sur des exemples significatifs comme `fibo`, définis de façon directe (double récursion enveloppée) ou avec des techniques plus élaborées comme la terminalisation par continuation (voir la section sur les fermetures dans le chapitre 3 du polycopié de LISP). □

Evaluation d'une liste d'expressions

Comme toutes les fonctions d'analyse par cas, `eval-li` doit être couplée avec une fonction d'évaluation d'une liste d'expressions, qui applique `eval-li` à chaque expression de la liste.

Cette fonction prend en fait trois formes :

— `map-eval-li` construit la liste des valeurs retournée par `eval-li` sur chaque élément de la liste ;
— `eval-li-progn` évalue en séquence les expressions de la liste et retourne la valeur de la dernière ;
— `map-eval-li*` est à `map-eval-li` ce que `list*` est à `list` (voir le manuel CLtL) : on en a besoin pour traiter `:apply`.

EXERCICE 5.10. Définir ces 3 fonctions `map-eval-li`, `eval-li-progn`, et `map-eval-li*`. □

Appel de fonction méta-définie

Pour appeler une fonction méta-définie, on peut définir deux fonctions alternatives :

— `make-env-eval-li` prend en paramètre un entier et une liste de valeurs (produite par `map-eval-li`), crée un environnement (tableau) de la taille de l'entier, et le remplit avec les valeurs de la liste ;
— `map-eval-li+make-env` prend en paramètre un entier, une liste d'expressions du LI, et un environnement, et crée un environnement (tableau) de la taille de l'entier, en le remplissant avec les résultats successifs de l'appel de `eval-li` sur les expressions de la liste, dans l'environnement passé en paramètre.

La deuxième version est bien sûr plus efficace, puisqu'on économise une stucture de donnée intermédiaire.

EXERCICE 5.11. Définir ces 2 fonctions `make-env-eval-li` et `map-eval-li+make-env`. □

EXERCICE 5.12. Traiter le cas de `:mcall` par appel de `make-env-eval-li` et `map-eval-li`, ou bien par `map-eval-li+make-env`. □

5.2.3 Le cas des fermetures

Fermetures dans le langage intermédiaire

Pour les fermetures, toute une combinatoire de possibilités se présente :

— les fermetures sur les λ-fonctions peuvent être traitées par un mot clé particulier `:lclosure`, qui construit la fermeture ;
— on traitera alors `defun`, par `set-defun` (voir plus haut), en utilisant `:lclosure` pour que la définition capture l'environnement courant,
— on modifiera `:mcall` en conséquence : `get-defun` retourne maintenant une fermeture dont l'application est légèrement différente ;

— pour les fermetures accédées par le nom d'une fonction pré- ou méta-définie, la fermeture est déjà construite, donc il n'est pas nécessaire d'avoir un mot-clé ; on peut les traiter comme des constantes, en générant l'expression `'(:const . ,<valeur fonctionnelle>)` ou avec un mot-clé spécial `:sclosure`, voire 2 mots-clés pour distinguer les deux types de fonctions. La première solution est plus efficace mais moins dynamique : l'interprétation dynamique de `:sclosure` permettrait de redéfinir la fonction sans problèmes.
— on peut traiter `apply` de plusieurs façons différentes : si l'argument fonctionnel n'est pas un appel direct de `function`, son type fonctionnel n'est pas connu et il faut faire soit un appel à une fonction spéciale, par exemple `mapply` en transformant `(apply ...)` en `(:call mapply ...)`, soit un mot clé particulier, `:sapply`, qui est traité comme `apply` dans `meval`.
Si l'argument fonctionnel de `apply` est un appel de `function`, son type est connu (fermeture selon `eval` ou selon `meval`) et le code particulier peut être appliqué : on peut donc avoir, soit 2 fonctions d'applications supplémentaires, en se servant de `:call`, soit 2 mots-clés particuliers.

Remarque 5.4. Avoir un mot-clé particulier est plus efficace que de faire appel à une fonction par le biais de `:call` qui rajouterait un appel fonctionnel. Cependant, augmenter inconsidérément le nombre de mots-clés réduirait l'efficacité.

Le plus efficace pour `function` est donc d'avoir un mot-clé pour les fermetures de λ-fonction (`:lclosure`) et de traiter ses autres usages comme des constantes. Pour `apply`, il serait peu utile de discriminer entre les 3 cas avec des mots-clés et un unique mot-clé `:apply` suffit. On peut souhaiter le remplacer par une ou plusieurs fonctions : cela ne change pas grand-chose.

Evaluation des fermetures

Le mot-clé `:lclosure` gouverne la syntaxe des fermetures dans le langage intermédiaire : l'expression prend deux arguments, la taille de l'environnement et le corps. L'évaluation d'une expression `:lclosure`, doit construire une structure de données LISP qui représente une fermeture et est constituée du contenu de l'expression `:lclosure` plus l'environnement courant. A défaut de la typer strictement, on pourra étiqueter cette structure de donnée par un mot-clé, par exemple `:closure`, comme on a fait avec `:lambda` précédemment.

```
eval-li (:lclosure 1 (:if ...)) #(() 4)  →  (:closure #(() 4) 1 (:if ...))
```

Il est essentiel de comprendre la différence entre les deux mots-clés : `:lclosure`, c'est de la syntaxe du langage intermédiaire, et ça désigne une instruction, alors que `:closure` est, comme `:lambda`, l'étiquette d'une structure de donnée, c'est-à-dire un type.

Fermetures au toplevel

On a réussi à éliminer le `:let` dans les fonctions, en intégrant les variables locales dans l'environnement d'appel. Mais comment faire pour les `let` au *toplevel*.

Dans l'exemple du compteur (cf. polycopié de LISP), on définit des fonctions qui se partagent un environnement capturé commun :

```
(let ((n 0))
     (defun compteur () n)
     (defun compteur++ () (setf n (+ 1 n))))
```

Cela va donc donner, dans le langage intermédiaire, en utilisant `:let` :

```
(:let 1
      (:set-var 1 (:lit . 0))
      (:call set-defun (:lit . compteur)
             (:lclosure 0 (:cvar 1 . 1)))
      (:call set-defun (:lit . compteur++)
             (:lclosure 0
                    (:set-cvar 1 1 (:call + (:lit . 1) (:cvar 1 . 1))))))
```

L'évaluation va conduire à la création d'un environnement pour 1 variable, qui sera affectée à 0. Les deux fonctions sont alors définies par une fermeture sans paramètre accédant à la variable de l'environnement capturé. On peut alors vérifier que le résultat ressemble à ceci :

```
get-defun compteur    →  (:closure #(() 0) 0 (:cvar 1 . 1))
          compteur++  →  (:closure #(() 0) 0 (:set-cvar 1 1 ...))
```

Le traitement de :mcall doit être adapté en conséquence.

On peut alors se passer de :let, en générant, puis appelant, une fonction qui va faire les définitions.

```
(:progn (:call set-defun (:lit . <un-symbole>)
            (:lclosure 1 (:progn
                (:call set-defun (:lit . compteur)
                       (:lclosure 0 (:cvar 1 . 1)))
                (:call set-defun (:lit . compteur++)
                       (:lclosure 0
                           (:set-cvar 1 1
                                 (:call + (:lit . 1) (:cvar 1 . 1))))))))
        (:call <un-symbole> (:lit . 0))
```

Dans cet exemple, <un-symbole> est un nom de fonction provisoire que l'on peut générer par la fonction LISP gensym.

5.2.4 Fonctions locales

Jusqu'ici, nous n'avons pas considéré le cas des fonctions locales dans ce nouveau méta-évaluateur. Si l'on veut les traiter, il est inutile d'introduire deux nouveaux mots-clés, comme :flet et :labels, mais il faut traiter les environnements fonctionnels. Cependant, il n'est pas non plus nécessaire de traiter ces environnements fonctionnels de façon spécifique à l'évaluation (dans eval-li) : on peut en réalité traiter flet et labels comme des let, les variables et les fonctions se différenciant par leurs positions dans l'environnement. Le traitement des fonctions locales n'impose donc plus d'ajouter un paramètre supplémentaire à toutes les fonctions du méta-évaluateur et son surcoût est nul, au moins dans eval-li.

En revanche, comme le même symbole peut être utilisé simultanément comme variable et comme fonction, par exemple dans

```
(labels ((foo (foo) (if (= foo 0) 1 (* foo (foo (- foo 1)))))) (foo 10))
```

il est indispensable de rajouter un paramètre d'environnement fonctionnel dans toutes les fonctions de lisp2li.

L'entrée d'un flet ou labels commence donc par allouer l'environnement lexical nécessaire, par un :let ou en l'intégrant dans l'environnement de la fonction englobante. Cet environnement est ensuite affecté par des :set-var successifs, dont la valeur est une fermeture :

```
(:set-var <int> . (:lclosure . <lambda-fun>))
```

Il y a néanmoins une subtilité due au fait que l'environnement à capturer dépend de la construction utilisée, flet et labels : dans le premier cas, il faut capturer l'environnement précédent, alors que dans le second il faut capturer l'environnement courant. On pourrait introduire un deuxième mot-clé pour distinguer les deux. Cependant, la distinction entre flet et labels n'a pas besoin d'être explicitée dans la définition ou l'appel des fonctions locales, ou à l'exécution : en effet, il s'agit d'une simple distinction dans l'interprétation des noms dans le corps des fonctions locales : cette différence d'interprétation se traduit, lors de la transformation, par des paramètres différents de :cvar.

Il faut noter aussi que la circularité des environnements pour labels ne pose pas de problème du fait que l'environnement courant est créé avant d'être rempli, ce qui permet naturellement aux fermetures qu'il contient d'y faire référence.

La valeur fonctionnelle (function) d'une fonction locale s'obtient par :var mais l'appel d'une fonction locale nécessite un mot-clé spécial :lcall, dans lequel le nom de la fonction est remplacé par sa position dans l'environnement. On généralisera en mettant une double position similaire à :cvar. Au total, on étend donc la syntaxe ainsi :

<expr-li>	:=	... |
		« (:**lcall** » <int> <int> <expr-li>* «) »

Une alternative à `:lcall` consisterait à passer par `:apply` et `:var` ou `:cvar`, car les deux expressions suivantes sont équivalentes :

```
(:lcall <i> <j> <e1> ... <ek>)
(:apply (:cvar <i> . <j>) <e1> ... <ek> (:lit . nil))
```

mais ce serait assez inefficace.

Exemple. Soit le code de la fonction `fibo` réécrit avec deux fonctions locales mutuellement récursives :

```
(defun FIBO (m)
   (labels ((f1 (n)
                (if (< n 2) 1 (+ (f2 (- n 1)) (f2 (- n 2)))))
            (f2 (n)
                (if (< n 2) 1 (+ (f1 (- n 1)) (f1 (- n 2))))))
      (f1 m))
```

La valeur fonctionnelle de la fonction `fibo` dans le langage intermédiaire aura alors l'allure suivante :

```
(:closure () 1 3
   (:progn (:set-var 2
              (:lclosure 1 1
                 (:if (:call < (:var . 1) (:lit . 2))
                      (:lit . 1)
                   (:call +
                          (:lcall 1 3 (:call - (:var . 1) (:lit . 1)))
                          (:lcall 1 3 (:call - (:var . 1) (:lit . 2)))))))
           (:set-var 3
              (:lclosure 1 1
                 (:if (:call < (:var . 1) (:lit . 2))
                      (:lit . 1)
                   (:call +
                          (:lcall 1 2 (:call - (:var . 1) (:lit . 1)))
                          (:lcall 1 2 (:call - (:var . 1) (:lit . 2)))))))
           (:lcall 0 2 (:var . 1))))
```

et ceci dans la version la plus complète où les `:let` sont éliminés et `<lambda-fun>` comporte le nombre de paramètres et la taille de l'environnement.

On peut décoder la valeur fonctionnelle de `fibo` comme suit :

— c'est une fermeture qui capture l'environnement vide,
— avec 1 paramètres et deux variables locales,
— dont le corps est un `progn` explicite,
— dont la première instruction affecte à la variable en position 2 une fermeture, qui appelle récursivement la fonction locale en position 3 dans son environnement capturé (celui créé par l'appel de `fibo`,
— l'instruction suivante fait une affectation symétrique,
— et la dernière appelle la fonction locale en position 2 dans l'environnement courant.

5.3 Extensions

5.3.1 Le filtrage

La transformation de `case-match` (Section 4.7.1) peut s'envisager de différentes manières. En toute généralité, on peut continuer à interpréter l'appariement ou on peut chercher à le compiler.

Interprétation de l'appariement

Pour chaque `case-match`, il faut calculer la taille de l'environnement nécessaire, c'est-à-dire le nombre maximum de variables pour tous les motifs. Cette taille détermine l'unique environnement qui sera créé à l'exécution et servira pour la totalité des clauses.

Pour chaque clause, il faut calculer l'environnement statique à partir des variables présentes dans le motif. Avec cet environnement statique, on transformera l'action de la clause de manière habituelle. Il faut aussi transformer le motif lui-même pour remplacer les variables par leur indice, ou un motif qui contient leur indice : si l'on excluait les entiers des motifs, on pourrait utiliser directement l'indice de la variable, mais il est préférable de les préfixer par une expression syntaxique peu ambiguë. On utilisera par exemple `(:var-simple . <indice>)` et `(:var-seg . <indice>)`.

Au total, l'expression transformée aura la syntaxe suivante :

<expr-li>	:=	... |
		« (:**match** » <int> <clause-match2>+ « **)** »
<clause-match2>	:=	« **(** » <pattern2> « **.** » <expr-li> « **)** »
<pattern2>	:=	<pattern-simple2> | <pattern-seg2>
<pattern-seg2>	:=	<symbol-seg2> | <pattern-simple2>
<pattern-simple2>	:=	<symbol-var2> | <constant> |
		« **(** » <pattern-seg2>+ « **.** » <pattern-simple2> « **)** »
<symbol-simple2>	:=	« (:**var-simple .** » <indice> « **)** »
<symbol-seg2>	:=	« (:**var-seg .** » <indice> « **)** »

Il faut enfin adapter la fonction `match` d'appariement à des environnements sous forme de tableau et à la nouvelle syntaxe pour les variables.

