Revision^4 du Rapport sur le Langage Algorithmique Scheme

William Clinger et Jonathan Rees (*Editeurs*)

H. Abelson	R. K. Dybvig	C. T. Haynes	G. J. Rozas
N. I. Adams iv	D. P. Friedman	E. Kohlbecker	G. L. Steele Jr.
D. H. Bartley	R. Halstead	D. Oxley	G. J. Sussman
G. Brooks	C. Hanson	K. M. Pitman	M. Wand

Dédié à la Mémoire d'ALGOL 60

2 Novembre 1991

SOMMAIRE

Ce rapport est une description qui définit le langage de programmation Scheme. Scheme est un dialecte de Lisp à portée statique et proprement post-récursif, inventé par Guy Lewis Steele Jr. et Gerald Jay Sussman. Il fut conçu pour bénéficier d'une sémantique exceptionnellement claire et simple, ainsi que d'un petit nombre de règles de formation d'expressions. Une grande variété de paradigmes de programmation peut s'exprimer en Scheme, et parmi eux les styles impératif, fonctionnel, et à passation de messages.

L'introduction offre une brève histoire du langage et de ce rapport.

Les trois premiers chapitres présentent les idées fondamentales du langage et décrivent les notations conventionnelles utilisées pour décrire le langage et pour écrire des programmes dans ce langage.

Les chapitres 4 et 5 décrivent la syntaxe et la sémantique des expressions, programmes, et définitions.

Le chapitre 6 décrit les procédures primitives de Scheme, concernant les manipulations de données et les entrées-sorties.

Le chapitre 7 fournit une syntaxe formelle de Scheme écrite sous une forme BNF étendue, ainsi qu'une sémantique dénotationnelle formelle. Un exemple d'utilisation du langage suit les syntaxe et sémantique formelles.

L'appendice décrivant un système de macros permettant d'étendre la syntaxe de Scheme a été omis dans cette traduction.

Le rapport se termine par une bibliographie et un index alphabétique.

CONTENU

INTRODUCTION

Les langages de programmation devraient être conçus non pas en empilant possibilité sur possibilité, mais en supprimant les faiblesses et les restrictions qui font que des capacités additionnelles s'avèrent nécessaires. Scheme démontre qu'un très petit nombre de règles de formation des expressions, sans aucune restriction sur la manière dont elles sont composées, suffisent à engendrer un langage de programmation pratique et efficace, suffisamment flexible pour supporter la plupart des paradigmes de programmation en usage aujourd'hui.

Scheme fut l'un des premiers langages de programmation à incorporer des procédures de première classe comme dans le lambda-calcul, prouvant par là même l'utilité des règles de portée statique et des structures de bloc dans un langage dynamiquement typé. Scheme fut le premier dialecte de Lisp à distinguer les procédures des lambda-expressions et des symboles, à utiliser un unique environnement lexical pour toutes les variables, et à évaluer l'opérateur tout autant que ses opérandes. En s'appuyant entièrement sur les appels de procédures pour exprimer l'itération, Scheme mit en exergue le fait que les appels de procédures en position terminale sont essentiellement des goto avec passage d'arguments. Scheme fut le premier langage de large diffusion à fournir des procédures d'échappement de première classe, à partir desquelles toutes les structures de contrôle séquentielles connues pouvaient être déduites. Plus récemment, s'inspirant de l'arithmétique générique de Common Lisp, Scheme introduisit les concepts de nombres exacts et inexacts. Avec l'appendice de ce rapport, Scheme est devenu le premier langage de programmation offrant des macros hygiéniques, qui permettent d'étendre de manière fiable la syntaxe d'un langage à structure de blocs.

Historique

La première description de Scheme fut rédigée en 1975 [90]. Un rapport révisé [84] apparut en 1978, qui décrivait l'évolution du langage au moment où son implémentation au MIT était mise à jour pour incorporer un compilateur innovant [79]. Trois projets distincts commencèrent en 1981 et 1982 à utiliser des variantes de Scheme pour des cours au MIT et aux Universités de Yale et d'Indiana [65, 56, 34]. Un livre d'introduction à la programmation utilisant Scheme fut publié en 1984 [2].

Au fur et à mesure que l'audience de Scheme s'élargissait, des dialectes locaux commencèrent à diverger, jusqu'au moment où les étudiants et les chercheurs eurent des problèmes occasionnels en essayant de comprendre du code écrit dans d'autres sites. Quinze représentants des implémentations principales de Scheme se réunirent ainsi en Octobre 1984 pour travailler en direction d'un standard pour Scheme qui soit meilleur et plus largement accepté. Leur rapport [8] fut publié au MIT et à l'Université d'Indiana dans le courant de l'été 1985. Une autre révision prit place au printemps 1986 [67]. Le présent rapport reflète des révisions ultérieures qui firent l'objet d'un accord lors d'une rencontre précédant la "Conference on Lisp and Functional Programming" de l'ACM en 1988, ainsi qu'à travers des échanges de courriers électroniques.

Nous souhaitons que ce rapport appartienne à la communauté Scheme tout entière, et nous donnons à cet effet l'autorisation de le copier tout ou partie sans aucun droit. En particulier, nous encourageons les implémenteurs de Scheme à utiliser ce rapport comme point de départ pour des manuels et autres documents, en le modifiant si nécessaire.

Remerciements

Nous aimerions remercier les personnes suivantes pour leur aide: Alan Bawden, Michael Blair, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Andy Freeman, Richard Gabriel, Yekta Gürsel, Ken Haase, Robert Hieb, Paul Hudak, Richard Kelsey, Morry Katz, Chris Lindblad, Mark Meyer, Jim Miller, Jim Philbin, John Ramsdell, Mike Shaff, Jonathan Shapiro, Julie Sussman, Perry Wagle, Daniel Weise, Henry Wu, et Ozan Yigit. Nous remercions Carol Fessenden, Daniel Friedman, et Christopher Haynes pour leur autorisation d'utiliser des textes issus du manuel de référence du Scheme 311 version 4. Nous remercions Texas Instruments, Inc. pour la permission d'utiliser des textes issus de leur *TI Scheme Language Reference Manual.* Nous reconnaissons avec fierté l'influence des manuels de MIT Scheme, T, Scheme 84, Common Lisp, et Algol 60.

Nous remercions aussi Betty Dexter pour le gros effort qu'elle déploya dans la mise sous forme TEX de ce rapport, et Donald Knuth pour avoir créé le programme qui lui causa tant de peines.

Le laboratoire d'Intelligence Artificielle du Massachusetts Institute of Technology, le Département d'Informatique de l'Université d'Indiana, et le Département des Sciences de l'Information de l'Université d'Oregon supportèrent la préparation de ce rapport. Le support pour le travail au MIT provint en partie de l'ARPA (Advanced Research Projects Agency) du DoD (Department of Defense) sous le contrat N00014-80-C-0505 (Office of Naval Research). Le support pour le travail à l'Université d'Indiana vient de fonds de la National Science Foundation NCS 83-04567 et NCS 83-03325.

Sur la traduction en français

Vous pouvez faire parvenir vos remarques concernant cette traduction (qui ne contient pas le chapitre sur les macros) à *roy@unice.fr*.

DESCRIPTION DU LANGAGE

1. Panorama de Scheme

1.1. Sémantique

Cette section offre un panorama de la sémantique de Scheme. Une sémantique informelle détaillée est le sujet des chapitres 3 à 6. Pour plus de références, la section 7.2 propose une sémantique formelle de Scheme.

Successeur d'Algol, Scheme est un langage de programmation à portée statique. Chaque utilisation d'une variable est associée à une liaison lexicalement visible de cette variable.

