Une courte introduction à Scheme

Notes de cours

Jean-Jacques Girardot
girardot@emse.fr
Février 2008

Ecole Nationale Supérieure des Mines de Saint-Etienne
158 Cours Fauriel
42023 Saint-Etienne Cédex

Document de travail. Release 0.32

Date d'impression 2 avril 2012

Chapitre 1

Scheme, un langage fonctionnel

1.1 Introduction

Nous allons présenter dans ce chapitre le langage Scheme, qui est un langage fonctionnel dérivé de Lisp, l'un des premiers langages pour l'intelligence artificielle. Scheme est défini formellement dans un rapport [25], et de nombreuses introductions à Scheme sont disponibles, tant sous forme d'ouvrages [28, 17, 27, 1, 3, 2, 8, 9] que sur l'Internet.

Le langage Scheme est le résultat d'une travail de formalisation et de simplification des diverses versions de Lisp[10], y compris l'une des normes industrielles, Common Lisp, mais également d'Algol 60 [16].

Lisp est un langage originellement dû à John McCarthy [23], pour lequel de nombreuses implémentations ont été réalisées ([10, 30, 11, 21, 29]).

Il existe nombre d'ouvrages de référence du langage Scheme, à commencer par la norme du langage ([24, 25]), des manuels de référence (par exemple [13, 14, 15, 19, 18, 11]), ainsi que des supports de cours divers utilisant le langage ([9, 28, 17, 27, 8]), le plus connu étant probablement « Structure and Interpretation of Computer Programs », [1, 3, 2].

Nous utiliserons ici une mini version du langage Scheme, SCH, écrit en C portable.

Le système SCH a été conçu afin d'être enchassable, c'est-à-dire de pouvoir servir de langage auxiliaire dans une application nécessitant un langage interne pour la définition de certaines fonctions, un langage de définition de macro commandes, ou un langage externe pour les interactions.

SCH a été utilisé pour l'écriture d'un processeur d'analyse de textes, pour la construction d'un analyseur XML. Il est aussi utilisé à l'occasion comme interprète *Scheme* en *stand-alone*.

1.2 Le langage SCHEME

Le langage *Scheme* est un dérivé des langages Lisp, qui va dans le sens d'une formalisation et d'une simplification des concepts. Scheme est un petit dialecte de *Lisp*, très propre, et, qui plus est, particulièrement agréable à utiliser. Le langage a été conçu pour proposer un petit nombre de constructions régulières, orthogonales [1], qui favorisent une grande variété de styles de programmation, en particulier les programmations *fonctionnelle*, *impérative* et *à objets*.

Scheme fut conçu en 1975, par Gerald J. Sussman et Guy L. Steele Jr., au MIT (*Massachussets Institute of Technology*), comme un prototype pour la validation de leur recherche autour des concepts d'*acteurs* (Cf. par exemple les travaux de Gull Agha, Carl Hewitt et de Henry Lieberman [4], [5] ou [22]) et de *fonctions*. Le langage, initialement outil de recherche et d'enseignement, a évolué au fil des ans, intégrant avec parcimonie certains des concepts les plus évolués de l'informatique, tout en restant un outil simple à maîtriser et à manipuler. Citons à ce sujet cet extrait de l'introduction à la norme du langage [24] :

> Les langages de programmation devraient être conçus, non en empilant extension sur extension, mais en supprimant les faiblesses et les restrictions qui semblent rendre nécessaire l'addition de ces nouvelles extensions. Scheme démontre qu'un petit nombre de règles de constitution d'expressions, sans restrictions sur la manière dont ces expressions sont composées, suffisent à définir un langage de programmation pratique et efficace, qui est suffisamment flexible pour supporter la majorité des paradigmes de programmation en usage aujourd'hui.

1. Ce terme indique ici que ces constructions peuvent être combinées entre elles (pratiquement) sans aucune restrictions.

Scheme fut l'un des premiers langages à considérer les procédures[2] comme des *citoyens de première classe*[3]. Scheme permet également de disposer de la plupart des concepts des langages fonctionnels, qui sont soit immédiatement disponibles dans le langage, soit simples à implanter ; accès à l'environnement, évaluation paresseuse, continuation, filtrage[4]...

La description complète du langage tient en une cinquantaine de pages (le document[5] *Revised*[5] *Report on the Algorithmic Language Scheme* [25] tient lieu de norme, et comporte une description formelle de la syntaxe et de la sémantique du langage on trouvera également dans [26] une sémantique opérationnelle de Scheme).

Scheme est fondé sur un modèle formel (le *lambda calcul* – cf. [7], ou encore [6], [20], etc.), si bien que les programmes Scheme sont pourvus de plein de propriétés sympathiques, et que l'on peut construire des outils fiables d'analyse et de transformation de code.

Le langage peut être qualifié de *fonctionnel* en ce sens qu'il offre une complète liberté dans la manipulation des fonctions, bien qu'il ne présente pas toutes les caractéristiques des « véritables » langages fonctionnels. Scheme présente aussi de nombreux traits empruntés aux langages impératifs (affectation, effets de bord), tout comme il offre des constructions permettant de mettre en place des outils de programmation par objets.

1.3 Syntaxe et Sémantique

Comme tout dialecte de Lisp, Scheme repose sur des règles syntaxiques fort simples : notation *préfixée* et entièrement *parenthésée* pour les programmes et les données.[6] La *fonction* précède donc ses arguments, et le tout est encadré par un couple de parenthèses[7]. Voici quelques exemples de cette syntaxe :

```
(+ 2 3)
(* 5 8)
```

Les blancs servent à séparer les éléments des listes. La seconde expression calcule le produit de cinq nombres représentés chacun par un seul chiffre décimal. La troisième calcule le produit du nombre `23` par le nombre `578`. Enfin, la dernière expression se réduit à un seul nombre. Sa valeur est `17`. Les résultats des expressions précédentes sont `5`, `1680` et `13294` respectivement. On notera donc que les fonctions (au moins certaines d'entre elles), admettent un nombre variable d'arguments.

La plupart des systèmes Scheme (et Lisp) opèrent en mode *interactif*, c'est à dire que la machine attend une commande de l'utilisateur, l'exécute immédiatement, imprime le résultat de cette exécution, puis se remet en attente. Les commandes attendues par Scheme sont tout simplement des expressions à calculer.

1.3.1 Expressions Symboliques

Tout programme Scheme est représenté par des expressions symboliques. Une expression symbolique est une structure arborescente de taille arbitraire, composée d'atomes.

1.3.1.1 Les atomes

Les atomes sont les données élémentaires du langage. Les atomes se décomposent typiquement en un petit nombre de catégories :

- les *symboles*, ou *identificateurs* ; les règles précises de formation des symboles diffèrent d'un système à un autre. De manière informelle, un symbole est une suite de caractères ne représentant ni un nombre, ni une chaîne de caractères. Hormis une dizaine de caractères utilisés comme ponctuations, la plupart des caractères du code *ASCII* sont admis à l'intérieur d'un identificateur. Voici quelques identificateurs :

  ```
  truc  +  symbol?  2@3?24  let*  set-car!  élément
  ```

- les *booléens*, représentent les valeurs logiques *vrai* et *faux*. Ils sont dénotés par les noms `#t` et `#f`.

2. On utilisera ici indifféremment les termes *fonction* et *procédure*, tous les sous-programmes rendant un résultat.

3. *First class citizen in the text*, ce qui veut dire que ces procédures peuvent être fournies comme paramètres à des fonctions, rendues comme résultat par une fonction, et conservées comme élément d'une structure.

4. Des exemples de ces concepts seront proposés dans la suite de ce cours.

5. Le chiffre « 5 » dans *Revised*[5] *Report...* ne fait pas référence à une note de bas de page, mais est une abréviation de *Revised revised revised revised revised Report...*, indiquant que ce fameux document en est maintenant à sa sixième mouture !

6. Cette notation est dite « polonaise préfixée de Cambridge ». La *notation polonaise*, due au logicien Polonais Jan Lukasiewicz, consiste à écrire une expression sous forme linéaire, en faisant suivre les opérandes par la fonction. Ainsi, $3x+5$ s'écrit $3x\times 5+$. Cette notation est utilisée dans le langage Forth. Sa variante préfixée consiste à placer la fonction en premier, ce qui donnerait ici : $+\times 3x5$. Enfin, la notation dite de Cambridge ajoute des parenthèses pour englober chaque sous-expression, ce qui permet aux fonctions d'accepter un nombre variable de paramètres. L'écriture devient alors : $(+(\times 3x)5)$.

7. On parle de *liste* pour désigner une telle suite d'éléments placés entre parenthèses.

```
#t  #f
```

- les *nombres* ; un nombre est une suite de chiffres, débutant éventuellement par un signe + ou un signe −, pouvant comporter un point décimal ou un exposant.

```
2 -45 12.34e-5
```

- les *caractères* ; un caractère s'écrit sous la forme d'un dièse, #, suivi d'un *backslash*, \, suivi du caractère ou du nom du caractère. Voici quelques caractères :

```
#\a  #\+  #\6
```

- les *chaînes de caractères* ; une chaîne de caractères est une suite arbitraire de caractères placée entre doubles quotes **"**. Une double quote doit être précédée du caractère d'échappement \. On en déduit qu'un \ doit aussi être précédé d'un autre \ pour être digne de figurer dans une chaîne.

```
"Salut, Monde"
```

1.3.1.2 Les expressions symboliques

Une expression symbolique est soit un atome, soit une suite d'expressions symboliques (séparées par des blancs), placée entre parenthèses. On parle alors de *liste*. Quelques exemples :

```
(a b c)
(2 3 (quote 4) (+ 5 6))
```

1.3.1.3 Les ponctuations

Voici une liste non exhaustive des caractères spéciaux du langage :

- L'espace : il sert de séparateur. Une suite quelconque d'espaces est équivalente à un seul. Le caractère de fin de ligne, le caractère de tabulation et la plupart des caractères dits "de contrôle" sont assimilés à des blancs.
- Le point **.** utilisé seul, le point sert de séparateur dans une *paire pointée* (nous y reviendrons ultérieurement). Le point sert également à séparer la partie entière de la partie décimale d'un nombre. Il peut également être utilisé (il n'a alors pas de signification particulière) à l'intérieur d'un identificateur.
- Le point-virgule **;** il introduit un commentaire. Le restant de la ligne est ignoré, et le tout équivaut à un blanc.
- La double quote **"** elle introduit une chaîne de caractères.
- Les parenthèses **( )** servent à délimiter listes et paires pointées.
- Le dièse **#** est un *macro-caractère*. Il précède certaines constructions particulières du langage ; par exemple, **#x2fc** est une notation *hexadécimale* du nombre entier 764 (c.f. [24], § 6.5.4).
- L'apostrophe **'**, ou *quote*, introduit des données symboliques.
- La *contre apostrophe*, **`** ou *back-quote*, joue un rôle voisin de celui de la quote.
- Les séquences **,** et **,@** sont utilisées en conjonction avec la back-quote.
- Citons enfin les accolades **{ }** et les crochets **[ ]**, qui ne sont pas utilisés dans le langage, mais qui sont « réservés » pour de futures extensions.