EXERCICE 5.13. Définir la fonction `pattern2var` qui calcule la liste de variables présentes dans un motif. □

EXERCICE 5.14. Définir la fonction `pattern-trans` qui transforme un motif (premier paramètre) suivant un environnement (deuxième paramètre). □

EXERCICE 5.15. Avec ces conventions, adapter la fonction `match` qui prend en argument un motif, une valeur et un environnement, remplit l'environnement au fur et à mesure de l'appariement, puis le retourne (ou `:fail` en cas d'échec). □

EXERCICE 5.16. Définir la fonction `lisp2li-match` qui transforme `case-match` dans la syntaxe de `:match`. □

EXERCICE 5.17. Définir la fonction `meval-match` qui méta-évalue la construction `:match`. □

Remarque 5.5. Il faut prendre garde à un détail important : l'environnement étant alloué à l'avance il faut arriver à distinguer l'absence de valeur. On initialisera pour cela l'environnement des variables de *matching* à une valeur distinguée, par exemple `:empty`. Après l'échec d'une clause, il faut bien entendu réinitialiser l'environnement avant de pouvoir s'attaquer à la suivante.

Compilation de l'appariement

Il est encore plus efficace de compiler l'appariement. Si l'on suppose que l'on apparie un motif `pattern` avec une donnée `data` dans un environnement `env`, la compilation de l'appariement correspond aux règles de réécriture suivantes, où chaque motif est traduit par une condition :

`<constant>`	→	`(eql pattern data)`
`_`	→	`t`
`(:var-simple . <i>)`	→	`(:call eql (:var . <i>) (:var . <data>))` `(:set-var <i> . (:var . <data>))`

`<consp>`	→	`(and (consp data)`
		`  (let ((pattern (car pattern))`
		`  (data (car data)))`
		`  <compile>)`
		`  (let ((pattern (cdr pattern))`
		`  (data (cdr data)))`
		`  <compile>)))`

Le cas des variables est particulier. La condition est exprimée dans la syntaxe transformée, qui seule permet l'accès aux environnements : `<data>` représente alors l'indice de la donnée dans l'environnement. D'autre part, la condition générée diffère suivant que la variable a déjà été rencontrée dans le flot de l'appariement ou pas : cela peut-être délicat à réaliser car l'ordre des transformations n'est pas forcément celui de l'exécution.

L'expression `<compile>` représente l'expression résultant de la compilation récursive des sous-expressions. En pratique, la fonction de compilation du motif a 4 arguments : le motif, la donnée, un environnement statique et enfin une "suite", qui est le code à insérer en cas de succès. Cela ressemble un peu aux continuations et cette suite peut en effet être implémentée par une continuation. Dans le cas d'une récursion double, on peut donc inclure une branche comme suite de l'autre.

Ce n'est plus aussi simple lorsqu'il faut traiter `case-match`, car il faut aussi traiter l'échec, en examinant la clause suivante. On pourrait passer en paramètre la "suite" à inclure dans l'expression générée en cas d'échec : tout va bien pour les variables et les constantes mais la récursion double complique la structure puisqu'il n'est pas envisageable d'inclure deux fois la suite d'échec. Il faut donc gérer l'échec par un test explicite portant sur le résultat de la double récursion. Du coup, il est plutôt plus simple de traiter le `case-match` à plat, clause après clause, mais il faut gérer la réussite et l'échec par les valeurs retournées, avec précision.

Une dernière optimisation consiste à factoriser toutes les parties communes des motifs, pour éviter de dupliquer des tests.

EXERCICE 5.18. Définir la fonction `compile-clause` qui génère l'expression du code intermédiaire correspondant à une clause. □

EXERCICE 5.19. Définir la fonction `compile-match` qui génère l'expression correspondant à un `case-match` complet : la construction `:match` n'est plus nécessaire, mais il faut un `:let` pour allouer l'environnement nécessaire. □

Remarque 5.6. Le traitement des variables segment est beaucoup plus délicat car il nécessite des retours en arrière. Il ne sera pas abordé ici.

Remarque 5.7. La compilation du filtrage gagnerait probablement à se servir d'*échappements*.

5.3.2 Les flots et le retardement

Comme le retardement peut être défini simplement, en COMMON LISP, avec une macro `delay` et une fonction `force`, il n'est pas utile d'introduire une transformation particulière dans le langage intermédiaire.

6

Parcours d'arbre et analyse par cas

Quel que soit le traitement que l'on veut faire subir au code LISP — génération de code, évaluation (Chapitre 4) ou transformation dans le langage intermédiaire (Chapitre 5) —, la structure générale repose sur un parcours de l'arbre syntaxique LISP que l'on effectue à l'aide d'une *analyse par cas*. De plus, le principe de cette analyse par cas, c'est-à-dire des cas eux-mêmes, est indépendante du traitement spécifique que l'on désire réaliser. Bien entendu, la manipulation du langage intermédiaire — génération de code, ou évaluation directe (Chapitre 5) —, réclame une analyse par cas similaire, de même que l'interprétation de la machine virtuelle (Chapitre 8). Cependant, ces langages intermédiaires ont justement été conçus pour que les cas soient très simples, ce qui n'est pas le cas du langage LISP.

Ce chapitre détaille donc l'analyse par cas à appliquer au langage LISP indépendamment de tout objectif particulier. De façon générale, cette analyse par cas doit reproduire fidèlement la syntaxe des *expressions évaluables* du langage considéré, ici celle de LISP telle qu'elle est explicitée dans le Chapitre 2 du polycopié de LISP [Duc13a]. L'analyse par cas du langage intermédiaire s'appuiera sur la syntaxe de ses propres expressions évaluables (Section 5.1).

6.1 Principe de l'analyse du code LISP

L'analyse des expressions LISP ne s'intéresse qu'aux *expressions évaluables*, celles qui représentent un programme. Le principe de l'analyse par cas consiste à identifier le "cas" représenté par une telle expression et à *parcourir récursivement ses sous-expressions évaluables*.

6.1.1 Les cas

Les différents cas se distinguent

— par la *syntaxe* (atome, cellule, symbole, etc.) de l'expression ou d'éléments distingués de l'expression (par exemple, celui de la sous-expression qui a le rôle de fonction) ;

— mais aussi par la *sémantique* : par exemple, un symbole peut jouer un rôle fonctionnel de (vraie) fonction, locale ou globale, de macro ou de forme syntaxique ; il peut aussi jouer un rôle de variable.

La complexité de l'analyse par cas du code LISP résulte de cette dualité. La syntaxe est le critère premier, mais elle n'est pas assez discriminante et la sémantique est donc nécessaire pour discriminer ensuite plus finement. Par contraste, le langage intermédiaire est conçu pour que la syntaxe et la sémantique y soient en relation bijective.

Enfin, il est absolument indispensable d'identifier tous les cas d'erreur, qui sont nombreux : l'expression en argument de la fonction d'analyse est, par construction, une expression LISP bien formée, mais rien n'impose que ce soit une expression *évaluable* bien formée.

Interface fonctionnelle des symboles. La partie la plus délicate de l'analyse par cas consiste à discriminer les différentes catégories de fonctions (globale, locales, macros, formes syntaxiques, etc.). Cette discrimination repose sur l'interface fonctionnelle du type `symbol` :

— `fboundp` indique si un symbole a une définition de fonction,

— `macro-function` retourne la définition en tant que macro (l'usage est principalement booléen),

— et `special-form-p` dit si le symbole est une forme syntaxique.

La première fonction signale une exception si son argument n'est pas un symbole, les deux autres si le symbole argument n'a pas de définition de fonction. Voir le manuel COMMON LISP [Ste90] ou le polycopié de LISP [Duc13a], Section 4.3.1, pour plus de détails.

L'analyse présentée couvre le langage LISP tel qu'il est rendu accessible par l'environnement de développement qui respecte les spécifications du langage. Ces spécifications n'incluent pas la possibilité d'accéder au code d'une fonction. Le type `function` des valeurs fonctionnelles retournées par les fonctions `function` ou `symbol-function` ne présente aucune interface pour accéder au code de la fonction. Il n'est donc possible de manipuler le code des fonctions qu'au moment de leur définition par `defun`.

Si l'on veut manipuler ce code après la définition, par exemple pour faire de l'évaluation (Chapitres 4 et 5), mais aussi pour faire de l'*inlining* en génération de code, il est nécessaire de gérer "à la main" leur définition, sans passer pas l'évaluateur LISP sous jacent : c'est ce que l'on appelle les fonctions *méta-définies* (voir `get-defun`, Section 4.3.2).

Remarque 6.1. Les différents dialectes de LISP se différencieront essentiellement par cette interface des symboles.

6.1.2 Parcours d'arbre récursif

Une expression évaluable est un arbre de sous-expressions évaluables. Le parcours de ces sous-expressions doit donc se faire par une fonction récursive. Cependant, il ne s'agit pas d'une récursion d'arbre binaire sur les cellules de l'expression, mais d'une récursion d'arbre premier-fils/frère (voir polycopié de LISP [Duc13a], Section 3.1.).

Le principe de ces récursions repose sur deux fonctions :

— la première, `foo`, prend en paramètre une expression évaluable ;

— la seconde, `map-foo`, prend en paramètre une liste d'expressions évaluables.

Ces deux fonctions s'appellent récursivement l'une l'autre, `foo` appelant `map-foo`, et `map-foo` appelant `foo` sur son premier fils (le `car`, puis `map-foo` sur ses frères (le `cdr`). Il s'agit néanmoins d'une forme typique qui devra varier suivant les cas, en ce qui concerne `foo` : dans certains cas, `foo` appelle directement `foo` sur certaines des sous-expressions de la liste, ou ne présente même pas de récursion. En revanche, la structure de `map-foo` reste assez constante.

Dans le cas de l'analyse d'expressions évaluables LISP formées uniquement d'atomes et d'appels de vraies fonctions, `foo` appelle `map-foo` sur la liste d'arguments d'un appel de fonction et la structure récursive serait la suivante :

```
(defun foo (expr)
    (if (atom expr)
        <init>
      (map-foo (cdr expr))))

(defun map-foo (lexpr)
    (if (atom lexpr)
        <init>
     (<glue> (foo (car expr)) (map-foo (cdr expr)))))
```

Dans ce motif général, `<init>` et `<glue>` dépendent de l'objectif de la fonction `foo`. Si l'on veut compter, `<init>` pourra être 0 et `<glue>` sera +. Si l'on veut construire du code, `<init>` sera `nil` et `<glue>` sera une fonction de construction de liste comme `cons` ou `append`. Enfin, si l'on se contente de vérifier le code, `<init>` sera `nil` et `<glue>` sera `progn`.

Toute fonction de parcours de code et d'analyse par cas, ici nommée `case-analysis` doit donc être associée à une fonction `map-case-analysis` qui assure la récursion sur les listes d'expressions évaluables.

6.1.3 Version arborescente ou linéaire

Techniquement, il y a deux façons d'effectuer cette analyse. La première consiste à mimer directement l'arbre binaire de décision permettant d'identifier les cas, on obtient alors des `if` imbriqués, de profondeur

```
(defun CASE-ANALYSIS (expr env)
   (if (atom expr)
       (if (constantp expr)
           ...                                                    ; littéraux
         ;;  EXPR est un symbole
         (if (assoc expr env)
             ...                                                   ; variable
             (error "~s variable inconnue" expr)))
      ;;  EXPR est une cellule
      (if (consp (car expr))
          (if (eq 'lambda (caar expr))
              (...                                          ; λ-expressions
              (map-case-analysis (cdr expr) env)   ; ⇒ récursion sur arguments
              ...                                ; ⇒ construction environnement
              (map-case-analysis (cddar expr) env)
              ...)                      ; ⇒ récursion sur corps de la λ-fonction
             (error "une expression évaluable ne commence pas par (( : ~s"
                    expr))
        (if (not (symbolp (car expr)))
            (error "~s n'est pas un symbole" (car expr))
         ;;  (CAR EXPR) est un symbole
         ;;  ici s'insère le cas des fonctions et macros méta-définies
         (if (not (fboundp (car expr)))
             (error "~s fonction inconnue" (car expr))
           ;;  (CAR EXPR) est un symbole qui joue le rôle de fonction
           (if (macro-function (car expr))
               (case-analysis (macroexpand-1 expr))                ; macros
             (if (special-form-p (car expr))             ; formes syntaxiques
                 (case (car expr)       ; ⇒ récursion sur certains arguments
                     (if ...)
                     (quote ...)
                     (setf ...)
                     (function ...)
                     (defun ...)
                     ...
                     (t   ; pour rattraper ce qui n'est pas encore implémenté
                        (error "forme spéciale NYI ~s" (car expr))))
                ...                                          ; fonctions normales
                (map-case-analysis (cdr expr) env)   ; récursion sur arguments
                ...           ; ⇒ éventuellement construction d'environnement
                               ; ⇒ et analyse du corps de la fonction appelée
                )))))))
```

FIGURE 6.1 – Parcours d'arbre syntaxique LISP et analyse par cas dans une version arborescente

variable.

La figure 6.1 donne le squelette de la fonction `case-analysis` sous la forme arborescente de `if` imbriqués.

Une alternative consiste à linéariser les cas, pour obtenir un `cond` assez long, dont chaque clause identifie, pour l'essentiel, un cas individuel :

La figure 6.2 donne le squelette de la fonction `case-analysis` sous la forme linéaire d'un unique `cond`.

Que choisir ? C'est une question de style.