Scheme est muni de types latents et non de types manifestes. Les types sont associés à des valeurs (nommées aussi objets) plutôt qu'à des variables. (Certains auteurs disent que les langages à types latents sont faiblement typés ou dynamiquement typés). APL, Snobol et d'autres dialectes de Lisp sont aussi des langages à types latents. Parmi les langages à types manifestes (dits aussi fortement typés ou à typage statique) on trouve Algol 60, Pascal et C.

Tous les objets créés au cours d'un calcul Scheme, y-compris les procédures et les continuations, ont une durée de vie illimitée. Aucun objet Scheme n'est jamais détruit. La raison pour laquelle les implémentations de Scheme ne saturent pas (en général !) la mémoire est qu'elles ont le droit de récupérer la place mémoire occupée par un objet si elles peuvent prouver que cet objet n'interviendra plus dans un calcul ultérieur. Parmi les autres langages dans lesquels la plupart des objets ont une durée de vie illimitée, on trouve APL et d'autres dialectes de Lisp.

Les implémentations de Scheme sont tenues d'être proprement post-récursives. Ceci permet l'exécution d'un calcul itératif dans un espace de pile constant, même si le calcul itératif est décrit par une procédure syntaxiquement récursive. Ainsi avec une implémentation post-récursive, l'itération peut être exprimée avec l'appel de procédure usuel, de sorte que les constructions spéciales pour l'itération ne sont utiles que comme sucre syntaxique.

Les procédures Scheme sont des objets à part entière. Elles peuvent être créées dynamiquement, stockées dans des structures de données, retournées comme valeurs de fonctions, etc. Parmi les autres langages offrant cette possibilité, on trouve Common Lisp et ML.

Un trait intéressant de Scheme est que les continuations, qui dans presque tous les autres langages opèrent derrière la scène, ont aussi un statut "de première classe". Les continuations sont utiles pour implémenter une grande variété de structures de contrôle avancées, comme les sorties non locales, les mécanismes de retour-arrière, ou les coroutines. Voir la section 6.9.

Les arguments des procédures Scheme sont toujours passés par valeur, ce qui signifie que les expressions en argument sont évaluées avant que la procédure ne prenne le contrôle, que cette dernière se serve des résultats des évaluations ou pas. Parmi les autres langages qui fonctionnent en appel par valeur, on trouve ML, C et APL. Ceci est distinct des sémantiques d'évaluation paresseuse d'Haskell, ou de l'appel par nom d'Algol 60 pour lequel un argument n'est évalué que si sa valeur est effectivement utilisée par la procédure.

L'arithmétique en Scheme est définie de sorte à être aussi indépendante que possible de la manière dont les nombres sont représentés à l'intérieur de l'ordinateur. En Scheme, tout entier est un rationnel, tout rationnel est un réel, et tout réel est un complexe. Donc la distinction entre entiers et réels, si importante dans tellement de langages, n'existe pas en Scheme. A sa place apparaît la distinction entre arithmétique exacte—qui correspond à l'idéal mathématique—et arithmétique inexacte sur les approximations. Comme en Common Lisp, l'arithmétique exacte n'est pas limitée aux entiers.

1.2. Syntaxe

Scheme, comme la plupart des dialectes de Lisp, emploie une notation préfixée complétement parenthésée pour les programmes et les (autres) données ; la grammaire de Scheme engendre un sous-langage du langage utilisé pour les données. Une conséquence importante de cette représentation simple et uniforme est la possibilité pour un programme Scheme de traiter un autre programme Scheme comme une donnée.

La procédure `read` procède à une décomposition lexicale et syntaxique de la donnée qu'elle lit. La procédure `read` considère son entrée comme une donnée (section 7.1.2), pas comme un programme.

La syntaxe formelle de Scheme est décrite dans la section 7.1.

1.3. Notation et terminologie

1.3.1. Traits essentiels et non essentiels

Il est exigé que toute implémentation de Scheme supporte des traits marqués comme étant *essentiels*. Les traits non explicitement marqués essentiels sont non essentiels. Les implémentations sont libres d'omettre les traits non essentiels de Scheme ou d'ajouter des extensions, pourvu que les extensions n'entrent pas en conflit avec le langage décrit ici. En particulier, les implémentations doivent permettre du code portable en fournissant un mode syntaxique qui ne va à l'encontre d'aucune convention lexicale de ce rapport et n'interdise l'usage d'aucun identificateur autre que ceux listés comme mots-clés syntaxiques dans la section 2.1.

1.3.2. Erreurs et comportements non spécifiés

Lorsqu'il parle de situation d'erreur, ce rapport utilise la phrase "une erreur est signalée" pour indiquer que les implémentations doivent détecter et signaler l'erreur. Si de tels mots n'apparaissent pas lors de la discussion d'une erreur, les implémentations ne sont pas tenues à détecter ou reporter l'erreur, bien qu'elles soient encouragées à le faire. Une situation d'erreur que les implémentations ne sont pas tenues à détecter sera nommée simplement "une erreur".

Par exemple, c'est une erreur de passer à une procédure un argument que la procédure n'est pas explicitement capable de traiter, même si de telles erreurs de domaines sont rarement mentionnées dans ce rapport. Les implémentations peuvent étendre le domaine de définition d'une procédure pour inclure de tels arguments.

Ce rapport utilise la phrase "peut signaler une violation de restriction d'implémentation" pour indiquer des circonstances dans lesquelles une implémentation est autorisée à signaler qu'elle est incapable de continuer l'exécution d'un programme à cause d'une certaine restriction imposée par l'implémentation. Les restrictions d'implémentation sont bien entendu à éviter si possible, mais les implémentations sont encouragées à signaler les violations de restrictions d'implémentation.

Par exemple, une implémentation peut signaler une violation de restriction d'implémentation si elle n'a pas assez de mémoire pour exécuter un programme.

Si la valeur d'une expression est dite "non spécifiée", alors l'expression doit évaluer à un certain objet sans signaler d'erreur, mais la valeur dépend de l'implémentation ; ce rapport ne dit pas explicitement quelle valeur retourner.

1.3.3. Format des entrées

Les chapitres 4 et 6 sont organisés en entrées. Chaque entrée décrit un trait du langage ou un groupe de traits reliés ; un trait est ou bien une construction syntaxique ou bien une procédure primitive. Une entrée commence par une en-tête d'une ou plusieurs lignes de la forme

modèle *catégorie* essentielle

si le trait est un trait essentiel, ou simplement

modèle *catégorie*

si le trait n'est pas essentiel.

Si *catégorie* est "syntaxe", l'entrée décrit un type d'expression, et l'en-tête donne la syntaxe du type d'expression. Les composants des expressions sont désignés par des variables syntaxiques, qui sont écrites avec des chevrons, par exemple ⟨expression⟩, ⟨variable⟩. Les variables syntaxiques sont censées dénoter des segments de texte; par exemple ⟨expression⟩ signifie n'importe quelle chaîne de caractères qui est une expression syntaxiquement valide. La notation

$\langle\text{chose}_1\rangle \ldots$

indique zéro ou plus occurrences d'une ⟨chose⟩, et

$\langle\text{chose}_1\rangle\ \langle\text{chose}_2\rangle \ldots$

indique au moins une occurrence d'une ⟨chose⟩.

Si *catégorie* est "procédure", alors l'entrée décrit une procédure, et l'en-tête donne un modèle d'appel de la procédure. Les noms des arguments dans le modèle sont en italiques. Donc l'en-tête

`(vector-ref` *vecteur k*`)` procédure essentielle

indique que la procédure primitive essentielle `vector-ref` prend deux arguments, un vecteur *vecteur* et un entier naturel exact *k* (voir ci-dessous). L'en-tête

`(make-vector` *k*`)` procédure essentielle
`(make-vector` *k obj*`)` procédure

indique que dans toutes les implémentations, la procédure `make-vector` doit être définie pour accepter un argument, et certaines implémentations l'étendront pour lui faire accepter aussi deux arguments.