1.3.2 Programmes et évaluation

Tout programme Scheme est représenté par une expression symbolique. Les règles d'évaluation d'une expression symbolique sont les suivantes :

- Les constantes (nombres, caractères, booléens, ch aines de caractères, etc.) représentent leur propre valeur. Ainsi :

```
3
;=  3
```

  est un programme Scheme, dont l'effet est de calculer la valeur 3. Cette réponse est ici affichée, précédée des caractères « **;=** ».
- Les atomes représentent les variables Scheme. Ces variables peuvent être prédéfinies, comme +, −, ⋆, sin, cos, remainder, etc., ou définies par l'utilisateur. L'évaluation d'un atome consiste à le remplacer par la valeur qui lui est associée dans le contexte courant. L'évaluation d'un atome auquel aucune valeur n'est associée dans le contexte courant provoque une erreur :

```
argent
undefined variable : argent
```

  La plupart des valeurs associées aux variables prédéfinies sont des valeurs fonctionnelles :

```
cons
;=  #<SUBR :cons :2>
```

- Une liste représente une application fonctionnelle, ou une forme spéciale.
  - ▷ Une *application fonctionnelle* consiste en *l'application* d'une fonction sur ses opérandes. Les éléments de la liste (constantes, atomes ou autres listes) sont d'abord évalués, dans un ordre non précisé.
    Le premier élément de la liste, *qui doit être une valeur fonctionnelle*, est alors appliqué aux autres éléments. Le résultat de cette application devient le résultat de l'expression symbolique.
  - ▷ Une *forme spéciale* est un cas très particulier d'expression Scheme. Les formes spéciales ne sont pas des applications fonctionnelles. Le premier élément d'une telle forme n'est pas une désignation de fonction, mais un *mot clef*. Les autres éléments d'une forme spéciale ne sont pas nécessairement évalués : l'utilisation qui en est faite dépend de la sémantique associée à la forme spéciale.
    Alors que le mécanisme d'exécution d'une application fonctionnelle est toujours le même, celui d'une forme spéciale est ainsi propre à chaque forme. Heureusement, il n'existe qu'un tout petit nombre de formes spéciales (c.f. page 7), qui, pour la plupart, correspondent aux *instructions de contrôle* des langages impératifs.

1.4 Présentation générale

1.4.1 Mise en oeuvre

Le système se lance par la commande **`sch`**. Il s'exécute en mode interactif, et affiche les résultats pour chaque expression introduite. Le système s'interrompt sur une fin de flot d'entrée, ou par exécution de la commande **`exit`** :

```
[kiwi]$ sch
SCH V0.17 Start
(+ 2 3)
;=  5
(define (fun x)
    (+ x 3))
;=  fun
(fun 5)
;=  8
(exit)
SCH End
[kiwi]$
```

1.4.2 La partie standard

En général, Sch est assez conforme au langage Scheme tel que décrit dans [25]. Il utilise la syntaxe parenthésée de Scheme, et en fournit la plupart des types de données. Les utilisateurs potentiels sont conviés à consulter le rapport [25]. Ceux qui recherchent un bon cours d'informatique abordant Scheme peuvent consulter [1] ou sa traduction française [2].

Dans la mesure où Sch est destiné à opérer en intégration avec une application écrite en C, des types de données de base sont compatibles avec celles du langage C.

1.4.3 Syntaxe

Le langage SCH utilise un sous-ensemble de la notation traditionnelle de Scheme. Les éléments du langage sont :

identificateurs une suite quelconque de caractères, ne comportant pas les séparateurs du langage.

chaîne une suite de caractères placés entre doubles apostrophes. À l'intérieur d'une chaîne, une double apostrophe peut être représentée par la séquence `\"` et le caractère back-slash par la séquence `\\`.

nombre un nombre s'exprime sous sa forme décimale, une suite de chiffres appartenant à l'intervalle `[0-9]`, éventuellement précédée du caractère -. Un nombre entier peut aussi s'exprimer sous forme binaire, octale ou hexadécimale. Il est alors précédé de la suite `#b`, `#o` ou `#x` selon le cas, suivie d'une suite de caractères appartenant à l'intervalle `[01]`, `[0-7]`, ou `[0-9a-zA-Z]` selon les cas.

Un nombre peut également se représenter par la séquence `#\` suivie d'un caractère ; la valeur est celle du code ASCII du caractère.

booléen un booléen représente l'une des valeurs **vrai** ou **faux**, qui se notent respectivement `#t` et `#f`. Lorsqu'une valeur booléenne est attendue par une opération du langage, n'importe quelle valeur du langage différente de **faux**, `#f`, est considérée comme vraie.

séparateurs ces caractères spécifiques servent à délimiter les constructions du langage. Les séparateurs sont :

- le blanc et les caractères assimilés (tabulation, fin de ligne), qui permettent la séparation des éléments lexicaux du langage ;
- les parenthèses, qui permettent la création de listes, qui elles-mêmes représentent des données ou des expressions du langage ;
- la quote, qui introduit une constante ;
- la double quote, qui est le caractère utilisé pour délimiter les constantes de type chaîne ;
- le point, utilisé dans la notation des paires pointées ;
- le point-virgule, qui introduit un commentaire se terminant à la fin de la ligne ;
- le dièse, qui introduit diverses notations (nombres).

1.5 Opérations

1.5.1 Description des opérations

1.5.1.1 Formes spéciales

Une forme spéciale est une liste dont le premier élément est un identificateur qui, dans ce contexte, représente un mot-clef du langage. La table suivante décrit, parmi les formes spéciales de Scheme, les mots-clefs du langage SCH.

Nom	Description	Voir...
=>	utilisé avec *cond*	§ 1.5.2, p. 9
and	expression conditionnelle	§ 1.5.2, p. 8
begin	séquencement	§ 1.5.2, p. 8
case	expression conditionnelle	§ 1.5.2, p. 8
cond	expression conditionnelle	§ 1.5.2, p. 9
define	déclaration de variable	§ 1.5.2, p. 9
delay	expression retardée	§ 1.5.2, p. 9
do	itération	*non implantée*
else	utilisé avec *cond* et *case*	§ 1.5.2, p. 9 et § 1.5.2, p. 8
if	expression conditionnelle	§ 1.5.2, p. 10
lambda	création de fonction	§ 1.5.2, p. 10
let	création d'environnement local	§ 1.5.2, p. 10
let*	création d'environnement local	*non implantée*
letrec	création d'environnement local	§ 1.5.2, p. 10
or	expression conditionnelle	§ 1.5.2, p. 10
quasiquote	quasi quotation	*non implantée*
quote	quotation	§ 1.5.2, p. 11
set !	affectation de variable	§ 1.5.2, p. 11
unquote	quasi quotation	*non implantée*
unquote-splicing	quasi quotation	*non implantée*

Les formes spéciales correspondent aux structures de contrôle des langages de programmation traditionnels (séquence, déclaration, affectation, instructions conditionnelles, etc.).

Notons que cette mini-implantation de Scheme ne propose pas la quasi-quotation (forme spéciales **`quasiquote`**, **`unquote`**, **`unquote-splicing`**), ni la forme itérative **`do`**.

1.5.1.2 Application de fonction

Une application de fonction est une liste dont le premier élément ne dénote pas un mot-clef du langage. La syntaxe d'une telle forme est la suivante :

$$(exp_0\, exp_1\, exp_2 \ldots exp_n)$$

dans laquelle les exp_i sont des formes SCH, c'est-à-dire des constantes, des symboles ou des listes.

L'interprète SCH évalue ces expressions dans un ordre non spécifié, fournissant un ensemble de valeurs val_0, val_1,... val_n. La première de ces valeurs doit être une procédure à n paramètres. Cette procédure est alors appliquée aux arguments val_1,... val_n, fournissant le résultat final de l'application.

1.5.1.3 Notes

Les identificateurs utilisés dans le langage Scheme font appel à nombre de caractères habituellement interdits dans les langages de programmation plus traditionnels : **-**, **?**, **!**, **+**, **<**, **>**, etc. La norme du langage utilise cette possibilité pour suggérer un certain nombre de conventions qui facilitent la lecture des programmes. En voici quelques unes :

- le caractère « **!** » est utilisé à la fin d'un nom de procédure pour indiquer que cette procédure réalise une modification physique sur l'un de ces paramètres : **set !**, **set-car !**, **set-cdr !**, **reverse !**...
- le caractère « **?** » est utilisé à la fin d'un nom de prédure pour indiquer que cette procédure est un prédicat, c'est-à-dire qu'elle fournit comme résultat une valeur logique, utilisable par exemple pour réaliser un test : **zero ?**, **positive ?**, **number ?**, **null ?**...
- le caractère « **-** » est utilisé au sein d'un nom représentant un groupe nominal ou un groupe verbal, afin d'en décomposer la lecture pour la rendre plus lisible : **read-char**, **string-length**, et même **call-with-current-continuation**...
- la séquence « -> » indique une conversion d'un type vers un autre : **char->integer**, **identifier->symbol**, **string->list**, etc.

Il est naturellement conseillé d'utiliser ces conventions lors de la création de variables par l'utilisateur...

1.5.2 Formes spéciales

Cette partie décrit les formes spéciales implantées dans le système. La syntaxe et la sémantique sont propres à chaque forme.

(and *forme forme...*) La forme spéciale **and** évalue séquentiellement ses paramètres, jusqu'à en trouver un dont l'évaluation rendre la valeur *false*. Dans ce cas, il n'y a pas évaluation des expressions qui suivent, et le résultat de la forme est la valeur *false*. Si aucune des expressions ne rend la valeur *false*, le résultat est la valeur de la dernière expression exécutée :

```
(and (> 3 2) #t 6)
;=  6
(and (< 3 2) #t 6)
;=  #f
```

(begin *forme forme...*) La forme spéciale **begin** évalue séquentiellement ses paramètres. Le résultat est la valeur de la dernière expression exécutée :

```
(begin (+ 2 3) (* 5 6))
;=  30
(begin (display "Hello!") (newline) (max 2.3 7.8))
Hello!
;=  7.8
```

(case *clef* $clause_1$ $clause_2$...) Dans cette forme spéciale, le premier argument, ***clef***, est une expression quelconque. Les autres arguments sont des clauses, dont la syntaxe est :

(liste exp_1 exp_2 exp_n***)***

dans laquelle le premier élément est une liste de valeurs, ou le mot-clef ***else***, et les autres éléments sont expressions quelconques.