Chaque version a ses avantages et inconvénients. La version arborescente est plus efficace, l'arbre n'étant pas trop déséquilibré, mais la lisibilité du code est réduite et l'indentation nécessite une page de grand format. L'ajout d'un cas ne nécessite que l'identification de l'endroit où l'insérer : en fait, on insère un sous-cas dans un cas précis déjà déterminé.

En revanche, la version linéaire est plus lisible et s'indente sans problèmes. Mais les cas sont maintenant ordonnés : la condition de chaque cas contient implicitement la conjonction des négations de tous les cas précédents. L'ajout d'un nouveau cas doit donc être fait avec précision : trop tôt, il va absorber d'autres cas mais, trop tard, l'ajout sera inopérant.

La première chose à faire quand on ajoute un nouveau cas est de vérifier que l'on y passe bien — c'est assez facile, puisqu'il n'y a qu'un cas à vérifier. Mais cela ne suffit pas : il faut aussi vérifier que tous les autres cas, en particulier les cas suivants, sont toujours bien identifiés. Des tests de non régression ne sont pas inutiles.

6.1.4 Commentaires sur le traitement des cas

— Pour les atomes, la fonction `constantp` permet de discriminer les symboles qui peuvent jouer le rôle de variable des autres atomes, littéraux ou symboles constants.

— Une *variable* n'est qu'un symbole en situation de variable : c'est sa présence dans un *environnement* qui fait la différence ; toute fonction d'analyse nécessite donc un paramètre environnement (`env`) qui contient, sous une forme ou sous une autre, l'ensemble des variables connues.

— Dans le cas général, un environnement est une liste d'association qui associe à chaque variable une information (valeur, position, etc.) et où l'on pourra chercher (par la fonction `assoc`) le symbole rencontré en position de variable. Dans certains cas, un ensemble (c'est-à-dire une liste) de variables suffit, et la recherche utilisera `member`, `find`, `position` ou autre.

— Pour la gestion des erreurs, on utilisera `warn` ou `error` (voir polycopié de LISP et manuel) suivant le traitement effectué : `error` est nécessaire à l'exécution, mais `warn` est suffisant pour toute analyse statique.

— Dans le cas des macros, le traitement par défaut consiste à expanser l'expression, par `macroexpand-1` (voir cette fonction dans le polycopié de LISP [Duc13a]), puis à appliquer récursivement l'analyse sur le résultat de l'expansion. Bien entendu, le traitement désiré peut nécessiter un comportement différent.

— Pour simplifier, les fonctions locales définies par `flet` ou `labels` n'ont pas été intégrées : voir le chapitre 4.

6.1.5 Exercices

EXERCICE 6.1. Compléter la fonction `case-analysis` pour traiter tous les cas de récursion et parcourir toutes les sous-expressions évaluables. □

EXERCICE 6.2. Compléter la fonction `case-analysis` pour construire les environnements dans le cas des λ-expressions et de `defun` : on pourra se limiter à une liste de variables. □

EXERCICE 6.3. Pour donner du corps à l'analyse par cas, compléter la fonction `case-analysis` pour qu'elle

— imprime le cas courant,

— imprime la profondeur de l'expression courante,

```
(defun CASE-ANALYSIS (expr env)
   (cond ((and (atom expr) (constantp expr))                          ; littéraux
          ...)
         ((symbolp expr)                                              ; symboles
          (if (assoc expr env)
              ...                                                     ; variable
              (error "~s variable inconnue" expr)))
        ;;  EXPR est maintenant une cellule
         ((and (consp (car expr))
               (eq 'lambda (caar expr)))                             ; λ-expressions
          ...                                          ; ⇒ récursion sur arguments
          (map-case-analysis (cdr expr) env)    ; ⇒ construction environnement
          (map-case-analysis (cddar expr) env)
          ...)                            ; ⇒ récursion sur corps de la λ-fonction
         ((not (symbolp (car expr)))
          (error "~s n'est pas un symbole" (car expr)))
        ;;  (CAR EXPR) est maintenant un symbole
        ;;  ici s'insère le cas des fonctions et macros méta-définies
         ((not (fboundp (car expr)))
          (error "~s fonction inconnue" (car expr)))
        ;;  (CAR EXPR) est maintenant un symbole qui joue le rôle de fonction
         ((macro-function (car expr))
          (case-analysis (macroexpand-1 expr)))                                ; macros
         ((not (special-form-p (car expr)))                ; fonctions normales
          ...
          (map-case-analysis (cdr expr) env)       ; ⇒ récursion sur arguments
          ...)                  ; ⇒ éventuellement construction d'environnement
                                 ; ⇒ et analyse du corps de la fonction appelée
        ;;  (CAR EXPR) est maintenant une forme spéciale
        ;;  ⇒  récursion sur certains arguments
         ((eq 'if (car expr))
          ...)
         ((eq 'quote (car expr))
          ...)
         ((eq 'setf (car expr))
          ...)
         ((eq 'function (car expr))
          ...)
         ((eq 'defun (car expr))
          ...)
         (t                  ; pour rattraper ce qui n'est pas encore implémenté
            (error "forme spéciale NYI ~s" (car expr))
         ))
```

FIGURE 6.2 – Parcours d'arbre syntaxique LISP et analyse par cas dans une version linéaire

— retourne le nombre de sous-expressions évaluables.

□

EXERCICE 6.4. Dans le contexte de de l'exercice 6.3, définir la fonction `map-case-analysis` qui prend en paramètre une liste d'expressions évaluables et leur applique récursivement l'analyse par cas. □

EXERCICE 6.5. Définir la fonction `free-variables` qui prend en paramètre une expression évaluable et retourne la liste de ses *variables libres*. □

EXERCICE 6.6. Définir la fonction `recursionp` qui prend en paramètre une expression `defun` et retourne vrai si la fonction ainsi définie est récursive. Pour simplifier, on commencera par ne pas considérer les récursions indirectes. □

EXERCICE 6.7. Définir la fonction `irecursionp` qui Dans un deuxième temps, on traitera aussi les récursions indirectes, en faisant attention aux boucles infinies. Que faut-il passer en argument de `case-analysis` ? □

EXERCICE 6.8. Définir la fonction `tail-recursion-p` qui prend en paramètre une expression `defun` et retourne vrai si la fonction ainsi définie est récursive terminale. □

EXERCICE 6.9. Définir la fonction `check-parameter` qui prend en paramètre une expression évaluable et vérifie que les appels de fonctions qu'elle contient ont un nombre correct d'arguments. On supposera que les fonctions *méta-définies* sont déclarées par la fonction `get-defun` (Chapitre 4) et on intégrera le cas dans `case-analysis`. En revanche, on ne traitera pas le cas des appels de fonctions prédéfinies. □

EXERCICE 6.10. Compléter la fonction `case-analysis` pour traiter le cas des fonctions locales (`flet` et `labels`), en analysant le corps des fonctions locales et en identifiant les cas d'appels de fonctions locales. Que faut-il lui passer en paramètre ? Utiliser cette version étendue dans les questions précédentes. □

EXERCICE 6.11. Définir la fonction `check-labels` qui prend en paramètre une expression évaluable `labels` et détermine si un `flet` peut être utilisé à la place. □

6.1.6 Application au *pretty-printer*

Traditionnellement, le *pretty printer* est une fonction LISP qui imprime les expressions LISP, en général évaluables, en les indentant dans le « bon style ». Le principe général est donc une fonction récursive à 2 paramètres, l'expression à imprimer et la position (retrait) dans la ligne à partir duquel l'imprimer. Chaque cas de l'analyse a ses propres règles.

L'indentation recouvre en fait 2 notions, le retrait après chaque passage à la ligne, et le passage à la ligne lui-même, qui dépend de la longueur imprimée de chaque expression : en effet, on ne passe à la ligne que si l'expression ne tient pas dans ce qui reste de ligne.

EXERCICE 6.12. Définir la fonction `expr-print-length` qui prend une expression LISP en paramètre et retourne le nombre de caractères de son impression si l'on ne passe pas à la ligne. On utilisera la fonction `format` qui permet d'imprimer dans une `string`. □

EXERCICE 6.13. Définir la fonction `expr-pretty-print` qui prend une expression LISP en paramètre, plus un paramètre optionnel de largeur de ligne, et qui imprime et indente l'expression sur le canal de sortie courant, en ne passant à la ligne qu'en cas de besoin. □

6.2 Application au code intermédiaire

Le langage intermédiaire est, comme LISP lui-même, un langage imbriqué : son parcours repose donc sur une fonction d'analyse récursive assez similaire à celle de LISP. Cependant, le langage intermédiaire a été spécifié pour que chaque cas soit discriminé par son étiquette, suivant la syntaxe décrite dans la

Section 5.1. On peut vérifier simplement à quel point la transformation dans le code intermédiaire simplifie l'analyse par cas.

EXERCICE 6.14. Définir la fonction d'analyse par cas sur le langage intermédiaire (Chapitre 5). Que faut-il lui passer comme paramètre ? Reprendre les exercices 6.4 et 6.1 dans ce contexte.
Les exercices 6.2 et 6.5 s'appliquent-ils ? □

EXERCICE 6.15. Traiter ensuite sur le langage intermédiaire les exercices 6.6 et 6.8. Que faut-il leur passer comme paramètre ? □

EXERCICE 6.16. Remplacer dans le code intermédiaire les étiquettes (`keyword`) par des entiers. Définir la fonction d'analyse par cas comme un arbre binaire équilibré d'inégalités numériques. Analyser la performance moyenne des deux versions sous l'hypothèse que tous les cas sont équiprobables. □

Voir aussi l'exercice 5.7 sur le profilage.

6.3 Application aux langages de machine virtuelle

En revanche, les langages de machine virtuelle (voir Chapitres 7 et 9) sont des langages linéaires et non plus imbriqués : le parcours n'est donc plus récursif et se simplifie d'autant.

7

Principe et langage de la machine virtuelle à registres

La machine virtuelle que nous utilisons représente un modèle très simplifié d'un processeur réel. Si son principe est donc assez précisément déterminé, nous avons une grande liberté dans les détails de sa spécification. Le schéma qui est présenté ici peut être adapté de très nombreuses façons.

7.1 Eléments de la machine virtuelle

La machine virtuelle (VM) dispose d'une mémoire (finie, mais suffisamment grande pour exécuter sans problème les programmes cible) et de trois registres généraux `R0`, `R1` et `R2`. La mémoire est un ensemble fini de N cellules (ou mots). La première cellule a par définition le numéro 0, la dernière le numéro $N-1$. On parlera plutôt d'adresse mémoire que de numéro de cellule. Afin de coller à une réalité simplifiée, nous distinguons d'une part le microprocesseur qui va interpréter le code et, d'autre part, la mémoire.

Outre les 3 registres généraux, la machine possède plusieurs registres dédiés. La pile d'exécution est gérée dans la mémoire. La pile commence à l'adresse indiquée dans le registre spécial `BP` (*base pointer*). A priori, on ne touchera jamais à ce registre qui est initialisé une fois pour toute au démarrage de la MV. Le sommet de pile est indiqué dans le registre spécial `SP` (*stack pointer*). La pile est vide quand `SP = BP`. On a toujours `SP` $\geq$ `BP` (notre pile est montante). Le compteur de programme (ou compteur ordinal) est contenu dans un registre spécial `PC` (*program counter*).

Dans la mémoire, une partie est alloué au stockage des programmes en cours d'exécution, une partie est allouée à la pile d'exécution, une partie au tas qui contient des structures de données. Enfin, une petite partie (dans la même zone que les programmes) aux variables globales du système.

Pour l'instant, nous avons donc les registres suivants : `R0`, `R1`, `R2`, `BP`, `SP`, `PC`. Nous introduirons dans la suite, un pointeur de cadre (*frame pointer*), `FP`, qui sert à définir des blocs de pile pour la structurer et faciliter les accès. Pour les comparaisons et les sauts conditionnels, on dispose de trois registres booléens (1 bit) appelés drapeaux (*flag*) : `FLT`, `FEQ`, `FGT` (pour drapeau plus petit, drapeau égal, drapeau plus grand).

En toute généralité, et pour être plus fidèle aux architecture matérielles nous pourrions décider d'avoir en plus des trois registres booléens, 8 registres généraux R0,... R7 dont certains seraient attribués aux rôles particuliers définis ci-dessus.

7.2 Instructions

Les instructions ont entre 0 et 2 opérandes. En général plusieurs adressages (normal, direct, indexé) sont possibles. Par souci d'homogénéité avec ce cours, elles sont présentées dans une syntaxe parenthésée à la LISP, sans distinction majuscule-minuscule.

Le tableau 7.1 récapitule les différentes opérations de la machine virtuelle.

TABLE 7.1 – Opérations de la machine virtuelle à registres

`(LOAD <src> <dest>)`	chargement de mémoire à registre
`(STORE <src> <dest>`	chargement de registre à mémoire
`(MOVE <src> <dest>)`	mouvement de registre à registre
`(ADD <src> <dest>)`	addition
`(SUB <src> <dest>)`	soustraction
`(MUL <src> <dest>)`	multiplication
`(DIV <src> <dest>)`	division
`(INCR <dest>)`	incrément
`(DECR <dest>)`	décrément
`(PUSH <src>`	empiler
`(POP <dest>`	dépiler
`(LABEL <label>)`	déclaration d'étiquette
`(JMP <label>)`	saut inconditionnel à une étiquette
`(JSR <label>)`	saut avec retour
`(RTN)`	retour
`(CMP <src1> <src2>)`	comparaison
`(JGT <label>)`	saut si plus grand
`(JGE <label>)`	saut si plus grand ou égal
`(JLT <label>)`	saut si plus petit
`(JLE <label>)`	saut si plus petit ou égal
`(JEQ <label>`	saut si égal
`(JNE <label>`	saut si différent
`(TEST <src>)`	comparaison à `NIL`
`(JTRUE <label>)`	saut si non-`NIL`
`(JNIL <label>)`	saut si `NIL`
`(NOP)`	rien
`(HALT)`	arrêt

7.2.1 Accès mémoire

`LOAD` et `STORE` permettent d'effectuer des échanges entre le processeur et la mémoire. Comme on prend toujours le point de vue du microprocesseur (la mémoire est extérieure), `LOAD` permet de charger un registre depuis la mémoire, `STORE` fait l'inverse, c'est-à-dire écrire dans la mémoire une valeur contenue dans un registre.