C'est une erreur pour une opération de recevoir un argument pour lequel elle n'est pas prévue. Brièvement, nous suivons la convention disant que si un nom d'argument est aussi le nom d'un type listé à la section 3.4, alors cet argument doit être du type concerné. Par exemple, l'en-tête de `vector-ref` donnée plus haut dit que le premier argument de `vector-ref` doit être un vecteur. Les conventions de noms suivantes impliquent aussi des restrictions sur les types:

obj	tout objet
liste, $liste_1$, … $liste_j$, …	liste (cf. section 6.3)
z, z_1, … z_j, …	nombre complexe
x, x_1, … x_j, …	nombre réel
y, y_1, … y_j, …	nombre réel
q, q_1, … q_j, …	nombre rationnel
n, n_1, … n_j, …	entier
k, k_1, … k_j, …	entier naturel exact

1.3.4. Exemples d'évaluation

Le symbole "⟹" utilisé dans les exemples de programmes se lit "évalue à". Par exemple,

```
(* 5 8)                    ⟹  40
```

signifie que l'expression `(* 5 8)` évalue à l'objet `40`. Ou, plus précisément: l'expression donnée par la suite de caractères "`(* 5 8)`" évalue, dans l'environnement initial, à un objet dont une représentation externe est "`40`". Voir la section 3.3 pour une discussion sur la représentation externe des objets.

1.3.5. Conventions de noms

Par convention, les procédures dont le résultat est une valeur booléenne ont un nom qui se termine par un "?". De telles procédures sont appellées des prédicats.

Par convention, les noms des procédures qui modifient les valeurs d'emplacements-mémoire déjà alloués (voir section 3.5) sont quant à eux suffixés par un "!". Ce sont les procédures de mutation. Par convention, la valeur retournée par une procédure de mutation est non spécifiée.

Par convention, les procédures qui prennent un objet d'un type donné et retourne un objet analogue d'un autre type contiennent un "->" dans leur nom, comme par exemple `list->vector`.

2. Conventions lexicales

Cette section fournit un exposé informel de certaines conventions lexicales utilisées lors de l'écriture de programmes Scheme. Pour une syntaxe formelle de Scheme, voir la section 7.1.

Les versions minuscules et majuscules d'une lettre ne sont jamais distinguées sauf dans les constantes caractères ou chaînes. Par exemple, `Foo` est le même identificateur que `FOO`, et `#x1AB` est le même nombre que `#X1ab`.

2.1. Identificateurs

La plupart des identificateurs autorisés par les autres langages de programmation sont acceptés par Scheme. Les règles précises pour former des identificateurs varient suivant les implémentations de Scheme, mais dans toutes celles-ci, une suite de lettres, chiffres et "caractères alphabétiques étendus" qui commencent par un caractère qui ne peut pas être le début d'un nombre est un identificateur. De plus, `+`, `-`, et `...` sont des identificateurs. Voici quelques exemples d'identificateurs :

```
lambda                   q
list->vector             soupe
+                        V17a
<=?                      a34kTMNs
le-mot-recursion-a-plusieurs-sens
```

Les caractères alphabétiques étendus peuvent être utilisés à l'intérieur des identificateurs comme s'ils étaient des lettres. Les caractères alphabétiques étendus sont :

```
+ - . * / < = > ! ? : $ % _ & ~ ^
```

Voir la section 7.1.1 pour une syntaxe formelle des identificateurs.

Les identificateurs sont utilisés de la manière suivante dans les programmes Scheme :

- Certain identificateurs sont réservés comme mots-clés syntaxiques : (voir ci-dessous).
- Tout identificateur qui n'est pas un mot-clé syntaxique peut être utilisé en tant que variable (voir la section 3.1).
- Quand un identificateur apparaît comme littéral ou à l'intérieur d'un littéral (voir la section 4.1.2), il est utilisé pour dénoter un *symbole* (voir la section 6.4).

Les identificateurs suivants sont des mots-clés syntaxiques, et ne devraient pas être utilisés en tant que variables :

```
=>        do        or
and       else      quasiquote
begin     if        quote
case      lambda    set!
cond      let       unquote
define    let*      unquote-splicing
delay     letrec
```

Quelques implémentations autorisent tous les identificateurs, y-compris les mots-clés syntaxiques, à être utilisés comme variables. Ceci est une extension compatible avec le langage, mais des ambigüités surgissent dans le langage lorsque la restriction est levée, et la manière de résoudre ces ambigüités dépend de l'implémentation.

2.2. Caractères blancs et commentaires

Les caractères *blancs* sont les espaces et les retours-chariot. (Les implémentations offrent des blancs supplémentaires comme la tabulation ou le changement de page.) Les blancs sont utilisés pour une meilleure lisibilité et permettent de séparer les tokens l'un de l'autre, un token étant une unité lexicale telle qu'un identificateur ou un nombre, mais sont autrement sans importance. Les blancs peuvent intervenir entre deux tokens, mais pas à l'intérieur d'un token. Un blanc peut aussi apparaître à l'intérieur d'une chaîne, mais là il est significatif.

Un point-virgule (;) indique le début d'un commentaire. Le commentaire continue jusqu'à la fin de la ligne contenant le point-virgule. Les commentaires sont invisibles pour Scheme, mais la fin de ligne est visible en tant que caractère blanc. Ceci empêche un commentaire d'apparaître au milieu d'un identificateur ou d'un nombre.

```
;;; La procedure FACT calcule la factorielle
;;; d'un entier naturel.
(define fact
  (lambda (n)
    (if (= n 0)
        1        ;Cas de base: retourne 1
        (* n (fact (- n 1))))))
```

2.3. Autres notations

Pour une description des notations utilisées pour les nombres, voir la section 6.5.

`. + -` Ils sont utilisés dans les nombres, et peuvent apparaître n'importe où dans un indentificateur, sauf comme premier caractère. Un signe plus ou moins isolé est aussi un identificateur. Un point isolé (pas à l'intérieur d'un nombre ou d'un identificateur) est isolé dans la notation des doublets (section 6.3),et pour indiquer un paramètre-reste dans une liste de paramètres formels (section 4.1.4). Une suite isolée de trois points consécutifs est aussi un identificateur.

`( )` Les parenthèses sont utilisées pour grouper et noter les listes. (section 6.3).

`'` Le caractère quote est utilisé pour indiquer des données littérales (section 4.1.2).

`` ` `` Le caractère backquote est utilisé pour indiquer des données presque constantes (section 4.2.6).

`, ,@` Le caractère virgule et la suite virgule-arobasque sont utilisés en conjonction avec la backquote (section 4.2.6).

`"` Le caractère guillemet est utilisé pour délimiter des chaînes (section 6.7).

`\` Backslash est utilisé dans la syntaxe pour les constantes caractères (section 6.6) et comme caractère d'échappement à l'intérieur des chaînes constantes (section 6.7).

`[ ] { }` Les crochets et les accolades sont réservés pour des extensions possibles du langage.

`#` Le signe dièse est utilisé dans plusieurs buts suivant le caractère qui le suit:

`#t #f` Ce sont les constantes boolénnes (section 6.1).

`#\` Il introduit un caractère constant (section 6.6).

`#(` Il introduit un vecteur constant (section 6.8). Les vecteurs constants sont terminés par `)` .

`#e #i #b #o #d #x` Ils sont utilisés dans la notation pour les nombres (section 6.5.4).

3. Concepts de base

3.1. Variables et régions

Un identificateur qui n'est pas un mot-clé syntaxique (voir la section 2.1) peut être utilisé comme variable. Une variable peut nommer un emplacement-mémoire où une valeur peut être stockée. Une telle variable est dite *liée* (*bound* en anglais)à cet emplacement. L'ensemble de toutes les liaisons (*bindings* en anglais) visibles en un certain point d'un programme est l'*environnement* en cours en ce point. La valeur stockée dans l'emplacement auquel une variable est liée est appelée la valeur de la variable. Par abus de langage, on dit quelquefois que la variable nomme la valeur ou est liée à cette valeur. Ceci n'est pas rigoureux, mais le risque de confusion est faible.