La forme spéciale **case** évalue en premier l'argument, ***clef***, dont le résultat est une valeur qui sera recherchées dans la liste des clauses successives de la forme ***case***. Cette recherche s'effectue au moyen de la procédure **memv** (le prédicat de comparaison utilisé étant donc **eqv ?**). Dès que la valeur de ***clef*** est trouvée dans l'une des listes, la clause correspondante est sélectable pour exécution. Une clause est également sélectable si son premier élément est, non pas une liste, mais le mot-clef ***else*** (il convient donc de placer une telle clause en dernier de la liste). La première clause sélectable est exécutée, c'est-à-dire que les éléments successifs exp_i sont évalués en séquence, et le résultat de la dernière évaluation devient le résultat de la forme ***case***.

(cond *clause*$_1$ *clause*$_2$...) Dans cette forme spéciale, les arguments sont des clauses, qui vont être successivement testées. Dès que le test associé à l'une des clauses est *vrai*, la clause est exécutée, et son résultat est fourni comme résultat de la forme *cond*. Les clauses obéissent à l'une des trois syntaxes suivantes :

1. ***(test exp*$_1$ *exp*$_2$ *... exp*$_n$*)***
2. ***(test => fun)***
3. ***(else exp*$_1$ *exp*$_2$ *exp*$_n$*)***

Dans la première syntaxe, la forme ***test*** est une expression quelconque, et le test de la clause consiste à exécuter cette expression. Si elle rend vrai, les ***exp***$_i$ sont exécutées en séquence, et le résultat de la dernière devient le résultat de la forme *cond*.

Dans la deuxième syntaxe, la forme ***test*** est une expression quelconque, et le test de la clause consiste également à exécuter cette expression. Si elle rend *vrai*, c'est-à-dire une valeur différente de **#f**, la forme ***fun*** est exécutée, et sont résultat, qui doit être une procédure, est appliqué au résultat de la forme ***test*** , fournissant à son tour une valeur qui devient le résultat de la forme *cond*.

Dans la troisième syntaxe, la clause est toujours vraie ; les ***exp***$_i$ sont exécutées en séquence, et le résultat de la dernière devient le résultat de la forme *cond*. Ce troisième type de clause ne peut apparaître qu'une fois dans une forme *cond*, et toujours en dernière position.

Exemples

```
(cond ((positive? 7) 'ok) ((odd? 3) 'bad))
;= ok
(cond ((assv 'b '((a 1) (b 2) (c 3))) => cadr) (else #f))
;=  2
```

(define *identificateur forme*) La forme spéciale **define** permet la création d'une liaison *nom-valeur*. Le premier paramètre de l'opération est un identificateur, le second une expression dont la valeur, après exécution, est affectée à l'identificateur. Si celui-ci existait déjà, la forme se comporte comme une affectation (c.f. forme **set !**, §1.5.2, p. 11).

```
(define res (* 5 6))
;=  res
(define fun (lambda (p) (* 2 p)))
;=  fun
(fun res)
;=  60
```

L'opération **define** admet aussi une simplification de la syntaxe dans le cas de la définition d'une procédure, et la notation :

```
(define (fun p) (* 2 p))
```

est identique à :

```
(define fun (lambda (p) (* 2 p)))
```

Ceci se généralise bien entendu à une procédure à *n* paramètres.

(delay *forme*) La forme spéciale **delay** accepte comme paramètre une expression qui ne va pas être évaluée, mais dont le calcul sera retardé. Le résultat de l'opération est une *promesse* (*promise*). Cette promesse peut être évaluée par la procédure **force**, qui est nilpotente sur tout autre objet :

```
(define toto (delay (+ 2 3)))
;=  toto
toto
;=  #<DELAY :0 :(+ 2 3)>
(force toto)
;=  5
toto
;=  5
```

(if *condition forme*1 *forme*2) La forme spéciale **`if`** représente une *instruction conditionnelle* qui admet trois opérandes, représentant respectivement un test, une forme à évaluer si le test rend vrai (*forme*1), et une forme à évaluer si le test rend faux (*forme*2). Dans tous les cas, seule l'une des deux formes (*forme*1 ou *forme*2) est évalué.

```
(if #t (+ 2 4) (/ 3 0))
;=  6
(if #f (+ 2 4) (/ 3 0))
division par 0
```

(lambda *paramètres forme*) La forme spéciale **`lambda`** prend comme premier opérande une liste, éventuellement vide, d'identificateurs, ou un identificateur unique, et comme second opérande une forme quelconque, et définit une procédure admettant autant de paramètres qu'il y a d'identificateurs dans la liste. Lors de l'appel de cette procédure avec une liste de valeurs, la forme fournie comme second opérande est exécutée dans le contexte de définition de la procédure, contexte enrichi d'un ensemble de liaisons correspondant aux identificateurs de la liste associés aux valeurs d'appel.

Le résultat de l'exécution est donc une valeur fonctionnelle, qui peut être affectée à une variable ou exécutée directement :

```
(define toto (lambda (x y) (+ x (* 2 y))))
;=  toto
(toto 3 7)
;=  17
((lambda (x y) (+ x (* 2 y))) 3 7)
;=  17
```

(let *liste forme*$_1$ *forme*$_2$...**)**
(letrec *liste forme*$_1$ *forme*$_2$...**)** Les formes spéciales let et letrec permettent la création d'environnements locaux, définis par ***liste***, *qui e*st une liste de couples symbole/expression placés entre parenthèses. Lors de l'exécution d'une forme *let* ou *letrec*, les symboles définissent les variables locales, les expressions les valeurs qui leur seront affectées. L'ensemble constitue un nouvel environnement, au sein desquels sont évaluées les expressions ***forme***$_1$; les deux deux formes rendent comme résultat la valeur de la dernière forme évaluée.

Les deux formes diffèrent en ce sens que la forme *let* évalue les expressions qui vont donner une valeur aux variables locales à l'extérieur de la forme *let*, dans l'environnement global, tandis que *letrec* évalue les expressions à l'intérieur de la forme *letrec*, permettant par exemple la définition de procédures locales à récursion croisée.

```
(let ((x 2) (y (+ 3 6))) (* x y))
;= 18
(let ((x 2) (y (+ 3 6)))
   (let ((x 5) (z (+ x y)))
       (* x z)))
;=  55
(define fact 7)
;=  fact
fact
;=  7
(letrec ((fact
            (lambda (x)
               (if (= x 0)
                   1
                   (* x (fact (- x 1)))))))
   (fact 5))
;=  120
fact
;=  7
```

(or *forme forme*...) La forme spéciale **`or`** évalue séquentiellement ses paramètres, jusqu'à en trouver un dont l'évaluation rendre une valeur différente de *false*. Dans ce cas, il n'y a pas évaluation des expressions qui suivent, et le résultat de la forme est cette valeur. Si aucune des expressions ne rend la valeur *true*, le résultat est la valeur de la dernière expression exécutée :

```
(or (< 3 2) #f 6 (/ 4 0))
;=  6
(or (< 3 2) #f (= 2 7))
;=  #f
```

(quote *forme*) La forme spéciale **`quote`** prend comme paramètre un objet unique et le retourne sans l'évaluer. C'est la forme qui permet de désigner un *littéral* dans le corps d'un programme.

```
(quote (+ 2 3))
;=  (+ 2 3)
(quote toto)
;=  toto
(quote (quote (* 5 6)))
;=  (quote (* 5 6))
```

La forme quote admet aussi d'être abrégée sous la forme d'une apostrophe :

```
'(+ 2 3)
;=  (+ 2 3)
'toto
;=  toto
"(* 5 6)
;=  (quote (* 5 6))
```

(set ! *identificateur forme*) La forme spéciale **`set !`** permet la modification d'une liaison *nom-valeur*. Le premier paramètre de l'opération est un identificateur, le second une expression dont la valeur, après exécution, est affectée à l'identificateur. Celui-ci doit avoir été antérieurement défini au moyen d'une forme *define*, ou encore la forme *set !* doit se trouver à l'intérieur d'une forme *let* ou *letrec* définissant l'identificateur comme variable locale. Ex :

```
(define toto 5)
;= toto
toto
;= 5
(set ! toto 8)
;= 8
toto
;= 8
(let ((jojo 3))
   (set ! jojo 5)
   (* jojo toto))
;= 40
(set ! bobo 3)
Undefined variable : bobo
```

1.5.3 Prédicats

Les *prédicats* représentent une classe d'opérations qui rendent des valeurs *booléennes*, c'est à dire *vrai* ou *faux*. Ces valeurs sont représentées par les notation **`#t`** (vrai) et **`#f`** (faux). Elles n'ont pas besoin d'être quotées :

```
'#t
;=  #t
#t
;=  #t
#f
;=  #f
```

Scheme étant un langage dont les données sont typées, mais non les variables, un premier ensemble de prédicats permet de tester l'appartenance d'un objet à un type. Ces types sont au nombre de dix, et recouvrent des ensembles disjoints d'objets : *null*, *booléens*, *symboles*, *paires*, *nombres*, *caractères*, *vecteurs*, *procédures*, *ports d'entrée-sortie*, *chaînes de caractères*. Ces prédicats s'appliquent à des objets quelconques du système, et fournissent la valeur *vrai* si leur paramètre est du type attendu, la valeur *faux* sinon.