`(LOAD <src> <dest>)` charge le contenu de l'adresse `<src>` en mémoire dans le registre `<dest>`. `<src>` est toujours interprété comme une adresse en mémoire et `<dest>` est toujours un registre.

Par exemple : `(LOAD 500 R0)` charge le contenu de l'adresse `500` dans le registre `R0`.

Il est aussi possible d'avoir comme source un registre, auquel cas le contenu du registre sera supposé contenir l'adresse. Par exemple : `(LOAD R1 R0)` charge le contenu de l'adresse contenue dans `R1` dans le registre `R0`

Un mode indexé (décalage) est disponible. Par exemple, `(LOAD (+ R1 2) R0)` charge le contenu de l'adresse contenue dans `R1` plus `2`. Par exemple si `R1` contient `500`, alors le contenu de l'adresse `502` est chargé dans `R0`. La valeur de l'index (ici 2) doit être un entier relatif et ne peut être un registre.

En sens inverse, `STORE` est l'instruction permettant d'écrire en mémoire depuis les registres.

Par exemple `(STORE R1 500)` écrit la valeur contenue dans `R1` à l'adresse `500`. Il est possible d'utiliser un registre comme registre d'adresse. Par exemple : `(STORE R1 R0)` écrit la valeur contenue dans `R1` en mémoire à l'adresse contenue dans `R0`. Le mode indexé est disponible pour la destination. `(STORE R1 (+ RO 3))` écrit la valeur contenue dans `R1` en mémoire à l'adresse contenue `R0` augmentée de `3`. Le mode direct existe aussi pour `STORE` : `(STORE (:CONST 3) R2)` écrit la valeur `3` en mémoire à l'adresse contenue dans `R2`.

Remarque sur :**CONST.** Le mot-clé :`CONST` peu paraître inutile dans le cas d'un entier. Cependant, cette machine virtuelle est le langage cible d'un compilateur LISP et elle doit pouvoir gérer les différents types de littéraux usuels en LISP, en particulier les symboles. La compilation du compilateur nécessitera de

compiler des littéraux désignant les registres, R0 par exemple. En LISP la distinction entre une variable et une valeur s'effectue avec la quote, x versus (quote x). Dans le langage de la machine virtuelle, c'est :CONST qui joue ce rôle : (:CONST R0) désigne le symbole R0, alors que R0 désigne la valeur du registre R0. Mais cela n'est valable que pour les opérandes qui peuvent être des valeurs immédiates, comme cet opérande de STORE.

En pratique, puisqu'on est maître de ce langage, on peut remplacer :CONST par QUOTE. Il suffira alors d'écrire 'R0 qui sera lu (QUOTE R0).

Remarque sur les adresses. En pratique, il est illusoire de chercher à programmer en manipulant directement des adresses, et l'on utilise à la place des étiquettes (voir Section 7.2.7). Pour distinguer les étiquettes des registres, on utilisera le mot-clé :REF. Ainsi, (STORE (:CONST 3) (:REF TOTO)) écrit la valeur 3 à l'adresse désignée par l'étiquette TOTO. Noter que la relation entre une étiquette et une adresse est statique (elle est déterminée à l'édition de liens, cf Chapitre 8) et doit être univoque.

Remarque sur les accès mémoire. On ne cite ici les accès mémoire que pour le principe. En effet, un programme fait des accès mémoire pour la pile, le tas (*heap*) et les variables statiques (sans parler du code lui-même). Notre simulation ne s'étend pas à la gestion mémoire du tas, qui sera faite par CLISP (eval). Comme la pile a ses propres instructions, des accès mémoire explicites ne seront requis que pour les variables statiques, qui sont rares car notre style de programmation ne les encourage pas. On risque juste de les rencontrer si l'on souhaite implémenter un cas particulier de fermetures, celui de l'exemple du compteur (voir polycopié de LISP).

7.2.2 De registre à registre

MOVE permet de transférer des valeurs entre registres (MOVE <src> <dest>) charge <src> dans <dest>. Les différents modes d'adressage présentés ci-dessus peuvent être utilisés. Nous avons le mode direct (on charge une constante) et le mode normal. (MOVE (:CONST 50) R0), en mode direct, charge la constante 50 dans R0. (MOVE R0 R1) recopie le contenu de R0 dans R1.

Contrairement à de nombreux langages d'assemblage réels, MOVE ne peut pas faire d'accès mémoire : il faut passer par LOAD et STORE.

Pour faire une addition de deux valeurs contenues en mémoire aux adresses x et y, il faut donc transférer ces valeurs dans des registres, faire l'addition, et transférer le résultat en mémoire vers par exemple l'adresse z. Ce qui devrait ressembler à quelque chose comme :

```
(LOAD x R0)
(LOAD y R1)
(ADD R0 R1)
(STORE R1 z)
```

7.2.3 Instruction arithmétiques

Dans toutes ces instructions, <dest> est un registre. <src> peut être une valeur en mode direct ou bien un registre en mode normal ou indexé.

(ADD <src> <dest>) ajoute <src> à <dest>, le résultat se trouve dans <dest>. Par exemple : (ADD R0 R1) écrit la valeur de l'addition entre les contenus de R0 et R1 dans R1. En mode direct, nous avons par exemple (ADD (:CONST 2) R0) qui ajoute 2 à la valeur contenue dans R0.

(SUB <src> <dest>) soustrait <src> à <dest>, le résultat se trouve dans <dest>. Par exemple, (SUB (:CONST 5) R2) permet de soustraire 5 au contenu de R2.

(MULT <src> <dest>) multiplie <src> à <dest>, le résultat se trouve dans <dest>. Par exemple, (MULT R2 R0) écrit dans R0 le produit entre les contenus de R2 et R0. L'instruction (MULT (:CONST 7) R3) écrit dans R3 le produit du contenu de R3 par 7.

(DIV <src> <dest>) divise <dest> par <src>, le résultat se trouve dans <dest>. Par exemple, (DIV (:CONST 3) R2) permet de diviser le contenu de R2 par 3.

`(INCR <dest>)` incrémente `<dest>` de `1`. `<dest>` est un registre. Par exemple, `(INCR R0)` augmente la valeur contenue dans `R0` de `1`. Cette instruction est équivalente à `(ADD (:CONST 1) R0)`, mais elle potentiellement plus efficace. On remarquera également qu'il est plus efficace de faire : `(ADD (:CONST 1) R0)` que de faire `(MOVE (:CONST 1) R1) (ADD R1 R0)`

`(DECR <dest>)` Décrémente `<dest>` de `1`. `<dest>` est un registre. Cette instruction est équivalente à `(SUB (:CONST 1) <dest>)` qui suit, mais elle est potentiellement plus efficace.

7.2.4 Instructions de pile

Les instructions de pile manipulent le registre de sommet de pile `SP`. Avec les instructions précédentes, nous serions capables d'écrire les instructions de pile. Toutefois, celles-ci peuvent raisonnablement être implémentées directement (code plus efficace et surtout plus clair).

`(PUSH <src>)` pousse sur la pile le contenu de `<src>`. Par exemple, `(PUSH R0)` empile le contenu de `R0`. Par exemple, `(PUSH (:CONST 3))` empile la valeur `3`. L'instruction, `(PUSH (+ R0 2))` pousse le contenu de `R0` augmenté de `2` sur la pile. Le code qui suit est globalement équivalent à `(PUSH <src>)` en mode normal quoique potentiellement moins efficace :

```
(INCR  SP)                  // on incrémente SP
(STORE  <src> SP)          // on écrit <src> à la nouvelle adresse de SP
```

Par exemple, `(PUSH R0)` empile le contenu de `R0` en sommet de pile. C'est équivalent à : `(INCR SP) (STORE R0 SP)`. Empiler la valeur `1` sur la pile s'écrit `(PUSH (:CONST 1))` et est équivalent à : `(INCR SP) (STORE (:CONST 1) SP)`.

Pour dépiler, on utilise l'instruction `POP`. `(POP <dest>)` dépile le sommet de pile, et met l'information dans `<dest>`. `(POP <dest>)` est équivalent à :

```
(LOAD SP <dest>)      //  on met le contenu de l'adresse de SP dans <dest>
(DECR SP)             //  on décrémente SP
```

7.2.5 Accès dans la pile

Si on veut lire (ou écrire) un élément de la pile (mais sans dépiler pour autant), on se basera sur le sommet de pile. Par exemple, pour lire de 2-ième élément de la pile (celui sous le sommet de pile) et le mettre dans `R0` on écrira : `(LOAD (- SP 1) R0)`

Si on veut dépiler *violemment* plusieurs éléments de la pile, il suffit de modifier SP. Par exemple, pour dépiler `10` éléments, nous pouvons écrire : `(MOVE (:CONST 10) R1) (SUB R1 SP)`

7.2.6 Généralisation de la pile

Dans ce qui précède, nous avons supposé qu'il existait le registre spécial `SP`. En pratique, on utilise à cette fin un registre général, par exemple `R7`, comme registre de sommet de pile. Les fonctions de manipulation de pile sont généralisées comme suit : `(PUSH <src> <dest>)` où `<dest>` est le registre utilisé pour `SP` Par exemple, `(PUSH R0 R7)` empile le contenu de `R0` dans la pile gérée avec `R7`. Ce code est équivalent à : `(INCR R7) (STORE R0 R7)` Si `R7` n'est utilisé que pour la pile, le code généralisé (avec donc `(PUSH x R7)` plutôt que `(PUSH x)`) reste tout aussi clair, et colle mieux à la réalité matérielle.

En sens inverse, `(POP <src> <dest>)` dépilera le registre `<src>` dans `<dest>`.

De façon similaire, le registre `R6` est souvent utilisé comme registre spécial de pointeur de cadre (`FP`, *frame pointer*).

7.2.7 Adresses, étiquettes, et instructions de saut inconditionnel

Dans le code de la machine virtuelle, une adresse dans le code est une valeur entière interprétée comme une adresse en mémoire. Cette adresse doit être une adresse où se trouve du code exécutable (sinon les choses risquent de mal se passer). Quand on écrit du code assembleur, on n'écrit cependant pas des adresses directement (on ne les comnnait pas en fait à ce moment là), mais on utilise à la place des « étiquettes » pour

nommer les adresses. On peut utiliser différents formats pour les étiquettes mais, en pratique, nous avons intérêt à utiliser des types simples du langage dans lequel nous travaillons LISP : en gros, des symboles ou des étiquettes.

Noter que chacun des deux peut poser un problème d'ambiguïté, puisque le symbole pourrait être le nom d'un registre, et qu'un entier pourrait être la valeur d'une adresse. Nous trancherons cette ambiguïté en excluant la possibilité de mentionner directement une adresse.

Avec l'instruction `JMP` (cf plus bas) on va écrire quelque chose comme `(JMP DEBUT-DE-FONCTION)`. Lors du chargement du code assembleur dans la mémoire, le chargeur va se charger de remplacer les étiquettes par les adresses en mémoire où elles se trouvent (cf. Chapitre 8).

Pour déclarer une étiquette, nous avons la (pseudo-) instruction suivante : `(LABEL <nom-d'étiquette>)`. Par exemple, `(LABEL DEBUT-DE-FONCTION)`, définit et positionne dans le code l'étiquette `DEBUT-DE-FONCTION`. Il est alors possible, dans le code, d'y faire référence.

`(JMP <label>)` saute à l'adresse `<label>`. Un saut consiste à modifier le compteur de programme, et `(MOVE <label> PC)` est équivalent à `(JMP <label>)`. En pratique on ne touche jamais directement le registre `PC`, il y a des instructions pour cela.

Supposons que nous avons le code suivant :

```
<bloc instruction A>
(JMP KIKI)
(LABEL COUCOU)
<bloc instruction B>
(JMP BYE)
(LABEL KIKI)
<bloc instruction C>
(JMP COUCOU)
(LABEL BYE)
<bloc instruction D>
```

Lors de l'exécution du code, le bloc d'instructions A est exécuté, puis on saute à l'étiquette `KIKI` et le bloc d'instructions C est exécuté. Puis on saute, à l'étiquette `COUCOU`, et on exécute le bloc B. Enfin on saute à l'étiquette `BYE` et le bloc D est exécuté. L'instruction `JMP` est mieux connue sous le nom de `GOTO` en BASIC ou FORTRAN. L'usage inconsidéré de cette instruction aboutit en général à du code en "plat de nouilles" qui est considéré comme une mauvaise pratique particulièrement délétère, et les langages évolués modernes, dits structurés, se passent en général de cette structure de contrôle. Dans l'exemple précédent, avant l'exécution, lors du chargement du code en mémoire (Section 8.3), les étiquettes sont remplacées par les adresses correspondantes :

```
1000    <bloc instruction A>
1020    (JMP 1053)
1021    <bloc instruction B>
1052    (JMP 1105)
1053    <bloc instruction C>
1104    (JMP 1021)
1105    <bloc instruction D>
```

Remarque sur le code inaccessible. L'instruction qui suit un `JMP` — mais aussi `RTN` et `HALT` — est inaccessible en séquence. Elle ne peut donc être que `LABEL`.