Certains types d'expressions sont utilisés pour créer de nouveaux emplacements et pour lier des variables à ces emplacements. La plus fondamentale de ces *constructions liantes* est la lambda expression, car toutes les autres constructions liantes peuvent s'expliquer en termes de lambda expressions. Les autres constructions liantes sont les expressions `let`, `let*`, `letrec`, et `do` (voir les sections 4.1.4, 4.2.2, et 4.2.4).

Comme Algol et Pascal, et contrairement à la plupart des dialectes de Lisp sauf Common Lisp, Scheme est un langage à portée statique et à structure de bloc. A chaque emplacement auquel est lié une variable dans un programme, il correspond une *région* du texte du programme au sein de laquelle la liaison est effective. La région est déterminée par la construction liante particulière qui a établi la liaison. Par exemple, si la liaison a été mise en place par une lambda expression, alors cette région est la lambda expression tout entière. Chaque utilisation (ou affectation) d'une variable fait référence à la liaison la plus intérieure contenant cette utilisation. S'il n'existe aucune liaison de la variable dont la région contient l'utilisation, alors cette dernière fait référence à l'environnement du toplevel (section 6). S'il n'existe aucune liaison pour l'identificateur, il sera dit *non lié* (*unbound* en anglais).

3.2. Vrai et Faux

Toute valeur Scheme peut être utilisée comme valeur booléenne dans un test. Comme expliqué dans la section 6.1, toute valeur compte pour vrai dans un test sauf `#f`.

Ce rapport utilisera le mot "vrai" pour toute valeur qui compte pour vrai, et le mot "faux" pour `#f`.

Note : Dans certaines implémentations de Scheme, la liste vide `()` compte pour faux au lieu de vrai.

3.3. Représentations externes

Un concept important en Scheme (et en Lisp) est celui de *représentation externe* d'un objet sous la forme d'une suite de caractères. Par exemple, une représentation externe de l'entier 28 est la suite de caractères "`28`", et une représentation externe d'une liste contenant les entiers 8 et 13 est la suite de caractères "`(8 13)`".

La représentation externe d'un objet n'est pas forcément unique. L'entier 28 admet aussi comme représentations "`#e28.000`" et "`#x1c`", et la liste du paragraphe précédent admet aussi (entre autres) les représentations "`( 08 13)`" et "`(8 . (13 . ()))`" (voir la section 6.3).

Plusieurs objets ont des représentations externes canoniques, mais certains, comme les procédures, n'en ont pas (bien que chaque implémentation soit libre d'en proposer une).

Une représentation externe peut être écrite dans un programme pour obtenir l'objet correspondant (voir `quote`, section 4.1.2).

Les représentations externes peuvent aussi être utilisées dans les entrées/sorties. La procédure `read` (section 6.10.2) scanne des représentations externes, et la procédure `write` (section 6.10.3) les engendre. Ensemble, elles procurent une facilité élégante et puissante pour les entrées-sorties.

Notez que la suite de caractères "`(+ 2 6)`" n'est *pas* une représentation externe de l'entier 8, même s'il s'agit d'une expression dont la valeur est l'entier 8. Il s'agit en fait d'une représentation d'une liste à trois éléments, dont les éléments sont le symbole `+` et les entiers 2 et 6. La syntaxe de Scheme a la propriété que toute suite de caractères formant une expression est aussi la représentation externe d'un certain objet. Ceci peut prêter à confusion, puisqu'il n'est pas toujours évident en-dehors du contexte de savoir si une certaine suite de caractères a pour but de dénoter une donnée ou un programme, mais il s'agit aussi d'une grande source de puissance, puique ceci facilite l'écriture de certains programmes comme les compilateurs ou les interprètes qui traitent les programmes comme des données (ou vice-versa).

La syntaxe des représentations externes de divers types d'objets accompagne la description des primitives de manipulation de ces objets dans les sections ad-hoc du chapitre 6.

3.4. Disjonction des types

Aucun objet ne satisfait plus d'un parmi les prédicats suivants:

```
boolean?          pair?
symbol?           number?
char?             string?
vector?           procedure?
```

Ces prédicates définissent les types *boolean*, *pair*, *symbol*, *number*, *char* (ou *character*), *string*, *vector*, et *procedure*.

3.5. Modèle de la mémoire

Les variables et objets comme les doublets, les vecteurs et les chaînes dénotent implicitement des emplacements ou des suites d'emplacements. Une chaîne, par exemple, dénote autant d'emplacements qu'il y a de caractères dans la chaîne. (Ces emplacements n'ont pas besoin de correspondre à un mot-mémoire entier.) Une nouvelle valeur peut être déposée dans l'un de ces emplacements avec la procédure `string-set!`, mais la chaîne continue à dénoter les mêmes emplacements qu'avant.

Un objet récupéré dans un emplacement, par une référence de variable ou par une procédure comme `car`, `vector-ref`, ou `string-ref`, est équivalente au sens de `eqv?` (section 6.2) au dernier objet déposé dans cet emplacement avant la récupération.

Chaque emplacement est marqué pour montrer s'il est utilisé. Aucune variable ni objet ne réfère jamais à un emplacement inutilisé. Chaque fois que ce rapport parle de mémoire allouée pour une variable ou un objet, cela signifiera qu'un nombre approprié d'emplacements sont choisis parmi les emplacements inutilisés, et les emplacements choisis sont marqués pour indiquer qu'ils sont dès lors utilisés, avant que la variable ou l'objet ne les dénote.

Dans beaucoup de systèmes, il est souhaitable que les constantes (i.e. les valeurs des expressions littérales) résident en mémoire morte. Pour exprimer ceci, on peut imaginer qu'à chaque objet qui dénote des emplacements on associe un drapeau disant si cet objet est mutable ou non. Les constantes et les chaînes retournées par `symbol->string` sont alors les objets non mutables, tandis que tous les objets créés par les autres procédures listées dans ce rapport sont mutables. C'est une erreur d'essayer de déposer une nouvelle valeur dans un emplacement dénoté par un objet non mutable.

4. Expressions

Une expression Scheme est une construction qui retourne une valeur, par exemple une variable, un littéral, un appel de procédure ou une conditionnelle.

Les expressions sont *primitives* ou *dérivées*. Les expressions primitives comprennent les variables et les appels de procédures. Les expressions dérivées ne sont pas sémantiquement primitives, mais peuvent s'expliquer en termes de constructions primitives (section 7.3). Elles sont redondantes à strictement parler, mais elles tiennent compte des usages courants, et sont là en tant qu'abréviations commodes.

4.1. Les types d'expressions primitives

4.1.1. Références de variables

⟨variable⟩ syntaxe essentielle

Une expression réduite à une variable (section 3.1) est une référence de variable. La valeur de la référence de variable est la valeur contenue dans l'emplacement auquel la variable est liée. Référencer une variable non liée provoque une erreur.

```
(define x 28)
x                         ⟹  28
```

4.1.2. Expressions littérales

`(quote ⟨donnée⟩)` syntaxe essentielle
`'⟨donnée⟩` syntaxe essentielle
⟨constante⟩ syntaxe essentielle

`(quote ⟨donnée⟩)` évalue à ⟨donnée⟩. ⟨Donnée⟩ peut être toute représentation externe d'un objet Scheme (voir la section 3.3). Cette notation permet d'inclure des constantes littérales dans du code Scheme.

```
(quote a)                 ⟹  a
(quote #(a b c))          ⟹  #(a b c)
(quote (+ 1 2))           ⟹  (+ 1 2)
```

`(quote ⟨Donnée⟩)` peut être abréviée en `'⟨donnée⟩`. Les deux notations sont parfaitement équivalentes.

```
'a                        ⟹  a
'#(a b c)                 ⟹  #(a b c)
'()                       ⟹  ()
'(+ 1 2)                  ⟹  (+ 1 2)
'(quote a)                ⟹  (quote a)
''a                       ⟹  (quote a)
```

Les constantes numériques, les constantes chaînes, les constantes caractères et les constantes booléennes évaluent "à elles-mêmes". Il est donc inutile de les quoter.

```
'"abc"                    ⟹  "abc"
"abc"                     ⟹  "abc"
'145932                   ⟹  145932
145932                    ⟹  145932
'#t                       ⟹  #t
#t                        ⟹  #t
```

Conformément à la section 3.5, c'est une erreur d'altérer une constante (i.e. la valeur d'une expression littérale) en utilisant une procédure de mutation comme `set-car!` ou `string-set!`.