1.5.3.1 Prédicats sur les types

(null ? *objet*) Ce prédicat rend vrai si son paramètre est l'objet *null*, faux sinon.

```
(null ? '())
;=  #t
(null ? 0)
;=  #f
```

(boolean ? *objet*) Le résultat est une valeur booléenne, indiquant si le paramètre est une valeur booléenne.

```
(boolean ? #f)
;=  #t
(boolean ? (> 3 4))
;=  #t
(boolean ? boolean ?)
;=  #f
```

(symbol ? *objet*) Le résultat est une valeur booléenne, indiquant si le paramètre est un symbole.

```
(symbol ? #f)
;=  #f
(symbol ? 'hello)
;=  #t
(symbol ? 246)
;=  #f
```

(pair ? *objet*) Ce prédicat rend vrai si son paramètre est une paire, faux sinon.

```
(pair ? (cons 3 5))
;=  #t
(pair ? 8)
;=  #f
```

(number ? *objet*) Ce prédicat rend *true* si son paramètre est un nombre, *false* sinon.

```
(number ? 7)
;=  #t
(number ? 3.25)
;=  #t
```

(char ? *objet*) Le résultat est une valeur booléenne, indiquant si le paramètre est un caractère.

```
(char ? 25)
;=  #f
(char ? #\A)
;=  #t
(char ? "X")
;=  #f
```

(vector ? *objet*) Le résultat est une valeur booléenne, indiquant si le paramètre est un vecteur.

```
(vector ? 'Hello)
;=  #f
(vector ? #(a b c))
;=  #t
(vector ? "hello")
;=  #f
```

(procedure ? *objet*) Le résultat est une valeur booléenne, indiquant si le paramètre est une procédure.

```
(procedure? 'x)
;=  #f
(procedure? #t)
;=  #f
(procedure? car)
;=  #t
(procedure? vector?)
;=  #t
(procedure? (lambda (x) (+ x 1)))
;=  #t
```

(port ? *objet*) Le résultat est une valeur booléenne, indiquant si le paramètre est un port d'entrée-sortie.

```
(port? 'hello)
;=  #f
(port? 1)
;=  #f
(current-error-port)
;=  #<PORT :84>
(port? (current-error-port))
;=  #t
```

(string ? *objet*) Le résultat est une valeur booléenne, indiquant si le paramètre est une chaîne de caractères.

```
(string? #t)
;=  #f
(string? 'hello)
;=  #f
(string? "hello")
;=  #t
(string? 123)
;=  #f
```

(environment ? *objet*) Le résultat est une valeur booléenne, indiquant si le paramètre est un environnement.

```
(environment? #t)
;=  #f
(scheme-report-environment)
;=  #<ENV :-151150872 :-151130136>
(environment? (scheme-report-environment))
;=  #t
```

1.5.3.2 Autres prédicats

(not *objet*) Ce prédicat s'applique à un objet quelconque du système ; il fournit la valeur *vrai* si son paramètre est l'objet **`#f`**, la valeur *faux* sinon.

```
(not #t)
;=  #f
(not 253)
;=  #f
(not #f)
;=  #t
```

1.5.4 Opérations arithmétiques

1.5.4.1 Nombres

Les opérations arithmétiques portent sur des nombres entiers. Les valeurs numériques sont des constantes qui n'ont pas besoin d'être quotées.

```
'25
;=  25
25
;=  25
(+ '10 '78)
;=  88
(+ 10 78)
;=  88
```

Le système sch reconnaît des *nombres entiers* ou *flottants*. Ces valeurs numériques sont compatibles avec les représentations des **long** ou des **double** sur les machines sur lesquelles est porté sch.

L'intervalle de définition des nombres entiers est :

```
[-2147483648   2147483647]
```

Les nombres entiers peuvent être introduis sous forme décimale, hexadécimale, octale, ou binaires. Le format s'indique en utilisant explicitement les préfixes **#d**, **#x**, **#o** et **#b**.

```
#d1234
;=  1234
#o6234
;=  3228
#x1234
;=  4660
#b10111
;=  23
```

Les nombres flottants sont notés avec une partie décimale, et/ou avec une partie exponentielle :

```
125.23
;=  125.23
-5e4
;=  -50000
1723e-2
;=  17.23
```

Un nombre au format entier, hors des limites de représentation des **long**, est également représenté au format flottant :

```
2147483649
;=  2147483649
```

1.5.4.2 Opérations arithmétiques

Les opérations proposées par le système SCH s'appliquent pour la plupart à des nombres entiers comme à des nombres flottants. Lorsque l'opération s'applique à des opérandes de natures différentes, les entiers sont convertis en flottants au préalable.

Les opérations suivantes sont définies :

(number ? *objet*) Ce prédicat rend *true* si son paramètre est un nombre, *false* sinon.

```
(number? 7)
;=  #t
(number? 3.25)
;=  #t
```

(integer ? *objet*) Ce prédicat rend *true* si son paramètre est un entier, *false* sinon.

```
(integer? 7)
;=  #t
(integer? 3.25)
;=  #f
```

(>= *nombre nombre*)
(<= *nombre nombre*)
(> *nombre nombre*)
(< *nombre nombre*)
(= *nombre nombre*)
(!= *nombre nombre*) Ces opérations effectuent des comparaisons numériques, rendant une valeur booléenne.

```
(> 8 4)
;=  #t
(<= 2 5.5)
;=  #t
(= 2.34 2.34)
;=  #t
```

(<=> *nombre nombre*) Comparaison des deux nombres passés en opérandes. Le résultat est négatif si le premier nombre est inférieur au second, nul si les deux nombres sont égaux, positif si le premier nombre est supérieur au second.

```
(<=> 3 5.5)
;=  -1
(<=> 3 3)
;=  0
(<=> 3 2.5)
;=  1
```

(- *nombre nombre*) Différence des deux nombres passés en opérandes.

```
(- 723 913)
;= -190
```

(/ *nombre nombre*) Quotient entier des deux nombres passés en opérandes. Si l'un des nombres est flottant, le résultat est le quotient exact de des deux nombres :

```
(/ 37 5)
;=  7
(/ 37. 5)
;=  7.4
```

(* *nombre nombre*) Produit des deux nombres passés en opérandes.

(+ *nombre nombre*) Somme des deux nombres passés en opérandes.

(modulo *nombre nombre*) Modulo des deux nombres passés en opérandes.

```
(modulo 7 3)
;=  1
(modulo -7 3)
;=  2
(modulo 7 -3)
;=  -2
(modulo -7 -3)
;=  -1
```

(remainder *nombre nombre*) Reste de la division des deux nombres passés en opérandes.

```
(remainder 7 3)
;=  1
(remainder 7 -3)
;=  1
(remainder -7 3)
;=  -1
```

```
(remainder -7 -3)
;=  -1
```

(abs *nombre*) Valeur absolue du nombre passé en opérande.

(max *nombre nombre...*) La fonction admet un nombre arbitraire d'éléments, et fournit en résultat le plus grand de ceux-ci. Lorsque la fonction est appelée sans paramètre, elle fournit la plus petite valeur numérique représentable sur la machine :

```
(max 2 3 6.43 3 4.4 5)
;=  6.43
(max)
;=  -inf
```

(min *nombre nombre...*) La fonction admet un nombre arbitraire d'éléments, et fournit en résultat le plus petit de ceux-ci. Lorsque la fonction est appelée sans paramètre, elle fournit la plus grande valeur numérique représentable sur la machine :

```
(min 2 3 6.43 3 4.4 5)
;=  2
(min)
;=  inf
```

(neg *nombre*) Opposé du nombre passé en opérande.

(negative ? *nombre*) Prédicat retournant *vrai* si son paramètre est négatif, *faux* sinon.

(positive ? *nombre*) Prédicat retournant *vrai* si son paramètre est positif, *faux* sinon.

(zero ? *nombre*) Prédicat retournant *vrai* si son paramètre est nul, *faux* sinon.

(odd ? *nombre*) Prédicat retournant *vrai* si son paramètre est impair, *faux* sinon.

(even ? *nombre*) Prédicat retournant *vrai* si son paramètre est pair, *faux* sinon.

1.5.4.3 Fonctions transcendantes

Scheme propose également la panoplie habituelle de fonctions arithmétiques. Voici la liste de celles qui dont disponibles dans SCH :

(sin *nombre*)
(cos *nombre*)
(tan *nombre*)
(exp *nombre*)
(log *nombre*)
(sqrt *nombre*)
(asin *nombre*)
(acos *nombre*)
(atan *nombre*) Ces procédures implémentent les fonctions transcendantes (arithmétiques, trigonométriques, etc) habituelles. Contrairement à ce que prescrit la norme du langage, ces fonctions ne fournissent pas des valeurs complexes (appels `(sqrt -1)`, `(log -1)`), mais des *nans*. Elles sont réalisées, dans l'implantation SCH, par utilisation de la bibliothèque mathématique du langage C sur des données de type *flottant double*, et en partagent donc les qualités et les défauts...

(isinf ? *nombre*)
(isnan ? *nombre*) Ces procédures indiquent si leur paramètre est une *valeur infinie* ou un *nan* (not a number) :

```
(log 0)
;=  -inf
(log -7)
;=  nan
(sqrt -2)
;=  nan
(isinf? (log 0))
;=  -1
(isinf? 3)
;=  #f
(isinf? (log -7))
;=  #f
(isnan? (log -7))
;=  1
(isnan? (sqrt -2))
;=  1
(isnan? (log 0))
;=  #f
```

1.5.5 Opérations sur caractères

Les caractères sont représentés par une notation spécifique, consistant en une séquence *dièse-backslash*, **#**, suivie d'un caractère. Les caractères sont des constantes, et n'ont pas besoin d'être *quotés*.

```
'#\A
;=  #\A
#\A
;=  #\A
```

(char ? *objet*) Ce prédicat indique si son paramètre est un objet de type caractère.

```
(char? #\A)
;=  #t
(char? 2.5)
;=  #f
```

(char *objet*) Cette opération convertit son paramètre en un caractère.

```
(char 65)
;=  #\A
(char 233)
;=  #\é
(char -23)
;=  #\é
```

(char-downcase *char*) Cette opération fournit le caractère minuscule correspondant au caractère majuscule passé en paramètre. Le paramètre doit être un objet de type numérique ou caractère.

```
(char-downcase #\a)
;=  #\A
(char-downcase #\É)
;=  #\é
(char-downcase #\Z)
;=  #\z
```

(char-upcase *char*) Cette opération fournit le caractère majuscule correspondant au caractère minuscule passé en paramètre. Le paramètre doit être un objet de type numérique ou caractère.

```
(char-upcase #\a)
;=  #\A
(char-upcase #\é)
;=  #\É
(char-upcase #\Z)
;=  #\Z
```

(char-alphabetic ? *objet*) Ce prédicat indique si son paramètre est un objet de type caractère ou numérique représentant une lettre.

```
(char-alphabetic? #\a)
;=  #t
(char-alphabetic? #\+)
;=  #f
(char-alphabetic? 65)
;=  #t
(char-alphabetic? 127)
;=  #f
(char-alphabetic? #t)
char expected : #t
```