7.2.8 Saut avec retour

`(JSR <label>)` saute à l'adresse `<label>` en empilant l'adresse de retour sur la pile. L'adresse de retour est l'adresse immédiatement suivant le `JSR`, en d'autre terme l'adresse `PC+1`. On revient à l'instruction qui suit le saut avec l'instruction `RTN` Le code qui suit est équivalent à `(JSR <label>)` :

```
(PUSH (+ PC 1))     // on pousse l'adresse courante sur la pile
(JMP <label>)       // on effectue le saut
```

L'instruction (RTN) saute à l'adresse contenue en sommet de pile, et la dépile. Le code qui suit est strictement équivalent :

```
(LOAD SP R0)
(DECR SP)
(JMP (:REF R0))
```

On charge le contenu du sommet de pile dans R0, on décrémente SP et on saute à l'adresse contenue dans R0.

Ce couple d'instructions JSR et RTN permet d'effectuer des appels de fonctions et d'en revenir. Historiquement, on parlait de *subroutine*, d'où le nom de JSR (*Jump to SubRoutine*).

7.2.9 Instruction de comparaison

(CMP <a> <b>) compare le contenu du registre <a> à celui du registre <b> et positionne les registres de drapeaux FLT, FEQ et FGT (dans cet ordre). Le mode direct est possible pour <a> comme pour <b>. Par exemple (CMP (:CONST 1) (:CONST 2)) positionne les drapeaux à 100 (1 veut dire vrai et 0 faux). Comme 1 est plus petit que 2, alors FLT est mis à vrai est les autres sont mis à faux. Si R0 contient 2, R1 contient 4 et R2 contient également 4 alors :

```
(CMP R0 R1) => 100
(CMP R1 R0) => 001
(CMP R2 R2) => 010
(CMP R0 (:CONST 1)) => 001
(CMP (:CONST 4) R2) => 010
```

L'usage est de faire suivre une instruction CMP par un saut conditionnel. Toutefois, il est possible d'accéder aux registres de drapeau, comme tout autre registre. La convention retenue est que 1 vaut vrai et que 0 vaut faux. Par exemple :

```
(MOVE (:CONST 8) R0)
(DIV 2 R0)
(CMP (:CONST 4) R0)
(MOVE FEQ R2)
```

à l'issue de l'exécution R2 vaut 1. En effet, R0 vaut 4.

On peut aussi compléter l'opération binaire CMP par une opération unaire TEST telle que (TEST <src>) compare <src> à 0 ou, dans le contexte de LISP, à NIL. Dans ce dernier cas, un drapeau FNIL est positionné.

7.2.10 Les sauts conditionnels

Les instructions (JLT <label>), (JLE <label>), (JGT <label>), (JGE <label>), (JEQ <label>), (JNE <label>) sautent à l'adresse d'étiquette <label> si les drapeaux de condition sont dans l'état attendu. La correspondance entre les instructions et les drapeaux est la suivante :

```
JLT  100  (plus petit, Lesser Than)
JLE  110  (plus petit ou égal, Lesser or Equal),
JGT  001  (plus grand, Greater Than),
JGE  011  (plus grand ou égal, Greater or Equal),
JEQ  010  (égal, EQual),
JNE  101  (non égal, Not Equal).
```

Dans le cas contraire, l'exécution se continue en séquence, c'est-à-dire que registre PC est simplement incrémenté de 1.

Techniquement, pour savoir si le saut doit être fait, on effectue un AND bit-à-bit entre le schéma (ci-dessus) et l'état des drapeaux. Si le résultat est différent de 000 alors le saut est effectué. Les instructions de saut n'ont de sens que si elles sont précédées par un CMP. L'état initial des drapeaux est indéfini. Donc

faire un saut conditionnel sans qu'un `CMP` n'ait été fait auparavant sera aléatoire. Par exemple : On a `001` comme états de drapeaux et le test est `JGE` : `001 and 011 = 001` (saut effectué). On a `001` comme états de drapeaux et le test est `JLE` : `001 and 110 = 000` (saut non effectué).

Toujours dans le contexte de LISP, les opérations `JTRUE` et `JNIL` sont analogues mais reposent sur le drapeau `FNIL` positionné par l'opération unaire `TEST`.

7.2.11 Instructions diverses

`(NOP)` est une instruction vide, qui ne fait rien. Cette instruction est parfaitement inutile, à moins de vouloir modifier le code en mémoire après son chargement (dans le cas d'un code auto-modificateur par exemple).

`(HALT)` termine l'exécution de la machine virtuelle. Cette instruction doit impérativement terminer la séquence de code sur laquelle on lance la machine virtuelle, sinon elle finira son exécution sur du code non valide.

7.2.12 Synthèse sur les opérandes

De façon générale, les opérandes des différentes instructions peuvent être de plusieurs types :

donnée : ce sont des opérandes nommés `<src>` dans les descriptions qui précèdent ; il peut s'agit de littéraux, sous la forme `(:CONST <lit>)`, où `:CONST` est obligatoire pour désambiguïser le cas où `<lit>` est un symbole ; il peut s'agir aussi de registres, dont on désigne le contenu, en mode normal ou indexé.

registre en tant que destination : cet opérande `<dest>` ne peut être qu'un registre dont le contenu va être écrasé par l'instruction.

adresse de donnée en mémoire : dans un `STORE` ou `LOAD`, l'adresse mémoire considérée peut être désignée par un entier (pas très pratique !), ou un registre dont l'adresse est le contenu. On peut aussi utiliser une étiquette, mais il faut alors être capable de distinguer le contenu du registre `R0` de l'adresse d'étiquette `R0`. On utilise `(:REF R0)` dans le dernier cas.

adresse d'instruction en mémoire : dans les instructions de saut, l'adresse où sauter peut être un entier (même remarque), un registre qui contient l'entrée, ou une étiquette. Il faut alors être capable de distinguer l'adresse contenue dans le registre `R0` de l'adresse d'étiquette `R0`. On utilise `(:REF R0)` dans le premier cas.

On remarque qu'il y a 3 cas d'ambiguïté possibles, pour lesquels on utilise une syntaxe parenthésée :

— entre registre et littéral symbole : on utilise `:CONST` (ou `QUOTE`) ;
— entre registre et étiquette pour une adresse de donnée : on annote les étiquettes avec `:REF` car c'est le cas le moins fréquent ;
— entre registre et étiquette pour une adresse d'instruction : cette fois-ci, on annote le registre, pas l'étiquette, mais pour la même raison.

8

Simulation de la machine virtuelle à registres

Ce chapitre décrit l'implémentation en LISP de la machine virtuelle à registres (Chapitre 7) qui sert de langage cible à la génération de code. Cette implémentation va permettre de tester la correction du code compilé en l'exécutant.

8.1 Code de la machine virtuelle

Le langage de la machine virtuelle et ses différentes conventions ont été définis au Chapitre 7, dans une syntaxe qui le rend directement lisible (par `read`) dans un environnement LISP. On conservera donc les conventions sur la syntaxe des instructions, et on y rajoutera juste celle-ci : une unité de code VM est une liste d'instructions VM.

8.2 Structure de la machine virtuelle

La machine virtuelle peut se décomposer en quelques éléments pour lesquels il faudra une implémentation : principalement les registres et la mémoire, cette dernière servant à la fois pour la pile, le code, le tas et les variables globales.

Une fonction doit permettre de créer et d'initialiser une machine virtuelle suivant divers paramètres (taille de mémoire, liste de registres, etc.).

8.2.1 La mémoire

Sur des architectures banalisées, la pile est dans le même espace mémoire que le code et le tas. Un espace est alloué pour l'ensemble du processus (ici la machine virtuelle) : le code et la pile occupent les deux extrémités de la zone mémoire, en croissant vers l'intérieur et le tas est au milieu, après la zone occupée par le code. Bien entendu, le débordement de la pile se traduit par l'écrasement successif du tas puis de la zone du code. Enfin, les variables statiques (souvent globales, mais non dynamiques) sont en général implémentées avec le code.

En pratique on n'aura pas besoin du tas puisque l'on se reposera sur la gestion mémoire de LISP.

On utilisera donc pour la mémoire un tableau de grande taille (voir les fonctions `make-array` et `aref`).

8.2.2 Les registres

Pour les registres, on a l'embarras du choix :

— il est possible de les mettre en mémoire, mais c'est une solution peu réaliste ;

— on peut utiliser des variables statiques mais ce n'est pas très propre ;

— les propriétés des symboles constituent une solution élégante et simple (voir la fonction `getprop`) ;

— une structure de donnée particulière (liste d'association, structure ou objet) est aussi possible.

8.2.3 Structure réentrante

L'idéal serait que la machine virtuelle soit réentrante, c'est-à-dire qu'il soit possible de compiler le code d'interprétation de la machine virtuelle pour faire interpréter la machine virtuelle par du code compilé et interprété par la machine virtuelle.

Il faut donc pouvoir faire coexister 2 machines virtuelles (ou plus) complètement indépendantes : une fois que c'est fait, il n'y a pas de raison que la machine virtuelle ne puisse pas s'interpréter elle-même.

8.3 Reliure et chargement

La machine virtuelle étant créée et le code disponible, il faut charger ce code avant de pouvoir l'exécuter. Le chargement consiste en 2 opérations principales :

— le chargement proprement dit ;
— la résolution des adresses.

8.3.1 Chargement proprement dit

Le chargement proprement dit consiste à mettre la séquence d'instructions dans la zone mémoire attribuée au code. Au cours de ce chargement diverses opérations peuvent être effectuées sur le code, en particulier la résolution d'adresses.

Le chargement se fait par adresse croissante ou décroissante suivant que la zone du code est au début ou à la fin de la zone mémoire. Quel que soit ce premier choix, le chargement du code peut se faire à l'endroit ou à l'envers, ce qui peut avoir une incidence sur le lancement de la machine virtuelle. Au total, lors de l'exécution du code en séquences, incrémenter le compteur ordinal pourra vouloir dire le décrémenter.

Parmi les choix de spécification, il faut déterminer si le chargement se fait en une seule fois ou peut se faire en plusieurs fois. La deuxième solution, plus générale, n'est guère plus compliquée.

Il faut enfin traiter le cas du progamme principal (`main`) qui déterminera l'adresse de lancement de la machine. Deux solutions sont possibles :

— avoir une étiquette convenue pour désigner l'adresse de lancement, par exemple `main` ;
— considérer que l'adresse de la dernière instruction chargée est l'adresse de lancement : cela suppose que le code soit chargé à l'envers.

Remarque 8.1. Le chargement d'une séquence de code consiste en une itération sur les instructions de la séquence (une liste). Dans le bon style LISP, il faudrait utiliser une fonction récursive. Cependant la séquence de code peut être très longue et une fonction récursive non optimisée casserait la pile. Il faudrait donc impérativement que la fonction soit en récursion terminale, et pour que l'optimisation soit faite par CLISP il faut que la fonction soit compilée. Il faudrait aussi que le compilateur vers la VM traite les récursions terminales, ce qui n'est pas si facile.

Aussi le plus sage est de faire une itération (avec la macro `loop`).

8.3.2 Résolution des adresses

La résolution des adresses consiste à mettre en correspondance les diverses entités du code qui représentent des adresses avec leur adresse en mémoire.

Cette mise en correspondance peut être statique ou dynamique : dans le premier cas les adresses sont substituées dans le code au moment du chargement, alors que dans le second cas, le code n'est pas modifié et une table de correspondance est créé au cours du chargement. Nous ne regarderons que la version statique qui est la plus réaliste et n'est guère plus difficile.

La résolution d'adresse va procéder avec l'aide de structures de données permettant d'associer des adresses aux variables et étiquettes rencontrées : on utilisera par exemple des tables de hachage (voir fonctions `make-hash-table` et `gethash`).

Etiquettes locales ou globales

Les étiquettes peuvent être locales à une étape de chargement ou globales. Il faut donc pouvoir les différencier syntaxiquement (en accord avec la génération de code) : on prendra par exemple des nombres pour les étiquettes locales et des symboles pour des étiquettes globales, qui sont en général les noms de fonctions.

On simplifiera tout en identifiant l'unité de chargement à la fonction.

Les structures d'indexation des étiquettes globales doivent être conservées d'un chargement à l'autre, au contraire des structures pour les étiquettes locales. Elles doivent donc faire partie de la structure de données VM.

Mais attention : la réinitialisation de la machine virtuelle doit aussi réinitialiser ces structures de données. De fait, les structures de données globales sont propres à chaque machine virtuelle : c'est la seule façon d'assurer la réentrance.

Variables statiques

Le traitement des variables statiques [1] se fait de la même manière que les étiquettes globales, à ceci près qu'il faut leur allouer de la mémoire : le mieux est d'allouer cette mémoire immédiatement après le code que l'on est en train de charger. Attention, cela peut poser un problème pour le point de lancement : peut être faut-il conserver 2 pointeurs, l'un sur le code, l'autre sur la dernière adresse occupée (code ou variable statique).

Résolution en une seule passe

La résolution en une seule passe va consister à charger le code tout en résolvant les adresses et en mémorisant les références en avant.

Pour cela, toute instruction est analysée au moment du chargement :

— si c'est une étiquette, on ne charge rien, mais on mémorise l'étiquette dans la structure d'association correspondante (locale ou globale) : si l'étiquette est déjà mémorisée, c'est une erreur à signaler ; s'il y a des références en avant, on les résout ;
— sinon, on charge l'instruction et on l'analyse pour voir si elle fait référence à une adresse qu'il faut résoudre : si c'est le cas, on regarde si cette adresse est déjà résolue, c'est-à-dire si on l'a dans les structures d'association :
  — si oui, on substitue l'adresse dans l'instruction ;
  — sinon, on mémorise l'instruction dans les références en avant associées à l'adresse considérée.