4.1.3. Appel de procédure

`(⟨opérateur⟩ ⟨opérande`$_1$`⟩ ...)` syntaxe essentielle

Un appel de procédure s'obtient en mettant entre une paire de parenthèses des expressions pour la procédure à appeler et pour les arguments qui vont lui être passés. Les expressions formant l'opérateur et les opérandes sont évalués (dans un ordre non spécifié) et la procédure-résultat reçoit les arguments-résultats.

```
(+ 3 4)                   ⟹  7
((if #f + *) 3 4)         ⟹  12
```

Un certain nombre de procédures sont disponibles en tant que valeurs de variables dans l'environnement initial ; par exemple, les procédures d'addition et de multiplication des exemples ci-dessus sont les valeurs des variables `+` et `*`. De nouvelles procédures sont créées en évaluant des lambda expressions (cf. section 4.1.4).

Les appels de procédures sont aussi nommés *combinaisons*.

Note : Contrairement à d'autres dialectes de Lisp, l'ordre des évaluations n'est pas spécifié et laissé à la discrétion de chaque implémentation. Il peut changer suivant l'appel de procédure. L'expression formant l'opérateur et les expressions formant les opérandes sont évaluées de la même façon.

Note : Bien que l'ordre d'évaluation soit non spécifié, l'effet de toute évaluation concurrente de l'opérateur et des expressions en opérandes doit être cohérent avec un certain ordre séquentiel d'évaluation. L'ordre d'évaluation peut être choisi différemment pour chaque appel de procédure.

Note : Dans plusieurs dialectes de Lisp, la combinaison vide `()` est une expression légale. En Scheme, une combinaison doit contenir au moins une sous-expression, ainsi `()` n'est pas une expression syntaxiquement valide.

4.1.4. Lambda expressions

`(lambda ⟨formels⟩ ⟨corps⟩)` syntaxe essentielle

Syntaxe: ⟨Formels⟩ doit être une liste de paramètres formels comme décrit ci-dessous, et ⟨corps⟩ doit être une suite d'une ou plusieurs expressions.

Sémantique: Une lambda expression a pour valeur une procédure. Un pointeur vers l'environnement en cours lorsque la lambda expression est évaluée est mémorisé à l'intérieur de la procédure. Par la suite, lorsque la procédure sera invoquée avec des arguments effectifs, l'environnement dans lequel la lambda expression a été évaluée sera étendu en liant les variables de la liste ⟨formels⟩ à de nouveaux emplacements-mémoire, les valeurs des arguments effectifs seront déposées dans ces emplacements, et les expressions formant le ⟨corps⟩ de la lambda expression seront évaluées en séquence dans l'environnement ainsi étendu. Le résultat de la dernière expression évaluée dans le ⟨corps⟩ sera retourné comme résultat de l'appel de procédure.

```
(lambda (x) (+ x x))          ⟹  une procédure
((lambda (x) (+ x x)) 4)      ⟹  8

(define reverse-subtract
  (lambda (x y) (- y x)))
(reverse-subtract 7 10)       ⟹  3

(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6)                      ⟹  10
```

⟨Formels⟩ doit prendre l'une des formes suivantes:

- (⟨variable$_1$⟩ ...): La procédure prend un nombre fixe d'arguments. Lorsque la procédure sera invoquée, les arguments seront stockés dans les liaisons des variables correspondantes.
- ⟨variable⟩: La procédure prend un nombre quelconque d'arguments. Lorsque la procédure sera invoquée, la suite des arguments effectifs sera convertie en une liste nouvellement allouée, et cette liste sera déposée dans la liaison de la ⟨variable⟩.
- (⟨variable$_1$⟩ ... ⟨variable$_{n-1}$⟩ **.** ⟨variable$_n$⟩): Si un point entouré d'espaces précède la dernière variable, alors la valeur stockée dans la liaison de la dernière variable sera une liste nouvellement allouée des arguments effectifs correspondants.

C'est une erreur pour une ⟨variable⟩ d'apparaître plus d'une fois dans la liste ⟨formels⟩.

```
((lambda x x) 3 4 5 6)        ⟹  (3 4 5 6)
((lambda (x y . z) z)
 3 4 5 6)                     ⟹  (5 6)
```

Chaque procédure créée en tant que résultat de l'évaluation d'une lambda expression est munie d'une marque d'emplacement-mémoire, de sorte à ce que `eqv?` et `eq?` fonctionnent sur les procédures (cf. section 6.2).

4.1.5. Conditionnelles

`(if` ⟨test⟩ ⟨sivrai⟩ ⟨sifaux⟩`)` syntaxe essentielle
`(if` ⟨test⟩ ⟨sivrai⟩`)` syntaxe

Syntaxe: ⟨Test⟩, ⟨sivrai⟩, et ⟨sifaux⟩ sont des expressions quelconques.

Sémantique: Une expression `if` est évaluée comme suit: d'abord, ⟨test⟩ est évalué. Si le résultat est une valeur vraie (cf. section 6.1), alors ⟨sivrai⟩ est évaluée et sa valeur est retournée. Sinon, ⟨sifaux⟩ est évaluée et sa valeur est retournée. Dans le cas où ⟨test⟩ produit une valeur fausse et où ⟨sifaux⟩ est omis, le résultat de l'expression conditionnelle est non spécifié.

```
(if (> 3 2) 'yes 'no)         ⟹  yes
(if (> 2 3) 'yes 'no)         ⟹  no
(if (> 3 2)
    (- 3 2)
    (+ 3 2))                  ⟹  1
```

4.1.6. Affectations

`(set!` ⟨variable⟩ ⟨expression⟩`)` syntaxe essentielle

⟨Expression⟩ est évaluée, et la valeur résultat est déposée à l'emplacement auquel ⟨variable⟩ est liée. ⟨Variable⟩ doit être liée soit dans une région entourant l'expression `set!`, soit au toplevel. Le résultat d'une expression `set!` n'est pas spécifié.

```
(define x 2)
(+ x 1)                       ⟹  3
(set! x 4)                    ⟹  non spécifié
(+ x 1)                       ⟹  5
```

4.2. Les types d'expressions dérivées

Dans un but de référence, la section 7.3 donne des règles de réécriture qui traduiront les constructions décrites dans cette section en constructions primitives décrites dans la section précédente.

4.2.1. Conditionnelles

`(cond` ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...`)` syntaxe essentielle

Syntaxe: Chaque ⟨clause⟩ doit être de la forme

(⟨test⟩ ⟨expression⟩ ...)

où ⟨test⟩ est une expression quelconque. La dernière ⟨clause⟩ doit être une "clause else", de la forme

(else ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...).