(char-lower-case ? *objet*)
(char-upper-case ? *objet*) Ces prédicats indiquent si leurs paramètres sont des objets de type caractère ou numérique représentant une lettre majuscule ou minuscule selon les cas.

```
(char-upper-case? #\E)
;=  #t
(char-lower-case? #\e)
;=  #t
(char-upper-case? #\+)
;=  #f
(char-upper-case? 65)
;=  #t
(char-lower-case? "Hello")
char expected : "Hello"
```

1.5.6 Opérations sur paires

1.5.6.1 Paires

La structure de donnée de base de Lisp est la *paire*, dite aussi *cons* ou *paire-pointée*. Une paire est créée dynamiquement pas l'opération *cons* :

(cons *objet objet*) L'opération prend deux paramètres, des objets quelconques, et fournit une paire nouvelle contenant ces objets :

```
(cons 3 5)
;=  (3 . 5)
(cons "abc" #\A)
;=  ("abc" . #\A)
(cons #f 23.45)
;=  (#f . 23.45)
```

Une paire contient deux champs, le *car* et le *cdr*. Les opérations **`car`** et **`cdr`** permettent d'accéder à ces champs :

(car *paire*) L'opération prend comme paramètre une paire, et fournit la valeur du *car* de cette paire.

```
(car (cons "abc" #\A))
;= "abc"
(car (cons (+ 2 3) (* 5 6)))
;= 5
```

(cdr *paire*) L'opération prend comme paramètre une paire, et fournit la valeur du *cdr* de cette paire.

```
(cdr (cons "abc" #\A))
;= #\A
(cdr (cons (+ 2 3) (* 5 6)))
;= 30
```

(set-car ! *paire expr*) L'opération prend comme premier paramètre une paire, dans laquelle le *car* va être physiquement remplacé par la valeur de la seconde expression :

```
(define p '(a . b))
;= p
p
;= (a . b)
(set-car! p (+ 3 5))
;= (8 . b)
p
;= (8 . b)
```

(set-cdr ! *paire expr*) L'opération prend comme premier paramètre une paire, dans laquelle le *cdr* va être physiquement remplacé par la valeur de la seconde expression :

```
(define q '(x . y))
;= q
(set-cdr! q 'z)
;= (x . z)
q
;= (x . z)
```

(caar *paire*)
(cadr *paire*)
...
(cddddr *paire*) Le langage fournit l'ensemble des combinaisons des opérations *car* et *cdr* jusqu'à quatre niveaux. **`cadr`** est ainsi le *car* du *cdr* d'une structure, **`cdddar`** le *cdr* du *cdr* du *cdr* du *car* d'une structure, etc...

Une notation spécifique existe pour la représentation des paires : le *car* et le *cdr* sont placés entre parenthèses, séparés par un point :

```
(a . b)
```

Un tel objet doit être *quoté* afin d'être reconnu comme une constante :

```
'(a . b)
;= (a . b)
```

(pair ? *objet*) Ce prédicat rend vrai si son paramètre est une paire, faux sinon.

```
(pair? (cons 3 5))
;= #t
(pair? 8)
;= #f
```

1.5.6.2 Listes

Une liste est une combinaison d'objets du langage Scheme qui répond à la définition suivante : une *liste* est soit l'objet spécial *null*, représentant la liste vide, noté **()**, soit une *paire* dont le *cdr* est une liste.

L'objet *null* n'a pas besoin d'être quoté dans le langages sch :

```
'()
;=  ()
()
;=  ()
```

Une notation particulière existe pour la représentation des objets du type listes : les éléments d'une liste sont placés entre parenthèses, séparés les uns des autres par des blancs, des passages à la ligne, ou des ponctuations du langage :

```
(voici une liste de 6 éléments)
```

Pour être considérée comme une constante par l'interprète, une liste doit être quotée :

```
'(1 2 3)
;=  (1 2 3)
'(+ 2 3)
;=  (+ 2 3)
'((une liste) contenant (d'autres listes))
;=  ((une liste) contenant (d (quote autres) listes))
```

Une liste est, en accord avec sa définition, une structure à base de paires pointées, et peut être ainsi introduite dans le système :

```
'(1 . (2 . (3 . ())))
;=  (1 2 3)
```

Lorsque possible, le système essaye de faire apparaître sous forme de liste les structures construites à base de paires pointées :

```
(cons 2 (cons (cons 5 (cons 7 ())) (cons 8 (cons 9 10))))
;=  (2 (5 7) 8 9 . 10)
```

La structure créée par cette instruction n'est pas une *liste* selon notre définition : l'objet le « plus à droite » de la structure n'est pas l'objet *null*, mais le nombre 10. Ceci est dénoté par l'apparition d'un *point* précédant le nombre dans l'impression du résultat.

(null ? ***objet*****)** Ce prédicat rend vrai si son paramètre est l'objet *null*, faux sinon.

```
(null? '())
;=  #t
(null? 0)
;=  #f
```

(append ***objet objet objet…*****)** L'opération crée une nouvelle liste, en plaçant bout à bout les contenus des listes passées en paramètre, sauf le dernier objet, qui n'est pas recopié, mais devient la fin de la liste.

```
(append '(a b c) '(x y))
;=  (a b c x y)
(append '(a b c) '(x y) 'z)
;=  (a b c x y . z)
(append '(a b c))
;=  (a b c)
```

Il est possible de créer une copie physique d'une liste au moyen de l'opération **append** :

```
(define l '(a b c d))
;=  l
(append l '())
;=  (a b c d)
```

(list *objet objet objet...*) L'opération fournit une liste contenant les objets passés en paramètres. L'opération accepte un nombre quelconque d'éléments.

```
(list 2 3 (+ 5 7))
;=  (2 3 12)
(list "ABC" #\A (> 3 4))
;=  ("ABC" #\A #f)
(list)
;=  ()
```

(list ? *objet*) Ce prédicat indique si son paramètre est une liste.

```
(list ? '())
;=  #t
(list ? 24)
;=  #f
(list ? '(1 2 3))
;=  #t
(list ? (cons 'a (cons 'b ())))
;=  #t
(list ? (cons 'a (cons 'b 'c)))
;=  #f
```

On notera, dans le dernier exemple, que le paramètre du prédicat est bien une structure à base de *paires*, mais que le *cdr* de la dernière paire n'est pas *null*.

(length *objet*) Cette procédure fournit, si son paramètre est une liste, la longueur de celle-ci, 0 dans le cas contraîre.

```
(length '(a b c d))
;=  4
(length '())
;=  0
(length 23)
;=  0
```

(reverse *objet*) Cette procédure s'applique à une liste et fournit une nouvelle liste contenant les éléments de la première en ordre inverse. La fonction ne modifie pas le paramètre :

```
(define p '(a b c d e f))
;= p
(reverse p)
;= (f e d c b a)
p
;= (a b c d e f)
(reverse '())
;= ()
(reverse 23)
pair expected : 23
```

(reverse ! *objet*) Cette procédure s'applique à une liste et fournit une nouvelle liste contenant les éléments de la première en ordre inverse. La fonction transforme physiquement le paramètre, afin de réutiliser les paires le composant :

```
(define p '(a b c d e f))
;= p
(define q (reverse ! p))
;= q
q
;= (f e d c b a)
p
;= (a)
```

```
(reverse! q)
;= (a b c d e f)
q
;= (f)
p
;= (a b c d e f)
```

1.5.7 Recherche et comparaisons

(eq ? ***exp*$_1$** ***exp*$_2$**) Cette opération teste l'identité de ses arguments. Ceux-ci sont considérés comme identiques s'ils désignent le même objet physique (par exemple, les deux expressions sont des références à la même variable), ou s'ils désignent le même atome. Dans certains autres cas, le résultat est laissé à la discrétion de l'implémentation : deux nombres de même valeur peuvent être considérés comme égaux, ou non. Voici quelques exemples d'utilisation de la procédure :

```
(eq? (quote toto) (quote toto))
;=  #t
(eq? '() '())
;=  #t
(eq? 125 125)
;=  #f
(eq? #\a #\a)
;=  #f
(define num 125)
;=  num
(eq? num num)
;=  #t
(eq? "Hello" "Hello")
;=  #f
(eq? '(a b c d) '(a b c d))
;=  #f
(define li '(a b c d))
;=  li
(eq? li li)
;=  #t
(eq? (caddr li) 'c)
;=  #t
(eq? (cddr li) (cddr li))
;=  #t
```

(eqv ? ***exp*$_1$** ***exp*$_2$**) Cette opération teste l'équivalence de ses arguments. Ceux-ci sont considérés comme identiques s'ils désignent le même objet physique (par exemple, les deux expressions sont des références à la même variable), le même atome, ou encore des entiers ou caractères de même valeur. Dans certains autres cas, le résultat est laissé à la discrétion de l'implémentation : deux nombres flottants ou deux chaînes de caractères de même valeur peuvent être considérés comme égaux, ou non. Voici quelques exemples d'utilisation de la procédure :

```
(eqv? (quote toto) (quote toto))
;=  #t
(eqv? 125 125)
;=  #f
(eqv? (* 2 3) (- 8 2))
;=  #t
(eqv? 23.5 23.5)
;=  #f
(eqv? #\a #\a)
;=  #t
(define num 125)
;=  num
(eqv? num num)
;=  #t
```

```
(eqv? "Hello" "Hello")
;=  #f
(eqv? '(a b c d) '(a b c d))
;=  #f
(define li '(a b c d))
;=  li
(eqv? li li)
;=  #t
(eqv? (caddr li) 'c)
;=  #t
(eqv? (cddr li) (cddr li))
;=  #t
```

(equal ? ***exp*$_1$** ***exp*$_2$)** Cette opération teste l'égalités de ses arguments. Deux objets simples sont égaux s'ils ont même valeur. Deux listes sont égales si elles ont même longueur et si leurs éléments correspondants sont égaux. Voici quelques exemples d'utilisation de la procédure :

```
(equal? (quote toto) (quote toto))
;=  #t
(equal? (* 2 3) (- 8 2))
;=  #t
(equal? 23.5 23.5)
;=  #t
(equal? #\a #\a)
;=  #t
(equal? "Hello" "Hello")
;=  #t
(equal? '(a b c d) '(a b c d))
;=  #t
(equal? '(a b c) '(a b c d))
;=  #f
```

(memq *expr liste*)
(memv *expr liste*)
(member *expr liste*) Ces trois opérations recherchent leur premier argument, qui peut être une valeur quelconque, dans le second argument, supposé être une liste. Si cet argument n'est pas une liste, ou si le premier argument n'est pas trouvé, la procédure rend la valeur *false*. Si le premier argument est trouvé dans le second, le résultat est la partie de la liste constituant le second argument qui commence par la première occurrence de la valeur cherchée. Les trois procédures diffèrent uniquement par la fonction de comparaison utilisée, qui est ***eq ?*** pour ***memq***, ***eqv ?*** pour ***memv***, et ***equal ?*** pour ***member*** :

```
(memq 'c '(a b c d e))
;=  (c d e)
(memq 5 '(2 3 5 8 9))
;=  #f
(memv 5 '(2 3 5 8 9))
;=  (5 8 9)
(memv "abc" '("xy" "abc" 5 (1 2 3) hello))
;=  #f
(member "abc" '("xy" "abc" 5 (1 2 3) hello))
;=  ("abc" 5 (1 2 3) hello)
(member '(1 2 3) '("xy" "abc" 5 (1 2 3) hello))
;=  ((1 2 3) hello)
(memv '(1 2 3) '("xy" "abc" 5 (1 2 3) hello))
;=  #f
```

1.5.7.1 Listes associatives

Ce terme recouvre des listes dont chaque élément est un couple *nom-valeur*. Les *listes associatives* ont une importance historique, en ce sens qu'elles servaient dans les premiers systèmes lisp, à représenter les *variables* du

langage. L'implantation utilisée, `sch`, utilise d'ailleurs cette représentation, les références aux objets étant éventuellement optimisées dynamiquement lors de la première exécution d'un code. La notation :

```
'((a . 1) (b . 2) (c 3 4 5) (d . "Hello"))
;=  ((a . 1) (b . 2) (c 3 4 5) (d . "Hello"))
```

pourrait ainsi représenter une variable `a`, de valeur `1`, une variable `b`, de valeur `2`, une variable `c`, de valeur `(3 4 5)`, etc.