A la fin du chargement, certaines adresses n'ont pas été résolues :

— s'il s'agit de variables statiques, on leur alloue de la mémoire à la suite du code qui vient d'être chargé, lequel doit se terminer par un saut non conditionnel sans retour (`JMP`), un retour (`RTN`) ou un arrêt du processeur (`HALT`) ! et l'on résout ensuite les références en avant à ces variables ; cependant, il est préférable que le générateur de code ait identifié les variables statiques et les aient initialisées, par une séquence comme :

```
(LABEL <variable>)
(STORE (:REF <variable>) <valeur>)
```

  qui doit elle aussi être placée à la suite d'une unité de code se terminant par `JMP`, `RTN` ou `HALT`.
— s'il s'agit d'étiquettes globales, on attend leur chargement ultérieur : on peut néanmoins émettre un avertissement (`warn`) ;
— pour les étiquettes locales, il s'agit d'une erreur qu'il faut signaler, mais un avertissement suffit car il vaut mieux poursuivre le chargement pour détecter d'autres erreurs.

8.3.3 Chargement du chargeur

Bien entendu, dans une vraie machine le chargement d'un programme n'est pas effectué par une deuxième machine. A la fin, il faut donc compiler le chargeur et le charger (par le chargeur interprété)

1. Qu'on appelle souvent improprement "globales" : une variable statique n'existe qu'à un seul exemplaire tout au long de l'exécution d'un programme, alors qu'une variable globale est accessible depuis tous les points du programme.

dans chaque machine virtuelle dès sa création.

8.4 Exécution

8.4.1 Interprète naïf

Dans sa version la plus naïve, la fonction d'exécution est une boucle infinie d'exécution de l'instruction courante (à l'adresse du compteur ordinal). L'instruction est traitée par un `case` qui discrimine les différents types d'instructions, et appelle le traitement approprié.

Une instruction d'arrêt de la machine virtuelle est nécessaire : le contenu du registre `R0` (ou toute autre convention) est alors retourné par la fonction.

Remarque 8.2. L'exécution est une boucle infinie et dans le bon style LISP, il faudrait utiliser une fonction récursive. Cependant la durée de l'exécution peut être très longue et une fonction récursive non optimisée casserait la pile. Il faudrait donc impérativement que la fonction soit en récursion terminale, et pour que l'optimisation soit faite par CLISP il faut que la fonction soit compilée.

Aussi, comme pour le chargement, le plus sage est de faire une vraie boucle (avec la macro `loop`).

8.4.2 Machine virtuelle *threadée*

Une approche beaucoup plus efficace est appelée *threaded virtual machine*. Elle consiste à remplacer, dans le code de la machine virtuelle, le code opération par l'adresse du traitement correspondant. De plus, au lieu de se contenter d'incrémenter le compteur de programme (`PC`), le traitement de chaque opération saute, *in fine*, à l'adresse de traitement de l'instruction qui est dans le compteur de programme.

C'est cependant une opération de bas niveau qui peut se faire relativement facilement en C ou en assembleur, mais ne peut pas être réalisée en LISP, puisqu'il n'est pas réellement possible de considérer une adresse dans le code comme une donnée.

On peut néanmoins le simuler en remplaçant l'adresse par une fonction : on va donc remplacer tous les symboles qui désignent une opération, par exemple `op`, par un symbole désignant la fonction LISP `vm-op` qui implémente l'opération. L'exécution d'une instruction `instr` va alors consister à évaluer

```
(apply (car instr) (cdr instr))
```

L'exécution commence par exécuter l'instruction à l'adresse du compteur d'instruction, et chaque traitement d'opération se termine par la mise à jour du compteur d'instruction et l'évaluation de l'instruction correspondante.

Cela marche très bien à un (gros) détail près : l'exécution du programme consiste ainsi en un ensemble de fonctions récursives, mais ces fonctions sont récursives terminales. Il est donc indispensable d'exécuter ce code LISP en mode compilé pour qu'il puisse s'exécuter sur un programme consistant, CLISP n'optimisant pas les récursions terminales en mode interprété.

Par ailleurs, l'appel de la fonction de traitement se fait au travers d'`apply`, qui n'est pas d'une efficacité parfaite (mais qui fait heureusement un appel terminal sur la fonction qui lui est passée en paramètre).

Noter que cet argument des récursions terminales justifie aussi le fait qu'il ne soit pas possible de *threader* l'évaluateur du langage intermédiaire. La notion d'évaluateur *threadé* ne s'applique pas à un langage imbriqué, qui nécessite des récursions d'arbre qui sont par nature enveloppées. En d'autres termes, la notion associées au *thread* (un fil) repose sur une linéarisation de l'exécution qui ne peut se comprendre que sur un code déjà linéarisé.

8.5 Méthodologie de développement, outils de mise au point

8.5.1 Structure de données VM

Au total, il faut définir une structure de données qui comporte les différents éléments de la machine virtuelle : la mémoire, la pile, les régistres, ainsi que les informations liées au chargement (table des symboles, première adresse libre).

8.5.2 Interface fonctionnelle

Vu les variations possibles de l'implémentation au fil du temps, et pour des raisons de simple méthodologie de développement, il est indispensable d'encapsuler les accès aux différentes parties de la machine dans une interface fonctionnelle (fonctions ou macros) qui permettent de faire complètement abstraction de cette implémentation : faire donc des fonctions `get-register`, `push`, `pop`, etc. et les utiliser systématiquement.

8.5.3 Interface fonctionnelle pour l'utilisateur

L'interface fonctionnelle pour l'utilisateur — à ne pas confondre avec l'interface fonctionnelle d'implémentation — minimale sera la suivante :

— `(vm-make &key ...)` crée une machine virtuelle avec divers paramètres optionels (taille, nom, etc.) ;

— `(vm-load code &key vm)` charge une liste d'instructions `code` dans la machine `vm`, par défaut la machine virtuelle courante ; si le chargeur est déjà chargé, `vm-load` doit lancer la machine virtuelle sur le chargeur (par `vm-apply`) ;

— `(vm-run &key main vm)` lance la machine `vm` à l'adresse `main` (par défaut la dernière adresse chargée) : le contenu du registre `R0` (ou toute autre convention) est retourné par la fonction à l'arrêt de la machine ;

— `(vm-apply fn vm &rest args)` est un équivalent de `apply` qui applique la fonction `fn` préalablement chargée dans la machine `vm` aux arguments `args` (même sémantique que `apply`) : cette fonction fait appel aux deux précédentes.

8.5.4 Outils de mise au point

Différents outils sont aussi nécessaires pour :

— visualiser le contenu de la mémoire ou des registres ;

— tracer l'exécution ;

— réinitialiser la machine virtuelle.

Ces outils sont nécessaires pour donner une vision synthétique (ne pas imprimer les 100000 mots de la mémoire mais uniquement les 50 occupés) et lisible (imprimer le code ligne par ligne, éventuellement sur plusieurs colonnes).

9

Machine virtuelle à pile

Les processeurs physiques sont en général des machines à registres qui simulent une pile au moyen de registres dédiés (`SP` et `FP`), d'instructions dédiées (`push`, `pop`), et avec l'aide du système d'exploitation qui dédie une partie de la mémoire à la pile proprement dite.

La machine virtuelle à registres vue en cours (polycopié de Mathieu Lafourcade et Chapitres 7 et 8 du présent polycopié) simule directement un processeur physique, avec ses registres et instructions, dont certains dédiés à la pile, et sa mémoire, dont une partie est dédiée à la pile.

En réalité, la plupart des machines virtuelles effectivement utilisées actuellement — celles de JAVA par exemple — sont des machines à pile, dont le langage n'explicite pas les registres.

9.1 Principe d'une machine à pile

Le principe d'une machine à pile est le suivant :

— toute instruction de calcul (ou appel de fonction) trouve ses opérandes (ou arguments) au sommet de la pile : l'instruction les consomme (en les dépilant) et empile à la place le résultat ;
l'invariant principal est que le résultat est toujours au sommet de la pile.
— le stockage de résultats intermédiaires se fait dans la pile, soit en réservant un bloc de pile a priori, soit par empilement/dépilement explicite ;
— le contrôle s'effectue toujours au moyen d'une pile qui stocke les adresses de retour et les environnements, c'est-à-dire l'état de la pile avant les appels ; en pratique on peut utiliser une seule pile, qui mélange contrôle et données, mais on peut séparer les 2 pour rendre les deux rôles plus explicites ;
— la pile (ou chaque pile) dispose d'un pointeur de pile `SP`, et pour la pile de données, s'ajoute un pointeur de cadre (`FP`) : les deux sont des registres cachés, implémentés dans le processeur mais inutilisable directement par les programmes.

Les appels de fonctions se font comme suit :

— on commence par évaluer les arguments, ce qui empile leurs valeurs successives dans la pile de données ;
— on empile le nombre d'arguments (nécessaire pour savoir combien d'arguments il faut extraire de la pile pour appeler des fonctions LISP, ou pour calculer le nouveau `FP`) ;
— on fait un saut avec retour (`:CALL`) à l'adresse de la fonction, en gérant les pointeurs de pile (`SP` et `FP`) ;
— à l'entrée de la fonction on réserve un bloc de pile pour les variables locales.
— l'instruction de retour (`:RTN`) défait ce que (`:CALL`) a fait.

Si les deux piles sont séparées, elles vérifient les conditions suivantes :

— la pile de données ne contient que des données du calcul ou des nombres d'arguments ;
— la pile de contrôle ne contient que des adresses : valeurs des pointeurs de pile de données ou adresses de retour.

TABLE 9.1 – Langage de la machine virtuelle à pile

`(:LIT l)`	empile le littéral l
`(:VAR n)`	empile la valeur de la n-ième variable du bloc de pile courant
`(:SET-VAR n)`	affecte la valeur dépilée à la n-ième variable du bloc de pile courant
`(:STACK n)`	réserve un bloc de pile de taille n
`(:CALL f)`	appelle la fonction f, qui dépile ses paramètres et empile son résultat
`(:RTN)`	retourne d'un appel précédent
`(:SKIP n)`	saute n instructions
`(:SKIPNIL n)`	saute n instructions si la valeur dépilée est `NIL`
`(:SKIPTRUE n)`	saute n instructions si la valeur dépilée n'est pas `NIL`
`(:LOAD n)`	empile le contenu du mot mémoire d'adresse n
`(:STORE n)`	affecte au mot mémoire d'adresse n la valeur dépilée
`(:ADD)` `(:SUB)` `(:MUL)` `(:DIV)`	opérations arithmétiques les 2 opérandes sont dépilés le résultat est empilé

9.1.1 Exemple

L'exemple ci-dessous reprend la classique fonction `fibo` dans ce qui pourrait être le langage d'une machine à pile, dans une syntaxe très proche du langage intermédiaire (Chapitre 5).

```
(:LABEL FIBO) ;; déclaration de l'étiquette
(:STACK 0)    ;; pas de variables locales
(:LIT 1)      ;; on empile le litéral
(:VAR 1)      ;; on empile la valeur de la variable 1 (1ère après FP)
(:LIT 2)      ;; le nombre d'arguments
(:CALL <=)    ;; saut avec retour
(:SKIPNIL 2) ;; saut de 2 instructions en avant
(:VAR 1)      ;; on empile la valeur de retour
(:RTN)        ;; retour
(:VAR 1)
(:LIT 1)
(:LIT 2)      ;; le nombre d'arguments
(:CALL -)
(:LIT 1)      ;; le nombre d'arguments
(:CALL FIBO)
(:VAR 1)
(:LIT 2)
(:LIT 2)      ;; le nombre d'arguments
(:CALL -)
(:LIT 1)      ;; le nombre d'arguments
(:CALL FIBO)
(:LIT 2)      ;; le nombre d'arguments
(:CALL +)
(:RTN)        ;; retour (la valeur de retour est déjà sur le sommet de la pile)
```

Remarque 9.1. Le code ci-dessus a été écrit à la main. Il est probable qu'un générateur de code sans optimisation ne générera pas le premier `:RTN`, mais plutôt un `(:SKIP 14)`. Pour générer le `:RTN`, il faudrait savoir que l'expression `:IF` constitue le corps de la fonction (mais la génération ne serait plus indépendante du contexte), ou bien faire une deuxième passe pour optimiser, en remplaçant les `:SKIP` par des `:RTN` lorsqu'ils y sautent.

9.1.2 Le langage de la VM à pile

Les opérations élémentaires sont donc les suivantes :

— `(:STACK n)` effectue ce qui est nécessaire pour réserver un bloc de pile pour les *n* variables locales qui vont être utilisées dans le corps de la fonction (augmente `SP` de n) pour empêcher que les empilements suivants empiettent sur les variables locales.
— `(:LIT c)` empile sur la pile de données le littéral *c* passé en paramètre.
— `(:VAR n)` empile la *n*-ième variable/paramètre prise à partir du `FP`.
— `(:CALL f)` effectue l'appel de la fonction dont le nom *f* est passé en paramètre. Empile dans la pile de contrôle l'adresse de retour ainsi que la valeur courante de `FP`, et calcule la valeur du nouvel `FP` à partir du `SP` et du nombre d'arguments (sommet de pile immédiatement dépilé).
— `(:SKIPNIL n)` saute *n* instructions si NIL est en sommet de pile (NIL est dépilé).
— idem pour `(:SKIPTRUE n)` si une valeur différente de NIL est en sommet de pile (cette valeur est dépilée).
— et `(:SKIP n)` saute inconditionnellement *n* instructions.
— enfin, `(:RTN)` retourne de la fonction en défaisant ce que `:CALL` et `:STACK` ont fait, de telle sorte que la pile de données se retrouve dans l'état où elle était avant l'appel, avec les paramètres en moins et la valeur de retour en plus, et la pile d'appels retrouve son état précédent exact.
— on peut augmenter le langage avec des opérateurs arithmétiques (`:ADD`, `:SUB`, `:MUL`, `:Div`) : dans le code de `fibo`, on remplacera alors `((:LIT 2)(:CALL -))` par `((:SUB))`.

Dans la suite, on considère une version de la machine avec des opérations arithmétiques.