Sémantique: Une expression `cond` est évaluée en évaluant les expressions ⟨test⟩ des ⟨clause⟩s successives dans l'ordre, jusqu'à ce que l'un d'eux évalue à une valeur vraie (cf. section 6.1). Dès qu'un ⟨test⟩ évalue à une valeur vraie, les ⟨expression⟩s restantes dans sa ⟨clause⟩ sont évaluées en séquence, et le résultat de la dernière ⟨expression⟩ de la ⟨clause⟩ est retourné comme résultat de l'expression `cond` tout entière. Si la ⟨clause⟩ en question ne contient qu'un ⟨test⟩ et pas d'⟨expression⟩, c'est la valeur du ⟨test⟩ qui sera retournée. Si tous les ⟨test⟩s évaluent à faux, alors s'il n'existe pas de clause else, le résultat du `cond` sera non spécifié, et s'il en existe une, ses ⟨expression⟩s seront évaluées en séquence, et le résultat de la dernière sera retourné.

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))        ⟹  greater
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))          ⟹  equal
```

`(case` ⟨clé⟩ ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...`)` syntaxe essentielle

Syntaxe: ⟨Clé⟩ est une expression quelconque. Chaque ⟨clause⟩ doit avoir la forme suivante :

((⟨donnée$_1$⟩ ...) ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...),

dans laquelle chaque ⟨donnée⟩ est la représentation externe d'un certain objet. Toutes les ⟨donnée⟩s doivent être distinctes. La dernière ⟨clause⟩ peut être une "clause else", de la forme :

(else ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...).

Sémantique: Une expression `case` est évaluée comme suit. ⟨Clé⟩ est évaluée et son résultat est comparé à chaque ⟨donnée⟩. Si le résultat de l'évaluation de ⟨clé⟩ est équivalent (au sens de `eqv?`, cf. section 6.2) à ⟨donnée⟩, alors les expressions de la ⟨clause⟩ correspondante sont évaluées de gauche à droite et le résultat de la dernière expression de la ⟨clause⟩ est retourné comme résultat de l'expression `case`. Si le résultat de l'évaluation de ⟨clé⟩ est distinct de chaque ⟨donnée⟩, alors s'il y a une clause else ses expressions sont évaluées et le résultat de la dernière expression est retourné, sinon le résultat de l'expression `case` est non spécifié.

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite)) ⟹  composite
(case (car '(c d))
  ((a) 'a)
  ((b) 'b))                 ⟹  non spécifié
(case (car '(c d))
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else 'consonant))        ⟹  consonant
```

`(and` ⟨test$_1$⟩ `...)` syntaxe essentielle

Les expressions ⟨test⟩ sont évaluées de gauche à droite, et la valeur de la première expression évaluant à une valeur fausse (cf. section 6.1) est retournée. Les expressions restantes ne sont pas évaluées ! Si toutes les expressions évaluent à une valeur vraie, la valeur de la dernière expression calculée est retournée. S'il n'y a aucune expression, le résultat est `#t`.

```
(and (= 2 2) (> 2 1))      ⟹  #t
(and (= 2 2) (< 2 1))      ⟹  #f
(and 1 2 'c '(f g))        ⟹  (f g)
(and)                      ⟹  #t
```

`(or` ⟨test$_1$⟩ `...)` syntaxe essentielle

Les expressions ⟨test⟩ sont évaluées de gauche à droite, et la valeur de la première expression évaluant à une valeur vraie (cf. section 6.1) est retournée. Les expressions restantes ne sont pas évaluées. Si toutes les expressions évaluent à une valeur fausse, la valeur de la dernière expression calculée est retournée. S'il n'y a aucune expression, le résultat est `#f`.

```
(or (= 2 2) (> 2 1))       ⟹  #t
(or (= 2 2) (< 2 1))       ⟹  #t
(or #f #f #f)              ⟹  #f
(or (memq 'b '(a b c))
    (/ 3 0))               ⟹  (b c)
```

4.2.2. Constructions liantes

Les trois constructions liantes `let`, `let*` et `letrec` munissent Scheme de structures de blocs, comme Algol 60. La syntaxe de ces trois constructions est identique, mais elles diffèrent quant aux régions qu'elles établissent pour leurs liaisons de variables. Dans une expression `let`, les valeurs initiales sont évaluées avant qu'aucune des variables ne devienne liée ; dans un `let*`, les liaisons et les évaluations sont faites en séquence ; tandis que dans un `letrec`, toutes les liaisons ont été mises en place lorsque les valeurs initiales sont calculées, permettant ainsi des définitions mutuellement récursives.

`(let` ⟨liaisons⟩ ⟨corps⟩`)` syntaxe essentielle

Syntaxe: ⟨Liaisons⟩ est une liste de la forme

`((`⟨variable$_1$⟩ ⟨init$_1$⟩`) ...)`,

où chaque ⟨init⟩ est une expression, et ⟨corps⟩ doit être une suite d'une ou plusieurs expressions. C'est une erreur pour une ⟨variable⟩ d'apparaître plus d'une fois dans la liste des variables devant être liées.

Sémantique: Les ⟨init⟩s sont évaluées dans l'environnement courant (dans un ordre non spécifié), les ⟨variable⟩s sont liées à de nouveaux emplacements contenant les résultats, le ⟨corps⟩ est évalué dans l'environnement étendu, et la valeur de la dernière expression de ⟨corps⟩ est retournée. Chaque liaison d'une ⟨variable⟩ a comme région ⟨corps⟩.

```
(let ((x 2) (y 3))
  (* x y))                 ⟹  6

(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))              ⟹  35
```

Voir aussi le `let` nommé, section 4.2.4.

`(let*` ⟨liaisons⟩ ⟨corps⟩`)` syntaxe

Syntaxe: La même que celle de `let`, sauf qu'une même variable peut apparaître plusieurs fois.

Sémantique: `Let*` est similaire à `let`, mais les liaisons sont effectuées séquentiellement de la gauche vers la droite, et la région d'une liaison indiquée par `(`⟨variable⟩ ⟨init⟩`)` est la partie du `let*` à droite de la liaison. Donc la seconde liaison est effectuée dans un environnement dans lequel la première liaison est visible, et ainsi de suite.

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))              ⟹  70
```

`(letrec` ⟨liaisons⟩ ⟨corps⟩`)` syntaxe essentielle

Syntaxe: La même que celle de `let`.

Sémantique: Les ⟨variable⟩s sont liées à de nouveaux emplacements contenant des valeurs indéfinies, les ⟨init⟩s sont évaluées dans l'environnement résultant (dans un ordre non spécifié), chaque ⟨variable⟩ reçoit le résultat de l'⟨init⟩ correspondante, le ⟨corps⟩ est évalué dans l'environnement résultant, et la valeur de la dernière expression du ⟨corps⟩ est retournée. Chaque liaison d'une ⟨variable⟩ a comme région l'expression `letrec` tout entière, permettant ainsi de définir des procédures mutuellement récursives.

```
(letrec ((even?
          (lambda (n)
            (if (zero? n)
                #t
                (odd? (- n 1)))))
         (odd?
          (lambda (n)
            (if (zero? n)
                #f
                (even? (- n 1))))))
  (even? 88))
                              ⟹  #t
```

Une restriction importante pour `letrec` : il doit être possible d'évaluer chaque ⟨init⟩ sans faire référence à la valeur d'une ⟨variable⟩. Si cette restriction est violée, c'est une erreur. Dans les usages courants de `letrec`, tous les ⟨init⟩s sont des lambda expressions, et cette restriction se trouve alors respectée.

4.2.3. Séquencement

`(begin` ⟨expression$_1$⟩ ⟨expression$_2$⟩ `...)` syntaxe essentielle

Les ⟨expression⟩s sont évaluées séquentiellement de la gauche vers la droite, et la valeur de la dernière ⟨expression⟩ est retournée. Ce type d'expression est utilisé pour séquencer les effets de bord comme les opérations de lecture et d'écriture.