Trois fonctions permettent de rechercher un couple, à partir de la valeur du *car* du couple :

(assoc expr liste)

(assv expr liste)

(assq expr liste) Ces trois opérations recherchent dans **liste** un couple dont le *car* est égal à **expr** ; elles différent par la procédure de comparaison utilisée : `assoc` utilise `equal ?`, `assv` utilise `eqv ?` et `assq` utilise `eq ?`.

```
(assv 'c '((a . 1) (b . 2) (c 3 4 5) (d . "Hello")))
;=  (c 3 4 5)
(cdr (assv 'c '((a . 1) (b . 2) (c 3 4 5) (d . "Hello"))))
;=  (3 4 5)
```

Les procédures rendent le couple complet ; appliquer la procédure `cdr` à ce résultat permet d'obtenir la valeur associée au nom. Les procédures rendent *faux* si l'objet n'est pas trouvé :

```
(assv 'g '((a . 1) (b . 2) (c 3 4 5) (d . "Hello")))
;=  #f
```

1.5.8 Opérations d'entrées-sorties

Le langage Scheme propose des opérations d'entrée ou de sortie ; ces opérations s'effectuent sur des , qui peuvent être liés au terminal (par l'intermédaire de *stdin*, *stdout* ou *stderr*), mais aussi à des fichiers ou des objets du langage, les chaînes de caractères.

1.5.8.1 Les ports

Les ports sont des objets du langage, reconnus par le prédicat **`port ?`**, sur lesquels des opérations de lecture ou d'écriture peuvent être effectuées.

(port ? *objet*) Le résultat est une valeur booléenne, indiquant si le paramètre est un port d'entrée-sortie.

```
(port ? 'hello)
;=  #f
(port ? 1)
;=  #f
(current-error-port)
;=  #<PORT :84>
(port ? (current-error-port))
;=  #t
```

(current-input-port)
(current-output-port)
(current-error-port) Les résultats de ces opérations sont, respectivement, les ports utilisés pour la lecture des entrées, l'écriture des sorties et l'écriture des messages d'erreur de l'interprète. Ces ports sont, par défaut, associés à *stdin*, *stdout* et *stderr*.

(input-port ? *objet*) Le résultat est une valeur booléenne, indiquant si le paramètre est un port d'entrée-sortie susceptible de délivrer des caractères.

```
(current-input-port)
;=  #<PORT :52>
(input-port ? (current-input-port))
;=  #t
```

(output-port ? *objet*) Le résultat est une valeur booléenne, indiquant si le paramètre est un port d'entrée-sortie susceptible de recevoir des caractères.

```
(current-output-port)
;=  #<PORT :84>
(output-port ? (current-output-port))
;=  #t
(output-port ? (current-input-port))
;=  #f
```

1.5.8.2 Création de ports

Quelques procédures permettent d'ouvrir des ports, permettant l'accès à d'autres périphériques que le terminal utilisé pour l'interaction. À l'heure actuelle, seules les opérations suivantes sont implémentées.

(open-output-string) Le résultats de cette opération est un port, ouvert en écriture, qui est susceptible d'accumuler les caractères qui lui sont transmis. La fonction **`get-port-string`** permet d'obtenir sous la forme d'une chaîne de caractères l'ensemble de ces valeurs :

```
(define x (open-output-string))
;=  x
x
;=  #<PORT :81>
(write 'hello x)
;=  hello
(write '(a b c) x)
;=  (a b c)
(display (+ 2 4) x)
;=  6
(get-port-string x)
;=  "hello(a b c)6"
(write "Hello World!" x)
;=  "Hello World!"
(get-port-string x)
;=  "hello(a b c)6\"Hello World!\""
```

(open-input-string *expr*) Le paramètre de cette opération doit être une chaîne de caractères. Le résultat est un port ouvert en lecture, susceptible de délivrer les caractères successifs de la chaîne ; le port délivre l'objet « fin de fichier » lorsque l'ensemble des caractères a été lu :

```
(define s (open-input-string
     "125 212 33.450 0000122 1000000000000000e-9 33"))
;= s
s
;= #<PORT :49>
(read s)
;= 125
(read s)
;= 212
(read s)
;= 33.45
(read s)
;= 122
(read s)
;= 1000000
(read s)
;= 33
(read s)
;= #<EOF>
(read s)
```

```
;= #<EOF>
(eof-object? (read s))
;=  #t
```

1.5.8.3 Procédures d'entrées-sorties

Les procédures d'entrée-sortie du langage Scheme utilisent, implicitement ou explicitement, des *ports* comme sources ou destinations.

(read *{port}*) Cette procédure lit un objet du langage (symbole, nombre, chaîne, liste, etc) sur le port d'entrée précisé, par défaut le port associé à *stdin*. Lorsque le port ne peut plus délivrer de caractères, le résultat est l'objet « fin de fichier ».

(read-char *{port}*) Cette procédure lit un caractère sur le port d'entrée précisé, par défaut le port associé à *stdin*. Lorsque le port ne peut plus délivrer de caractères, le résultat est l'objet « fin de fichier ».

1.5.9 Opérations d'application

(apply *fonction {argument . . .} liste*) Cette opération applique son premier argument, une fonction, à une liste de paramètres, qui est obtenue en *consant* les arguments optionnels en tête du dernier argument, qui doit être une liste :

```
(apply + '(3 4))
;=  7
(apply + 3 '(4))
;=  7
(apply cons '(a b))
;=  (a . b)
(apply cons 'a 'b '())
;=  (a . b)
(apply apply apply + 3 4 '((())))
;=  7
```

(force *argument*) Cette opération retourne la valeur de son argument si celui-ci n'est pas une promesse (elle est dite alors *nilpotente*), ou la valeur de l'expression associée à l'argument si celui-ci est une promesse :

```
(define toto (delay (+ 2 3)))
;=  toto
toto
;=  #<DELAY :0 :(+ 2 3)>
(force toto)
;=  5
toto
;=  5
(force "Hello")
;=  "Hello"
```

1.5.10 Opérations diverses non standard

1.5.10.1 Formes spéciales non standard

1.5.10.2 Procédures non standard

Les procédures décrites ici ne font pas partie du standard ; elles sont rendues disponibles à des fins pragmatiques, ou de test, ou pour tout autre raison. . .

(end) Cette opération interrompt l'exécution de l'interprète et retourne au système d'exploitation, probablement le shell à partir duquel l'exécution a été lancée. Elle équivaut à l'appel **`(exit 0)`**.

(exit *argument*) Cette opération interrompt l'exécution de l'interprète et retourne au système d'exploitation, en retournant comme code de retour la valeur de son argument, qui doit être un entier. Une valeur de 0 indique qu'aucune erreur ne s'est produite, une autre valeur signale une erreur. Ce code de retour peut être testé dans un shell, est utilisé par **`make`**, etc.

(rand *argument*) Cette opération fournit un nombre entier pseudo-aléatoire, compris entre 0 (inclus) et son argument (exclus), qui doit être un nombre entier strictement positif. La fonction utilisée est la fonction Unix RAND(3) :

```
(rand 1000)
;=  383
(rand 1000)
;=  886
(rand -10)
domain error -10
(rand 0)
domain error 0
```

(random-seed *argument*) Cette opération permet de définir la « graine », c'est-à-dire le point de départ de la suite pseudo-aléatoire générée par les appels successifs à **`rand`**. Le paramètre doit être un nombre entier. Le résultat est la valeur précédente de la graine.

```
(rand 1000)
;=  383
(rand 1000)
;=  886
(random-seed 1)
;=  1
(rand 1000)
;=  383
(rand 1000)
;=  886
(random-seed  1000)
;=  1
(rand 1000)
;=  790
```

1.6 Scheme, en quelques exemples

Cette partie propose quelques exemples typiques de programmation en Scheme.