9.1.3 Exemple

Les différents états de la pile de données pour un appel de `(FIBO 10)` sont les suivants (la base de la pile est à gauche, le sommet de pile est à droite et l'instruction concernée est donnée à gauche) :

	10
`(:VAR 1)`	10 10
`(:LIT 1)`	10 10 1
`(:LIT 2)`	10 10 1 2
`(:CALL <=)`	10 NIL
`(:SKIPNIL 2)`	10
`(:VAR 1)`	10 10
`(:LIT 1)`	10 10 1
`(:SUB)`	10 9
`(:LIT 1)`	10 9 1
`(:CALL FIBO)`	10 34 (on ne détaille pas ici l'effet de l'appel récursif)
`(:VAR 1)`	10 34 10
`(:LIT 2)`	10 34 10 2
`(:SUB)`	10 34 8
`(:LIT 1)`	10 34 8 1
`(:CALL FIBO)`	10 34 21 (on ne détaille pas ici l'effet de l'appel récursif)
`(:ADD)`	10 55
`(:RTN)`	55

9.1.4 La pile de contrôle et la gestion de la pile de données

La pile de contrôle contient exclusivement les valeurs du `FP` et de l'adresse de retour, qui sont successivement empilées par `:CALL` et dépilées par `:RTN`.

Les actions associées sont les suivantes :

:call 1. empile l'adresse de retour (valeur du compteur ordinal + 1) puis `FP` dans la pile de contrôle ;
2. dépile le nombre d'arguments et calcule la nouvelle valeur du `FP` ;
3. saute à l'adresse de la fonction.

:stack réserve le bloc de pile pour les variables locales en incrémentant `SP` ; le bloc de pile d'une activation de fonction est donc la portion de pile de données comprise entre la valeur de `FP` à la sortie du `:CALL` et celle de `SP` à la sortie du `:STACK`.

:rtn 1. dépile la valeur de retour ;

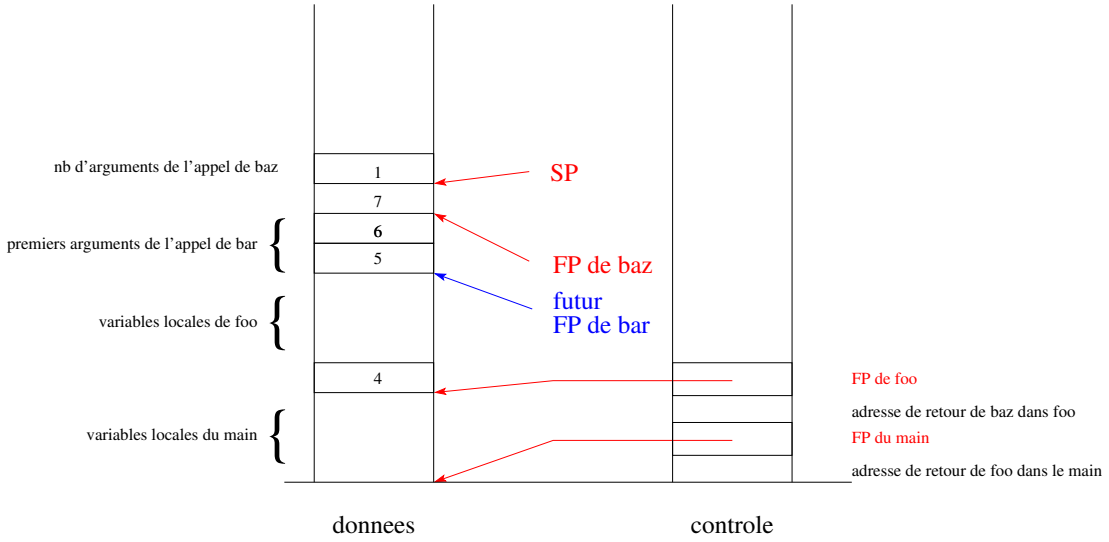


FIGURE 9.1 – Exemple d'état des piles

2. affecte à `SP` la valeur de `FP` ;
3. rempile la valeur de retour ;
4. dépile la pile de contrôle dans `FP` ;
5. dépile l'adresse de retour et y saute.

Si l'on fusionne les 2 piles, il faut bien veiller à faire les opérations dans le bon ordre.

9.1.5 Exemple

La figure 9.1 montre l'état de la pile, lorsqu'un programme principal appelle `(foo 4)` et que la fonction `foo` appelle `(bar 5 6 (baz 7))`. L'état de la pile est pris à la fin de l'exécution de l'opération `(:CALL baz)`. Le nombre d'arguments de l'appel de `baz` a été dépilé (mais la valeur est toujours physiquement dans la mémoire). S'il y a des variables locales, `SP` sera ensuite incrémenté par l'instruction `:STACK` qui commence le code de `baz`.

On a noté en bleu le futur `FP` de `bar`, où commencera le bloc de pile de `bar`, mais il ne sera réalisé que lors de l'appel de `(:CALL bar)` : ne pas oublier que l'imbrication `(bar 5 6 (baz 7))` se traduit par une inversion des exécutions effectives :

`(:LIT 5)(:LIT 6)(:LIT 7)(:LIT 1)(:CALL baz)(:LIT 3)(:CALL bar).`

Lors de l'appel effectif de `(:CALL bar)`, 7 aura été remplacé, dans la pile de données, par la valeur de `(baz 7)`.

9.2 Comparaisons

9.2.1 Machine à pile versus langage intermédiaire

La comparaison entre le langage intermédiaire du chapitre 5 et celui de la machine à pile est instructive.

La grosse différence est que le code de la machine à pile est un *code linéaire*, comme celui de la machine à registres et tout langage machine, alors que le code du langage intermédiaire est un *code imbriqué*, arborescent, comme celui de tout langage de haut niveau.

Les instructions sont cependant similaires : on retouve, inchangées, `:var` (ou `:set-var`) et `:lit`, pour accéder respectivement aux variables de l'environnement et aux littéraux. On retrouve `:call` pour les appels de fonction, mais sans les arguments puisque ceux-ci ont été préalablement désimbriqués. Bien sûr, les instructions `:mcall` et `:unknown` n'ont plus de raison d'être dans un code qui a été définitivement compilé, de même que `:progn` qui se dissout dans la linéarisation du code.

On pourra bien sûr exploiter la proximité des deux langages en adoptant exactement la même syntaxe, par exemple avec des paires pointées pour toutes les opérations.

9.2.2 Machine à pile versus machine à registres

L'intérêt de la machine à pile est que le code généré est beaucoup plus compact puisqu'il ne contient rien d'explicite pour la manipulation de la pile et des registres, la sauvegarde du contexte, etc.

Conditionnelles et sauts. Le traitement des conditionnelles représente une différence qui est relativement mineure. Par principe de la machine à pile, l'évaluation de la condition se termine en poussant la valeur de la condition (suivant les types manipulables par la machine ou la sémantique du langage : 0/1, booléens, ou `nil`/non-`nil`) sur la pile. Le langage fournit donc un ensemble d'opérations de sauts conditionnels suivant l'état du sommet de la pile. Dans un processeur à registres, une instruction (`:cmp` par exemple) positionne des flags d'après le contenu d'un registre ou la comparaison de 2 registres. C'est très similaire, et on peut même envisager d'ajouter une opération `:CMP` pour faire des comparaisons numériques entre les 2 sommets de pile, en lui associant des sauts dédiés aux conditions numériques (`:SKIPEQ`, `:SKIPGT`, `:SKIPGE`, etc.). Dans l'exemple de `fibo`, on remplacerait la séquence `(:LIT 2)(:CALL <=)(:SKIPNIL 2)` par `(:CMP)(:SKIPGT 2)`. On peut aussi compléter les sauts conditionnels booléens, par des sauts conditionnels arithmétiques où la condition est une comparaison à zéro implicite.

Nous avons ici illustré les sauts avec des instructions permettant de sauter une partie du code (`:skip`). Il s'agit donc de sauts relatifs. Un saut absolu à une étiquette ou adresse (`:jump`) serait aussi possible, comme dans la machine à registres (et vice-versa). Noter que les instructions de type `:skip` sont plus simples pour le générateur de code, puisqu'il n'a pas besoin de gérer des noms d'étiquettes locales (cf. Chapitre 8), mais sont moins simples à utiliser pour les boucles (`:skip` négatif), et compliqueraient fortement le travail d'un optimiseur qui supprimerait les instructions inutiles. Un tel optimiseur a cependant beaucoup moins de justifications qu'avec une machine à registres.

Accès mémoire. La machine à pile diffère de la machine à registres par un autre point : nous n'avons pas spécifié d'opérations pour les accès mémoire. Ces accès mémoire sont en pratique inutiles dans notre cadre, car nos machines virtuelles ne font pas d'allocations dans le tas : elles se reposent donc sur LISP pour les accès mémoire à partir des structures LISP. Si l'on veut implémenter des variables statiques, il sera néanmoins nécessaire de rajouter les équivalents de `STORE` et `LOAD`.

Encore une fois, ces machines virtuelles sont très arbitraires, puisqu'il n'y a pas de contrainte physique à respecter : vous avons donc proposé deux styles assez différents pour les deux machines virtuelles, et vous pouvez donc choisir.

9.3 Outillage de la machine à pile

9.3.1 Transformation du langage intermédiaire en langage de la machine à pile

La transformation du langage intermédiaire vers la machine à pile (par une fonction `li2svm`) est relativement simple, en tout cas beaucoup plus simple que la génération de code vers la machine à registres (`li2vm`) : l'essentiel est en effet de linéariser le code en le désimbriquant, et la mécanique des piles reste ici cachée.

9.3.2 Chargement de code dans la machine à pile

En théorie, le fait que la machine soit à pile ne change rien. Ce qui change, en revanche, dans la présentation qu'on en a faite, c'est le remplacement des opérations de saut absolu (`jump`) par des sauts relatifs (`skip`). Les étiquettes locales deviennent inutiles, et la résolution des symboles en est simplifiée d'autant : il n'y a plus que des étiquettes globales.

9.3.3 Exécution de la machine à pile

En compensation, la simulation de la machine à pile est plus compliquée, puisque toute la mécanique des piles doit être implémentée derrière les instructions d'apparence anodine comme `:call`, `:rtn`, `:stack`, ... Cependant, la réalisation de cette mécanique, qui s'apparente à de l'interprétation, n'est pas plus compliquée que sa génération dans une machine à registres, qui relève de la compilation !

De façon générale, la machine à pile s'implémente de la même manière que la machine à registres. Si l'on sépare pile de contrôle et pile de données, il faut cependant allouer un espace spécifique pour l'une des deux, soit dans la mémoire, soit dans une autre mémoire dédiée. Il suffit bien sûr de donner des valeurs appropriées aux 2 pointeurs de pile (`SP`) lors de la création de la machine : mais gare aux débordements ! Pour éviter les erreurs irréparables, il est prudent de tester les débordements de pile à l'exécution.

10

Projet de Compilation-Interprétation

L'objectif du projet est de réaliser, en LISP (ou en SCHEME), 2 machines d'exécution de programmes LISP ou SCHEME :

— en version interprétée, avec un méta-évaluateur et différentes variantes ;

— en version compilée, avec un générateur de code dans une machine virtuelle qui sera interprétée.

Au total, donc, 3 modules à développer : génération de code, interprète de machine virtuelle et méta-évaluateur.

Deux grandes variantes sont possibles.

— la transformation du code LISP dans le langage intermédiaire (Chapitre 5) représente une étape de plus, mais c'est un investissement car les étapes ultérieures sont plus simples ;

— on peut considérer une machine virtuelle à registres ou à piles (Chapitre 9) : le langage de la seconde cache plus de choses mais il est plus simple à réaliser. Il est aussi possible de traiter les 2 machines virtuelles, par exemple en trabnsformant le code de la machine à pile dans celui de la machine à régistres (l'inverse peut poser de gros problèmes).

Les Figures 10.1, 10.2 et 10.3 montrent trois projets possibles.

ATTENTION ! En 2012-13, le projet doit impérativement inclure la séquence sur fond rouge (gris dans la version papier) de la Figure 10.3.

10.1 Couverture du langage LISP

La couverture du langage doit être la plus grande possible. Dans tous les cas, pour que la réflexivité soit la plus grande possible, il est conseillé d'utiliser un sous-ensemble restreint du langage dans le développement des 3 modules à réaliser.

Méta-évaluateur de LISP Quasiment tout COMMON LISP peut être traité, mais on peut se restreindre à l'union :

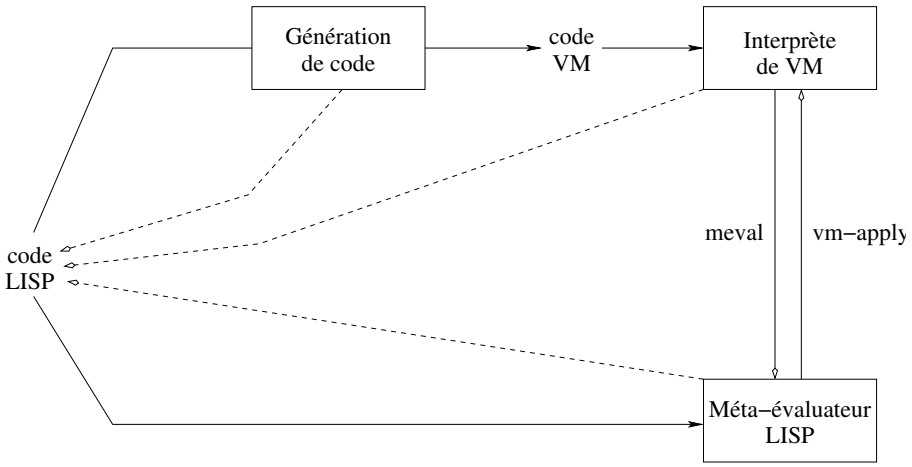
— de ce qui est utilisé dans le méta-évaluateur lui-même, qui **doit** se méta-évaluer lui-même (Section 4.4) ;

— des traits du langage vus en cours comme les fermetures, continuations et `labels` (Section 4.5).