```
(define x 0)

(begin (set! x 5)
       (+ x 1))              ⟹  6

(begin (display "4 + 1 = ")
       (display (+ 4 1)))    ⟹  non spécifié
           et affiche  4 + 1 = 5
```

Note : [2] utilise le mot-clé `sequence` au lieu de `begin`.

4.2.4. Iteration

`(do ((`⟨variable$_1$⟩ ⟨init$_1$⟩ ⟨pas$_1$⟩`)`
`...)`
`(`⟨test⟩ ⟨expression⟩ `...)`
⟨commande⟩ `...)` syntaxe

`Do` est une construction d'itération. Elle spécifie un ensemble de variables destinées à être liées, comment elles vont être initialisées au départ, et comment elles seront mises à jour à chaque itération. Lorsqu'une condition de terminaison est rencontrée, la boucle termine avec un résultat spécifié.

Une expression `do` est évaluée comme suit : les expressions ⟨init⟩ sont évaluées (dans un ordre non spécifié), les ⟨variable⟩s sont liées à de nouveaux emplacements, les résultats des ⟨init⟩ sont placés dans les liaisons des ⟨variable⟩s, et la phase d'itération proprement dite débute.

Chaque itération commence par évaluer le ⟨test⟩. Si le résultat est faux (cf. section 6.1), alors les expressions ⟨commande⟩ sont évaluées dans l'ordre pour leurs effets, les expressions ⟨pas⟩ sont évaluées dans un ordre non spécifié, les ⟨variable⟩s sont liées à de nouveaux emplacements, les résultats des ⟨pas⟩ sont placés dans les liaisons des ⟨variable⟩s, et la phase suivante d'itération débute.

Si ⟨test⟩ évalue à une valeur vraie, alors les ⟨expression⟩s sont évaluées de gauche à droite et la valeur de la dernière ⟨expression⟩ est retournée comme valeur de l'expression `do`. S'il n'y a aucune ⟨expression⟩, le résultat de l'expression `do` est non spécifié.

La région de la liaison d'une ⟨variable⟩ est l'expression `do` tout entière à l'exception des ⟨init⟩s. C'est une erreur pour une ⟨variable⟩ d'apparaître plus d'une fois dans la liste des variables d'un `do`.

Un ⟨pas⟩ peut être omis, auquel cas l'effet est le même que si `(`⟨variable⟩ ⟨init⟩ ⟨variable⟩`)` avait été écrit à la place de `(`⟨variable⟩ ⟨init⟩`)`.

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i i))    ⟹  #(0 1 2 3 4)

(let ((L '(1 3 5 7 9)))
  (do ((L L (cdr L))
       (somme 0 (+ somme (car L))))
      ((null? L) somme)))   ⟹  25
```

`(let` ⟨variable⟩ ⟨liaisons⟩ ⟨corps⟩`)` syntaxe

Certaines implémentations de Scheme offrent une variante de `let`, le "`let` nommé", qui procure une boucle plus générale que `do`, et peut aussi s'utiliser pour exprimer des récursions.

Le `let` nommé a la même syntaxe et sémantique que le `let` usuel, sauf que ⟨variable⟩ est liée à l'intérieur de ⟨corps⟩ à une procédure dont les paramètres sont les variables liées et dont le corps est ⟨corps⟩. Donc l'exécution de ⟨corps⟩ peut être répétée en invoquant la procédure nommée par ⟨variable⟩.

```
(let loop ((L '(3 -2 1 6 -5))
           (nonneg '())
           (neg '()))
  (cond ((null? L) (list nonneg neg))
        ((>= (car L) 0)
         (loop (cdr L)
               (cons (car L) nonneg)
               neg))
        ((< (car L) 0)
         (loop (cdr L)
               nonneg
               (cons (car L) neg)))))
          ⟹  ((6 1 3) (-5 -2))
```

4.2.5. Evaluation retardée

`(delay` ⟨expression⟩`)` syntaxe

La construction `delay` est utilisée avec la procédure `force` pour implémenter l'*évaluation retardée* ou *appel par nécessité*. `(delay` ⟨expression⟩`)` retourne un objet appelé *promesse* qui pourra être plus tard invoqué (par la procédure `force`) pour évaluer ⟨expression⟩ et en fournir la valeur.

Voir la description de `force` (section 6.9) pour une description plus complète de `delay`.

4.2.6. La Quasiquote

`(quasiquote` ⟨modèle⟩`)` syntaxe essentielle
`` ` ``⟨modèle⟩ syntaxe essentielle

Les expressions "quasiquote" (ou "backquote") sont utiles pour construire une liste ou un vecteur dont presque tout le contenu est connu à l'avance. Si aucune virgule n'apparaît dans le ⟨modèle⟩, la quasiquote est équivalente à la quote. Si une virgule apparaît à l'intérieur du ⟨modèle⟩, l'expression qui suit la virgule est évaluée ("unquotée") et sa valeur est insérée dans la structure à la place de la virgule et de l'expression. Si une virgule apparaît suivie immédiatement d'un arobasque (,@), alors l'expression qui suit doit avoir pour valeur une liste ; les parenthèses ouvrante et fermante de cette liste sont alors "gommées" et les éléments de la liste sont insérés à la place de la suite virgule arobasque expression.

```
`(list ,(+ 1 2) 4)          ⟹  (list 3 4)
(let ((nom 'a)) `(list ,nom ',nom))
          ⟹  (list a (quote a))
`(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)
          ⟹  (a 3 4 5 6 b)
`((foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
          ⟹  ((foo 7) . cons)
`#(10 5 ,(sqrt 4) ,@(map sqrt '(16 9)) 8)
          ⟹  #(10 5 2 4 3 8)
```

Les formes quasiquote peuvent être imbriquées. Les substitutions sont faites seulement pour les composants non quotés apparaissant au même niveau d'imbrication que la quasiquote la plus externe. Le niveau d'imbrication augmente d'une unité à l'intérieur de chaque quasiquotation successive, et décroît d'une unité à l'intérieur de chaque "unquotation".

```
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
          ⟹  (a `(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((nom1 'x)
      (nom2 'y))
  `(a `(b ,,nom1 ,',nom2 d) e))
          ⟹  (a `(b ,x ,'y d) e)
```

Les notations `` ` ``⟨modèle⟩ et `(quasiquote` ⟨modèle⟩`)` sont identiques. La forme `,`⟨expression⟩ est identique à `(unquote` ⟨expression⟩`)` tandis que `,@`⟨expression⟩ est identique à `(unquote-splicing` ⟨expression⟩`)`. La syntaxe externe engendrée par `write` pour les listes à deux éléments dont le car est l'un de ces symboles peut varier suivant les implémentations.

```
(quasiquote (list (unquote (+ 1 2)) 4))
          ⟹  (list 3 4)
'(quasiquote (list (unquote (+ 1 2)) 4))
          ⟹  `(list ,(+ 1 2) 4)
     i.e., (quasiquote (list (unquote (+ 1 2)) 4))
```

Attention cependant : un comportement imprévisible peut résulter de l'utilisation des symboles `quasiquote`, `unquote` ou `unquote-splicing` autrement qu'à l'intérieur d'un ⟨modèle⟩ décrit ci-dessus.

5. Structure d'un programme

5.1. Programmes

Un programme Scheme consiste en une suite d'expressions et de définitions. Les expressions sont décrites dans le chapitre 4 ; le reste du chapitre en cours traite des définitions.

Les programmes sont typiquement stockés dans des fichiers ou entrés interactivement lors d'une session Scheme, bien que d'autres paradigmes soient possibles : les questions d'interface utilisateur sortent du cadre de ce rapport. (En réalité, Scheme serait utile comme notation pour exprimer des stratégies calculatoires même en l'absence d'une implémentation effective).

Les définitions intervenant au niveau supérieur (toplevel) d'un programme peuvent être interprétées de manière déclarative. Elle provoquent des créations de liaisons dans l'environnement du toplevel. Les expressions intervenant au niveau supérieur d'un programme sont interprétées interactivement ; elles sont exécutées dans l'ordre lorsque le programme est lancé ou chargé, et réalisent typiquement des initialisations.