1.6.1 Tri

Voici un exemple classique de tri par fusion. Le tri d'une liste de nombre s'effectue par l'appel `(sort liste <)`.

```
(define (sort l less)
    (define (merge l1 l2)
        (if (null? l1)
            l2
            (if (null? l2)
                l1
                (if (less (car l1) (car l2))
                    (cons (car l1) (merge (cdr l1) l2))
                    (cons (car l2) (merge l1 (cdr l2)))))))
    (define (sort-aux l)
        (if (or (null? l) (null? (cdr l)))
            l
            (cut l '() '())))
    (define (cut l l1 l2)
        (if (null? l)
```

```
            (merge (sort-aux l1) (sort-aux l2))
            (cut (cdr l) (cons (car l) l2) l1)))
    (sort-aux l))
```

Voici un exemple d'utilisation :

```
(define l '(2 9 4 6 3 9 1 20 5 2 8 3 5 10 0 4 9 7 3 2 11))
;=  l
(sort l <)
;=  (0 1 2 2 2 3 3 3 4 4 5 5 6 7 8 9 9 9 10 11 20)
```

1.6.2 Exploration d'arbre

Il est quelquefois nécessaire d'explorer une structure de données arbitraire, à la recherche d'éléments satisfaisant un prédicat particulier. La fonction suivante, *make-walker*, construit, étant donné un arbre et un prédicat, une fonction qui explore l'arbre en fournissant les éléments satisfaisant ce prédicat. La fonction présente la particularité de générer un résultat unique à chaque appel, fournissant le résultat *false* à la fin de l'exploration.

```
(define (make-walker T pred?)
    (define (explore S cont!)
        (if (pair? S)
            (explore (car S) (lambda () (explore (cdr S) cont!)))
            (if (null? S)
                (cont!)
                (if (pred? S)
                    (begin (set! glob-cont! cont!) S)
                    (cont!)))))
    (define (glob-cont!)
        (explore T (lambda () #f)))
    (lambda () (glob-cont!)))
```

La structure de données suivante est utilisée pour les tests :

```
(define Tree '(1 2 3 -9 3 (5 6 -2 8 -7 3)
        (9 (11 -3 12) 7 11 (13 (15 (17 ((19 20) -3 -7)
        -11)))) 6 56 -17 (((((((3))))) 7 -8)))))
```

Nous construisons maintenant trois explorateurs : `toto` recherche les éléments *négatifs*, `titi` les éléments *impairs*, et `tutu` les éléments congrus à 1 modulo 5 :

```
(define toto (make-walker Tree negative?))
;=  toto
(define titi (make-walker Tree odd?))
;=  titi
(define tutu (make-walker Tree (lambda (x) (= (modulo x 5) 1))))
;=  tutu
```

Voici les résultats de quelques appels de ces procédures :

```
(toto)
;=  -9
(titi)
;=  1
(tutu)
;=  1
(list (toto) (titi) (tutu))
;=  (-2 3 -9)
(list (toto) (titi) (tutu))
;=  (-7 -9 6)
(list (toto) (titi) (tutu))
;=  (-3 3 11)
```

1.6.3 Un système « expert »

Voici un minuscule système expert, basé sur l'article « Introduction aux systèmes experts » [12] qui décrit une base de données botanique.

Voici la traduction en Scheme de la base de règles, dans laquelle le symbole « - » représente la négation :

```
(define BDD '(
     phanérogame (fleur graine)
     sapin (phanérogame graine-nue)
     monocotyledone (phanérogame 1-cotyledone)
     dicotyledone (phanérogame 2-cotyledone)
     muguet (monocotyledone rhizome)
     anemone (dicotyledone)
     lilas (monocotyledone - rhizome)
     cryptogame (feuille - fleur)
     mousse (cryptogame - racine)
     fougere (cryptogame racine)
     thallophyte (- feuille plante)
     algue (thallophyte chlorophylle)
     champignon (thallophyte - chlorophylle)
     collibacille (- feuille - fleur - plante)))
```

Voici l'ensemble des buts :

```
(define BUTS '(
       sapin muguet anemone lilas mousse
       fougere algue champignon collibacille))
```

Supposons que notre spécimen présente les caractéristiques suivantes :

```
rhizome fleur graine 1-cotyledone
```

Le programme va essayer toutes les hypothèses possibles, en utilisant ces faits, et en ajoutant les conclusions obtenues étape par étape. Selon nos règles, voici les conclusions que l'on peut obtenir :

```
(fleur graine) ⟹ phanérogame
(phanérogame 1-cotyledone) ⟹ monocotyledone
(monocotyledone rhizome) ⟹ muguet
```

Le *muguet* est l'un des buts ; la nature du spécimen est donc déterminée.

Voici le programme SCHEME :

```
(define (deduire vrais faux BDD BUTS)
    (define (tente truc vrais faux)
        (define (valide? proposition vrais faux)
            (or (null? proposition)
                (and (eq? '- (car proposition))
                     (memq (cadr proposition) faux)
                     (valide? (cddr proposition) vrais faux))
                (and (memq (car proposition) vrais)
                     (valide? (cdr proposition) vrais faux))))
        (if (valide? (cadr truc) vrais faux)
            (car truc)
            #f))
    (define (balaye base vrais faux)
        (if (null? base)
            #f
            (essai (tente base vrais faux) base vrais faux)))
    (define (essai fait base vrais faux)
        (if (and fait (not (memq fait vrais)))
            fait
            (balaye (cddr base) vrais faux)))
    (define (itere vrais faux)
        (itere2 (balaye BDD vrais faux) vrais faux))
    (define (itere2 fait vrais faux)
```

```
            (if (not fait)
                #f
                (if (memq fait BUTS)
                    fait
                    (itere (cons fait vrais) faux))))
        (itere vrais faux))
```

Voici enfin quelques exemples d'utilisation de notre programme :

```
(deduire '(rhizome fleur graine 1-cotyledone) '() BDD BUTS)
;=  muguet
(deduire '(fleur graine 2-cotyledone) '() BDD BUTS)
;=  anemone
(deduire '(thallophyte) '(chlorophylle) BDD BUTS)
;=  champignon
(deduire '(chlorophylle) '(fleur feuille rhizome) BDD BUTS)
;=  #f
```

Ce dernier exemple est un cas dans lequel aucune conclusion ne peut être tirée des prémices proposées.

Chapitre 2

Implantation d'un langage fonctionnel : Scheme

2.1 Introduction

Ce chapitre décrit une approche à l'interprétation d'un langage fonctionnel, Scheme.

Qu'est-ce qu'un interprète Scheme ? C'est, à la base, une boucle « infinie » dont le corps a pour fonction de lire une expression, de l'évaluer, et d'imprimer le résultat, ce que nous symboliserons par l'expression :

```
(write (eval (read)))
```

qui est souvent désignée (pour des raisons historiques, « print » étant le nom originel de la procédure « write ») sous le nom de boucle « read-eval-print ».

La procédure **`read`** effectue la lecture d'une *expression symbolique*, dite parfois *s-exp* (*symbolic expression*). Elle a pour tâche de réaliser l'analyse lexicale et syntaxique de l'expression lue, et d'allouer les paires pointées servant à représenter cette expression (y compris ses sous-expressions).

La procédure **`eval`** effectue l'interprétation de la *s-exp* qui lui est passée en paramètre. Nous étudierons plus particulièrement son fonctionnement dans ce chapitre.

La procédure **`write`**, enfin, imprime son paramètre ; son fonctionnement est aisément modélisable par une procédure récursive.

Nous allons proposer plusieurs modèles successifs, implémentant les diverses fonctionnalités du langage.

2.2 Modèle 1

Notre première version sera la suivante :

```
(define (scheme-1)
   (display " :  ")
   (let ((r (eval-1 (read) env)))
       (display "=> ")
       (write r)
       (newline))
   (scheme-1))
```

Cette version (qui réalise l'itération par un appel en récursion terminale), fait appel aux procédures du système **`read`** et **`write`**, mais utilise sa propre version de l'évaluateur, qui est la suivante :

```
(define (eval-1 exp env)
  (cond
    ;; Les constantes manifestes
    ((or (null? exp) (boolean? exp)
         (number? exp) (string? exp)) exp)
    ;; Les symboles
    ((symbol? exp) (get exp env))
    ;; Les listes
    ((list? exp)
```

```
            (apply
                (eval-1 (car exp) env)
                (map (lambda (x) (eval-1 x env)) (cdr exp))))
        (else
           (error "Expression non interpretable : " exp))))
```

Notre première version ne fait plus appel à la procédure eval ; elle accepte deux paramètres, une expression symbolique à évaluer, et un environnement. Si l'expression est une constante (telle un nombre, un booléen, etc), la procédure renvoie directement l'objet. Si c'est un symbole, il représente une variable, dont la valeur va être cherchée dans l'environnement fourni (le second paramètre de la fonction). On notera que cette version ne s'intéresse pas (encore) aux formes spéciales du langage.

Il reste ici à représenter les environnements, et leur manipulation au travers de « `get` », ainsi que la procédure « `apply` ». Les autres procédure utilisées se programment sans difficulté particulière dans un langage tel que C. Une implémentation de « `get` » est la suivante :

```
(define (get sym env)
  (let ((v (assq sym env)))
     (if (pair? v)
         (cdr v)
         (error "Undefined symbol : " sym))))
```

Cette implémentation fait donc appel à un environnement représenté sous forme de liste associative. Un tel environnement peut être représenté par :

```
(define  env (list '(x . 5) (cons '+ +)
   (cons '- -) (cons '* *) (cons '< <) (cons '= =) (cons '> >)
   (cons '<= <=) (cons '>= >=)
   (cons 'cons cons) (cons 'car car) (cons 'cdr cdr)
   (cons 'null? null?) (cons 'pair? pair?) (cons 'list? list?)
   (cons 'number? number?) (cons 'symbol? symbol?)
   (cons 'string? string?) (cons 'boolean? boolean?) (cons 'quit quit)
 ))
```

Le modèle proposé fait bien sûr appel aux opérations primitives du langage (`cons`, `car`, `cdr`, `+`, etc), mais on peut, là aussi aussi, se convaincre que ces opérations sont triviales à implémenter.

Le modèle est directement exécutable ; la « pré-définition » d'une variable, `x`, permet de tester le système :

```
(load "evaluator-1.scm")
   (scheme-1) ; to run the evaluator
;=  #t
(scheme-1)
:  x
=> 5
:  (+ x 3)
=> 8
:  (> (* x 2) 8)
=> #t
:  (quit)
```

L'opération « `quit` » prédéfinie dans `sch` interrompt l'exécution courante (donc l'interprète « `scheme-1` ») et provoque un retour au mode terminal.

Notons encore que la procédure « `apply` », dans la mesure où les paramètres de la fonctions ont été évalués, se ramène à un simple appel de fonction classique ; on verra comment sa définition doit évoluer pour prendre en compte les procédures définies du langage.