Une fois cela acquis, la méta-évaluation des 2 autres modules ne doit pas poser de grands problèmes.

Faire attention néanmoins à certaines macros comme `loop` qui s'expansent en utilisant des traits mineurs du langage que l'on n'aura pas eu le temps d'aborder.

Transformation vers le langage intermédiaire Pour le langage intermédiaire et les VM, de nombreux traits de LISP sont plus difficiles à traiter (par exemple, les mots-clés `&rest` et `&optional`). De façon générale, la transformation dans le langage intermédiaire du code de la transformation est plus difficile à réaliser et n'est pas exigée. C'est néanmoins possible.



Les boites désignent des programmes. Les flèches pleines à tête noire désignent le flot des données à travers ces programmes. Les flèches en tireté montrent la boucle de réflexion qui permet de prendre ces programmes comme donnée du projet. Enfin, les flèches noires à tête blanche montrent des exemples d'interopérabilité entre les deux interprètes, de VM et de LISP.

FIGURE 10.1 – Schéma du projet LEC, version originale.

Génération de code vers les VM Le code usuel doit être traité, en particulier celui du chargeur. La compilation du méta-évaluateur, du générateur de code n'est pas exigée mais elle peut être faite sans grande difficulté.

Quant aux fermetures, c'est-à-dire `function`, c'est un problème difficile. Il est en revanche possible de compiler `labels` en toute généralité (sauf l'usage de `function`), mais cela reste assez délicat.

Machine virtuelle Bien entendu, l'interprète de la machine virtuelle doit accepter n'importe quel code syntaxiquement correct, en particulier le code généré par la compilation de n'importe quel code LISP couvert par la génération de code.

Traitement des fichiers On programme usuellement en stockant les programmes dans des fichiers.

En LISP, ces fichiers peuvent être chargés (par la fonction `load`). Il faut donc prévoir des équivalents pour charger les fichiers pour le méta-évaluateur (en définissant la fonction `mload`), mais aussi pour transformer un fichier d'un langage dans un autre (LISP vers LI ou VM), ou pour charger toutes les définitions d'un fichier dans une VM.

10.2 Le méta-évaluateur et ses variantes

Sur la base du méta-évaluateur qui se méta-évalue lui-même (Section 4.4) le méta-évaluateur réalisé devra implémenter au moins une des variantes ou extensions suivantes :

- les fermetures et fonctions locales (Section 4.5), avec application à la terminalisation des fonctions récursives enveloppées (cf. polycopié de LISP) ;
- un méta-évaluateur de ou en SCHEME (Section 4.8) ; le méta-évaluateur de SCHEME en SCHEME doit se méta-évaluer lui-même ; un méta-évaluateur de COMMON LISP en SCHEME devrait pouvoir méta-évaluer un méta-évaluateur de COMMON LISP ;
- le filtrage avec son application aux macros, à la SCHEME (Section 4.7.1) ;
- le méta-évaluateur moins naïf du Chapitre 5.

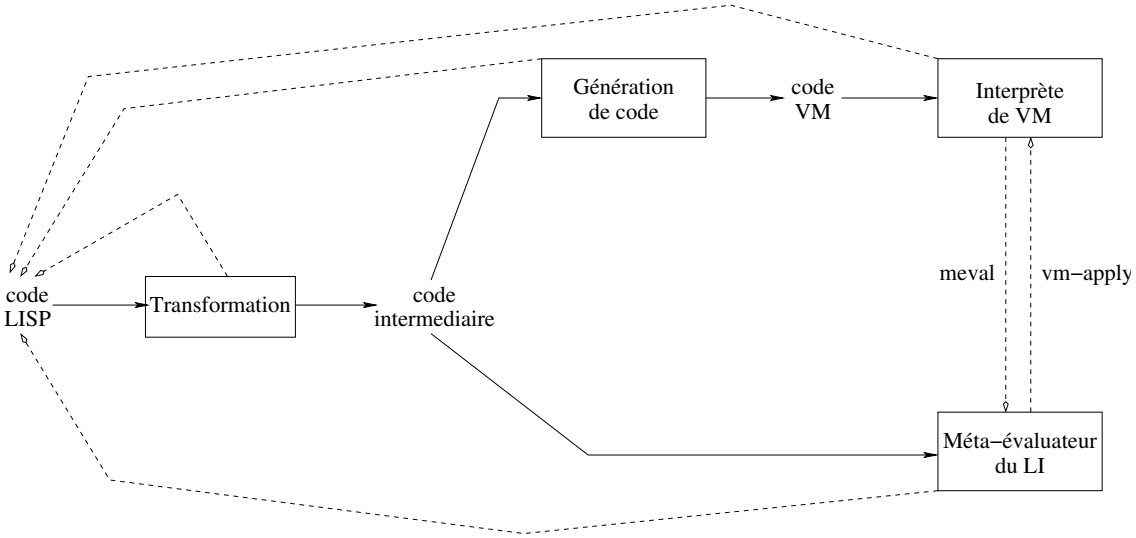


FIGURE 10.2 – Schéma du projet LEC, version 2

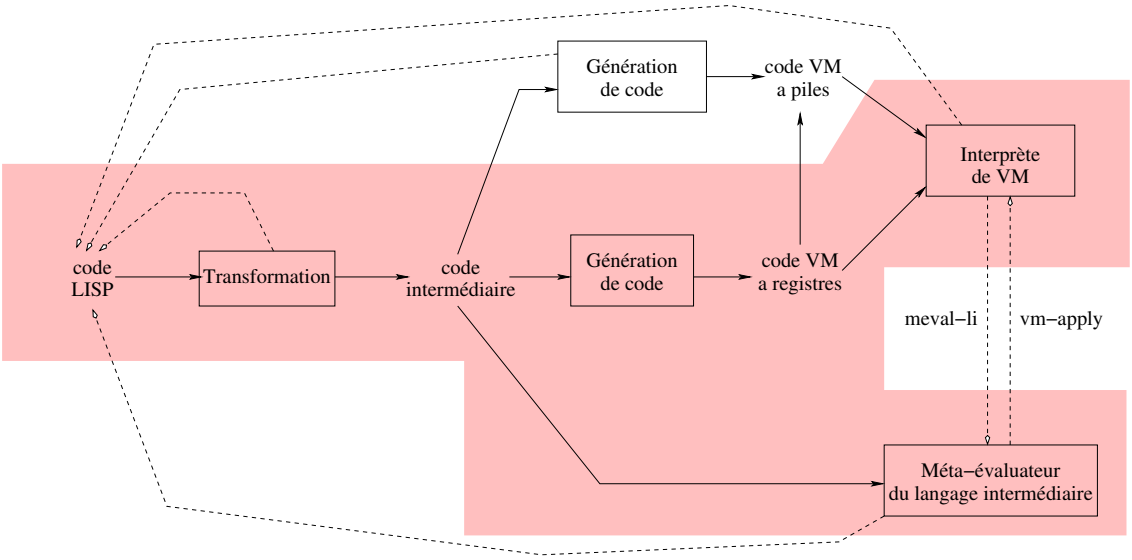


FIGURE 10.3 – Schéma du projet LEC, version 3. En 2015-16, la partie grisée est exigée, avec la machine à registres. L'année précédente, c'était la machine à pile.

10.3 Alternative : génération de code à partir du langage intermédiaire

Une alternative plus efficace consiste à passer par le méta-évaluateur moins naïf du langage intermédiaire, pour la génération de code aussi bien que pour l'évaluation. Toute l'analyse statique réalisée pour la transformation n'est plus à faire pour la génération de code.

10.4 Alternative : machine virtuelle « *threadée* »

Pour l'exécution de la machine virtuelle, une alternative plus efficace consiste à faire une machine virtuelle « *threadée* », telle qu'elle est succinctement développée en Section 8.4.2. L'impact principal porte sur la fonction d'exécution, qui doit être récursive et compilée par CLISP : il sera donc hors de question de la compiler et de l'exécuter dans la VM, sauf à optimiser les récursions terminales... Un impact secondaire mineur porte sur la fonction de chargement, puisqu'il faut remplacer le code opération par la fonction qui exécute l'opération.

Cette variation s'applique aussi aux machines virtuelles à registres ou à pile.

10.5 Interopérabilité évaluateur / machine virtuelle

Ces 2 machines d'exécution doivent être *interopérables* :

— la machine virtuelle doit pouvoir appeler le méta-évaluateur pour les fonctions qui ne seraient pas compilées,

— le méta-évaluateur doit pouvoir lancer la machine virtuelle pour l'évaluation des fonctions qui y seraient compilées et chargées.

Exemple de test d'interopérabilité : les 2 fonctions suivantes, mutuellement réflexives sont respectivement compilées et chargées dans la machine virtuelle ou interprétée par le méta-évaluateur.

```
(defun fibo1 (n)                    (defun fibo2 (n)
   (if (<= n 1)                        (if (<= n 1)
       1                                   1
     (+ (fibo2 (- n 1))                  (+ (fibo1 (- n 1))
        (fibo2 (- n 2)))))                  (fibo1 (- n 2)))))
```

On peut compliquer, avec 3 fonctions dont chacune appelle les deux autres et dont une est interprétée et les deux autres compilées et exécutées dans 2 machines différentes, respectivement à registres et à pile.

10.6 Réflexivité

> Car, pour forger, il faut un marteau, et pour avoir un marteau, il faut le fabriquer.
> Ce pourquoi on a besoin d'un autre marteau et d'autres outils,
> et pour les posséder, il faut encore d'autres instruments, et ainsi infiniment.
>
> Spinoza, *De la réforme de l'entendement*, 1661.

Enfin, ces 2 machines d'exécution sont *réflexives* : le code de chaque module peut être méta-évalué ou compilé puis vm-évalué. En particulier, pour simuler une machine véritable, *il faut que le chargeur de la machine virtuelle soit lui-même compilé et chargé* : c'est le *bootstrap* (ou *amorçage*) des systèmes d'exploitation.

La réflexivité impose que le code soit *réentrant*. En pratique, pour la machine virtuelle, il faut pouvoir en faire coexister plusieurs exemplaires dans la même session CLISP, que l'on active alternativement. L'utilisation de variables statiques est donc à exclure.

On peut ensuite envisager de charger le code compilé de l'interprète de la VM dans une VM, de telle sorte que cette VM assure l'exécution d'une autre VM.

La figure 10.1 synthétise l'ensemble. Les flèches en tireté décrivent la réflexivité : chaque boîte consiste en du code LISP qui peut suivre le même traitement. La figure 10.2 décrit le même projet dans lequel le

bénéfice de la transformation dans le langage intermédiaire est partagée par la génération de code et le méta-évaluateur.

Sur tout ce qui concerne la réflexivité et le « méta », voir le module GMIN̈310 *Méta-Programmation et Réflexivité* du S3, ainsi que le polycopié associé [Duc13b].

10.7 Exemple d'application en vraie grandeur

Un groupe d'étudiants de l'année 2000-01 a réalisé, en TER, une implémentation de LISP en JAVA, sur la base des différentes briques développées dans le cadre de ce projet :

— transformation du générateur de code pour lui faire produire le code de la JVM [MD97] ;
— compilation du méta-évaluateur, avec le générateur précédent, pour lui faire produire un évaluateur LISP en JVM ;
— développement de la fonction READ avec le générateur d'analyseur syntaxique JAVACC ;
— implémentation en JAVA des types de bases (essentiellement `cons`).

L'ensemble est disponible sur le site ? ? ?.

EXERCICE 10.1. Formaliser cette application à JAVA en utilisant les notations $Int_I(J)$ et $Comp_I(J, K)$ du Chapitre 2, avec I, J et K pris dans les langages LISP, JAVA et JVM. □

Index

Bibliographie

[ASS89] H. Abelson, G.J. Sussman, and J. Sussman. *Structure et interprétation des programmes informatiques*. InterÉditions, Paris, 1989.

[ASU89] A. Aho, R. Sethi, and J. Ullman. *Compilateurs : principes, techniques et outils*. InterEditions, Paris, 1989.

[DEMN98] R. Ducournau, J. Euzenat, G. Masini, and A. Napoli, editors. *Langages et modèles à objets : État des recherches et perspectives*. Collection Didactique. INRIA, 1998.

[Duc13a] R. Ducournau. Petit Imprécis de LISP. Université Montpellier 2, polycopié de Master Informatique, 85 pages, 2013.

[Duc13b] R. Ducournau. Programmation par Objets : les concepts fondamentaux. Université Montpellier 2, polycopié de Master Informatique, 215 pages, 2013.

[Kar97] J. Karczmarczuk. Implantation des langages de programmation — compilateurs et interprètes, 1997. Polycopié Licence d'Informatique, Caen.

[Kle71] S.C. Kleene. *Logique mathématique*. Collection U. Armand Colin, Paris, 1971.

[LMB94] J. R. Levine, T. Mason, and D. Brown. *LEX & YACC*. O'Reilly, 1994.

[MD97] J. Meyer and T. Downing. *JAVA Virtual Machine*. O'Reilly, 1997.

[Que90] Ch. Queinnec. *Le filtrage : une application de (et pour) Lisp*. InterÉditions, Paris, 1990.

[Que94] Ch. Queinnec. *Les langages Lisp*. InterÉditions, Paris, 1994.

[SJ93] E. Saint-James. *La programmation applicative : de* LISP *à la machine en passant par le lambda-calcul*. Hermès, 1993.

[Ste90] G. L. Steele. *Common Lisp, the Language*. Digital Press, second edition, 1990.

[Ter96] P.D. Terry. *Compilers and Compiler Generators an introduction with C++*. Pearson, 1996.

Table des matières