2.3 Modèle 2

Notre nouvelle version s'intéresse à l'implantation des formes spéciales. Une forme spéciale est simplement une liste dont l'interprète reconnaît le premier élément comme étant un mot-clef du langage, et à laquelle il associe une sémantique spéciale. Il convient donc de reconnaître chaque forme spéciale, et d'implémenter une procédure spécifique pour l'évaluation. C'est ce qui est fait dans notre seconde version du modèle pour les formes « `quote` » et « `if` » :

```
(define (eval-2 exp env)
  (cond
    ;; Les constantes manifestes
    ((or (null? exp) (boolean? exp)
         (number? exp) (string? exp)) exp)
    ;; Les symboles
    ((symbol? exp) (get exp env))
    ;; Les listes
    ((list? exp)
      (cond
         ;;  Formes speciales?
         ((eq? (car exp) 'quote) (cadr exp))
         ((eq? (car exp) 'if)
            (if (= 4 (length exp))
                (if (eval-2 (cadr exp) env)
                    (eval-2 (caddr exp) env)
                    (eval-2 (cadddr exp) env))
                (error "if syntax error : " exp)))
         (else (apply
                   (eval-2 (car exp) env)
                   (map (lambda (x) (eval-2 x env)) (cdr exp))))))
    (else
       (error "Expression non interpretable : " exp))))
```

Cette version permet donc d'utiliser les formes spéciales « `if` » et « `quote` » :

```
(load "evaluator-2.scm")
   (scheme-2); to run the evaluator
;=  #t
(scheme-2)
: (if (= x 3)
      'curious
      'ok)
=> ok
```

2.4 Modèle 3

La version 3 du modèle s'intéresse à la définition de variables, et ajoute les formes « `define` » et « `set!` ».

```
(define (eval-3 exp env)
  (cond
   ;; Les constantes manifestes
    ((or (null? exp) (boolean? exp)
         (number? exp) (string? exp)) exp)
   ;; Les symboles
    ((symbol? exp) (get exp env))
   ;; Les listes
    ((list? exp)
      (cond
        ;; Formes speciales?
         ((eq? (car exp) 'quote) (cadr exp))
         ((eq? (car exp) 'if)
            (if (= 4 (length exp))
                (if (eval-3 (cadr exp) env)
                    (eval-3 (caddr exp) env)
                    (eval-3 (cadddr exp) env))
                (error "if syntax error : " exp)))
         ((eq? (car exp) 'define)
            (if (and (= 3 (length exp)) (symbol? (cadr exp)))
                (begin
```

```
                    (set! globenv (cons (cons (cadr exp)
                           (eval-3 (caddr exp) env)) globenv))
                    (cadr exp))
                (error "define syntax error : " exp)))
          ((eq? (car exp) 'set!)
              (if (and (= 3 (length exp)) (symbol? (cadr exp)))
                  (let ((s (cadr exp)) (v (eval-3 (caddr exp) env)))
                        (let ((p (assv s env)))
                             (if (pair? p)
                                 (set-cdr! p v)
                                 (let ((p (assv s globenv)))
                                     (if (pair? p)
                                         (set-cdr! p v)
                                         (error "undefined variable : "
                                                s)))))
                        v)
                  (error "set! syntax error : " exp)))
          (else (apply
                    (eval-3 (car exp) env)
                    (map (lambda (x) (eval-3 x env)) (cdr exp))))))
    (else
       (error "Expression non interpretable : " exp))))
```

Cette formulation de « `define` » et « `set !` » fait appel à une recherche du symbole, soit dans l'environnement local (variable `env`, paramètre de l'évaluateur), soit dans l'environnement global, représenté ici par la variable « `globenv` ». Nous le représentons ainsi :

```
(define globenv '())
(set! globenv (list
   (cons '+ +) (cons '- -) (cons '* *) (cons '< <)
   (cons '= =) (cons '> >) (cons '<= <=) (cons '>= >=)
   (cons 'cons cons) (cons 'car car) (cons 'cdr cdr)
   (cons 'null? null?) (cons 'pair? pair?) (cons 'list? list?)
   (cons 'number? number?) (cons 'symbol? symbol?)
   (cons 'string? string?) (cons 'boolean? boolean?)
   (cons 'quit quit)
 ))
```

Avec cette nouvelle définition, nous pouvons effectuer la session suivante :

```
(load "evaluator-3.scm")
   (scheme-3); to run the evaluator
;=  #t
(scheme-3)
:  (define toto 2)
=> toto
:  toto
=> 2
:  (set! toto (* (+ toto 1) 3))
=> 9
:  toto
=> 9
```

2.5 Modèle 4

La dernière version du modèle doit maintenant s'attaquer à la définition de procédures. Il suffit en fait de traiter les cas des *lambda-expressions*, la forme « `define` » pour des procédures étant simplement du sucre syntaxique enveloppant une lambda expression.

Le choix fait ici est de conserver le corps de la lambda expression sous sa forme arborescente. Exécuter une lambda-expression se ramène tout simplement à évaluer le corps de la lambda-expression dans un environnement

où les variables locales de la lambda-expression ont reçu les valeurs des paramètres. On peut formuler ainsi la chose (en ne représentant ici que la partie nouvelle du code) :

```
((eq? (car exp) 'lambda)
    (list xlambda (cadr exp) (caddr exp) env))
```

Une lambda-expression est donc représentée ici par le mot-clef « `xlambda` », mais – c'est la chose importante à voir – on associe à cette représentation l'environnement courant (la variable `env` de l'évaluateur), qui sera utilisé par la suite pour l'évaluation de la lambda expression. L'autre transformation du code est le remplacement de l'appel à « `apply` » par cette expression :

```
(else (xapply-4
          (eval-4 (car exp) env)
          (map (lambda (x) (eval-4 x env)) (cdr exp))))))
```

La procédure `xapply-4` est elle-même définie ainsi : le cas de `xlambda` est reconnu spécifiquement, et se traduit par l'appel de l'évaluateur, appel dans lequel on introduit les liaisons (variables locales) de la lambda-expression en tête de l'environnement :

```
(define (xapply-4 fct liste)
  (cond
    ((procedure? fct)  (apply fct liste))
    ((and (pair? fct) (eq? xlambda (car fct)))
        (eval-4 (caddr fct)
                (append (map cons (cadr fct) liste) (cadddr fct))))
    (else (error "xapply error : " (cons fct liste)))))
```

On dispose dès lors d'un modèle complet et opérationnel de l'évaluateur, comme le montre la session récapitulative suivante :

```
(load "evaluator-4.scm")
   (scheme-4); to run the evaluator
;=  #t
(scheme-4)
:  (+ 23 31)
=> 54
:  (if #t 2 5)
=> 2
:  'toto
=> toto
:  (define toto 5)
=> toto
:  toto
=> 5
:  (set! toto 8)
=> 8
:  toto
=> 8
:  (define titi (lambda (x) (+ x 2)))
=> titi
:  (titi toto)
=> 10
:  (define fact (lambda (x)
      (if (<= x 0) 1 (* x (fact (- x 1))))))
=> fact
:  (fact 8)
=> 40320
:  (define adder (lambda (x)
      (lambda (y) (+ x y))))
=> adder
:  (define add7 (adder 7))
=> add7
:  (add7 2)
=> 9
```

2.6 Conclusion

Le développement par étapes et raffinements successifs d'un modèle de l'évaluateur permet de comprendre la simplicité des concepts mis en oeuvre. Le travail décrit ici est suffisant pour l'écriture d'une première version de l'interprète dans un langage compilé, version qui peut ensuite être complété par l'ajout de nouvelles primitives, par l'optimisation des structures de données utilisées pour la représentation des objets du système, etc. Le modèle présenté fait naturellement l'impasse sur bien des aspects du langage, comme la gestion de la mémoire, le traitement des continuations, etc. Il est cependant suffisant comme point de départ pour bien appréhender les aspects de *scheme*.

Bibliographie

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure et Interprétation des Programmes Informatiques*. InterEditions, Paris, France, 1989.

[3] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. Structure and Interpretation of Computer Programs. `http://mitpress.mit.edu/sicp/`, 2001.

[4] Gull Agha. *Actors : A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.

[5] Gull Agha and Carl Hewitt. Actors : a conceptual Foundation for Concurrent Object-Oriented Programming. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[6] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North Holland, Amsterdam, 1984.

[7] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, N.J., 1941.

[8] R.Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, INC., Englewood Cliffs, New Jersey, 1987.

[9] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, 1989.

[10] Jean-Jacques Girardot. *Les Langages et Les Systèmes Lisp*. EDITest, 1985.

[11] Jean-Jacques Girardot. *gLisp* : Manuel de Référence. Technical report, Ecole des Mines, Saint-Etienne, Octobre 1990.

[12] M. Gondran. Introduction aux systèmes experts. *Bulletin de la Direction des Etudes et Recherches d'E.D.F., Série C, Mathématiques Informatiques*, (2), 1983.

[13] Chris Hanson. MIT Scheme Reference Manual. Technical report, Massachusetts Institute of Technology, Boston, Mass., 1991.

[14] Chris Hanson. MIT Scheme User's Manual. Technical report, Massachusetts Institute of Technology, Boston, Mass., 1991.

[15] David H.Bartley, John C.Jensen, and Donald W.Oxley. *PC Scheme User's Guide and Language Reference Manual*. the MIT Press, Cambridge, Massachussets, 1988.

[16] J. W. Backus, F. L. Bauer, J. McCarthy, P. Naur, A. J. Perlis, and others... Modifier Report on the Algorithmic Language Algol 60. Technical report, 1975.

[17] Jean-Jacques Girardot. Initiation à la programmation fonctionnelle par Scheme. `http://kiwi.emse.fr/Misc/bookintro.ps.gz`, Septembre 1995.

[18] Jean-Jacques Girardot. Misc User's Manual. Technical report, École des Mines, Saint-Étienne, France, 2001.

[19] Jean-Jacques Girardot. Sch User's Manual. Technical report, École des Mines, Saint-Étienne, France, 2003.

[20] J-L. Krivine. *Lambda calcul : types et modèles*. Masson, 1990.

[21] John Kunz editor. *Common Lisp, The Reference*. Addison Wesley, Reading, MA, 1988.

[22] Henry Lieberman. Reversible Object-Oriented Interpreters. In *ECCOP'87 proceedings*, volume 276, pages 13–21, Paris, France, June 1987. Spinger Verlag.

[23] John McCarthy. Lisp 1.5 Programmer's Manual. Technical report, M.I.T., Cambridge, Mass., 1962.

[24] R4RS. Revised ^{4} Report on the Algorithmic Language Scheme. *ACM Sigplan Notices*, 21(12), Décembre 1990.

[25] R5RS. Revised ^{5} Report on the Algorithmic Language Scheme. Technical report, 1998.

[26] John D. Ramsdell. An Operational Semantics for Scheme. *Lisp Pointers*, V(2), Avril 1992.

[27] Régis Girard. Programmation Fonctionnelle en Scheme. `http://www.univ-reunion.fr/~rgirard/Enseignements/SchemeCours/cours.html`.

[28] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming*. MIT Press, Cambridge, MA, 1992.

[29] Guy L. Steele Jr. *Common Lisp, The language, 2nd edition*. Digital Press, Digital Equipment Corporation, 1990.

[30] Warren Teitlman. Interlisp Reference Manual. Technical report, Xerox, Palo Alto, California, 1981.

Index

Table des matières