5.2. Définitions

Les définitions sont valides dans certains contextes (mais pas tous) au sein desquels les expressions sont acceptées. Elles sont valides seulement au niveau supérieur d'un ⟨programme⟩ et, dans certaines implémentations, au début d'un ⟨corps⟩.

Une définition doit avoir l'une des formes suivantes :

- `(define` ⟨variable⟩ ⟨expression⟩`)`

  Cette syntaxe est essentielle.

- `(define (`⟨variable⟩ ⟨formels⟩`)` ⟨corps⟩`)`

  Cette syntaxe n'est pas essentielle. ⟨Formels⟩ doit être une suite d'aucune ou plusieurs variables, ou une suite d'une ou plusieurs variables suivie d'un point séparé par des espaces et d'une autre variable (comme dans une lambda-expression). Cette forme équivaut à :

  ```
  (define ⟨variable⟩
    (lambda (⟨formels⟩) ⟨corps⟩)).
  ```

- `(define (`⟨variable⟩ `.` ⟨formel⟩`)` ⟨corps⟩`)`

  Cette syntaxe n'est pas essentielle. ⟨Formel⟩ doit être réduit à une seule variable. Cette forme équivaut à :

  ```
  (define ⟨variable⟩
    (lambda ⟨formel⟩ ⟨corps⟩)).
  ```

- `(begin` ⟨définition$_1$⟩ ...`)`

  Cette syntaxe est essentielle. Cette forme est équivalente à l'ensemble des définitions formant le corps de `begin`.

5.2.1. Définitions au toplevel

Au toplevel d'un programme, une définition

`(define` ⟨variable⟩ ⟨expression⟩`)`

a essentiellement le même effet que l'affectation

`(set!` ⟨variable⟩ ⟨expression⟩`)`

si ⟨variable⟩ est liée. Dans le cas contraire cependant, la définition va lier ⟨variable⟩ à un nouvel emplacement avant de réaliser l'affectation, alors que ce serait une erreur de modifier par `set!` une variable non liée.

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3)                    ⟹  6
(define first car)
(first '(1 2))              ⟹  1
```

Toutes les implémentations de Scheme doivent accepter des définitions au toplevel.

Quelques implémentations de Scheme utilisent un environnement initial dans lequel toutes les variables possibles sont liées à des emplacements, dont la plupart contiennent des valeurs indéfinies. Les définitions au toplevel dans une telle implémentation sont vraiment équivalentes à des affectations.

5.2.2. Définitions internes

Quelques implémentations de Scheme autorisent des définitions au début d'un ⟨corps⟩ (le corps d'une `lambda`, d'un `let`, `let*`, `letrec` ou `define`). De telles définitions sont dites *internes* contrairement aux définitions faites au toplevel et décrites ci-dessus. La ⟨variable⟩ définie par une définition interne est locale au ⟨corps⟩, ce qui signifie qu'elle est liée plutôt qu'affectée, et la région de la liaison est le ⟨corps⟩ tout entier. Par exemple,

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))            ⟹  45
```

Un ⟨corps⟩ contenant des définitions internes peut toujours être converti en une expression `letrec` équivalente. Par exemple, le `let` précédent équivaut à:

```
(let ((x 5))
  (letrec ((foo (lambda (y) (bar x y)))
           (bar (lambda (a b) (+ (* a b) a))))
    (foo (+ x 3))))
```

A l'instar de l'expression `letrec` équivalente, il doit être possible d'évaluer chaque ⟨expression⟩ d'une définition interne d'un ⟨corps⟩ sans affecter ou faire référence à la valeur d'une ⟨variable⟩ en train d'être définie.

6. Procédures standards

Cette section décrit les procédures primitives de Scheme. L'environnement initial (ou "toplevel") démarre avec un certain nombre de variables liées à des emplacements contenant des valeurs utiles, principalement des procédures primitives qui vont manipuler les données. Par exemple, la variable `abs` est liée à (un emplacement contenant initialement) une procédure à un paramètre qui calcule la valeur absolue d'un nombre, et la variable `+` est liée à une procédure qui calcule des sommes.

6.1. Booléens

Les objets booléens standard pour vrai et faux s'écrivent `#t` et `#f`. Ce qui importe vraiment, cependant, sont les objets que les expressions conditionnelles de Scheme (`if`, `cond`, `and`, `or`, `do`) considèrent comme vrais ou faux. La phrase "une valeur vraie" (ou simplement "vrai") signifie n'importe quel objet considéré comme vrai par les expressions conditionnelles (idem pour faux).

De toutes les valeurs Scheme standard, seule `#f` compte pour faux dans les expressions conditionnelles. A part `#f`, toutes les valeurs Scheme standard, y-compris `#t`, les doublets, la liste vide, les symboles, les nombres, les vecteurs, les procédures, comptent pour vrai.

Note : Dans certaines implémentations, la liste vide `()` compte pour faux, contrairement à ce qui vient d'être dit. Néammoins, certains exemples de ce rapport supposent que la liste vide compte pour vrai, comme dans [49].

Note : Les programmeurs habitués à d'autres dialectes de Lisp sont prévenus que Scheme distingue à la fois `#f` et la liste vide du symbole `nil`.

Les constantes booléennes évaluent à elles-mêmes, il n'est donc pas nécessaire de les quoter dans un programme.

```
#t                  ⟹  #t
#f                  ⟹  #f
'#f                 ⟹  #f
```

`(not` *obj*`)` procédure essentielle

`Not` retourne `#t` si *obj* est faux, et retourne `#f` sinon.

```
(not #t)            ⟹  #f
(not 3)             ⟹  #f
(not (list 3))      ⟹  #f
(not #f)            ⟹  #t
(not '())           ⟹  #f
(not (list))        ⟹  #f
(not 'nil)          ⟹  #f
```

`(boolean?` *obj*`)` procédure essentielle

`Boolean?` retourne `#t` si *obj* vaut `#t` ou `#f`, et retourne `#f` sinon.

```
(boolean? #f)       ⟹  #t
(boolean? 0)        ⟹  #f
(boolean? '())      ⟹  #f
```

6.2. Prédicats d'équivalence

Un *prédicat* est une procédure à valeur booléenne (`#t` or `#f`). Un *prédicat d'équivalence* est l'analogue en programmation d'une relation d'équivalence en mathématiques (réflexive, symétrique et transitive). Parmi les prédicats d'équivalence décrits dans cette section, `eq?` est le plus fin, ou le plus discriminant, et `equal?` est le moins fin. `Eqv?` est un peu moins discriminant que `eq?`.

`(eqv?` obj_1 obj_2`)` procédure essentielle

La procédure `eqv?` définit une relation d'équivalence utile sur les objets. Brièvement, elle retourne `#t` si obj_1 et obj_2 peuvent normalement être considérés comme le même objet. Cette relation est légèrement sujette à interprétation, mais la spécification suivante de `eqv?` est valide pour toutes les implémentations de Scheme.

La procédure `eqv?` retourne `#t` si:

- obj_1 et obj_2 sont tous deux `#t` ou tous deux `#f`.
- obj_1 et obj_2 sont des symboles et

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                              ⟹  #t
```

  Note : Ceci suppose que ni obj_1 ni obj_2 n'est un "symbole non interné" comme il y est fait allusion dans la section 6.4. Ce rapport ne prétend pas dicter le comportement de `eqv?` en ce qui concerne les extensions dépendantes de chaque implémentation.

- obj_1 et obj_2 sont des nombres, numériquement égaux (voir `=`, section 6.5), et sont tous les deux exacts ou tous les deux inexacts.
- obj_1 et obj_2 sont des caractères et sont le même caractère au sens de la procédure `char=?` (section 6.6).
- obj_1 et obj_2 sont tous deux la liste vide.
- obj_1 et obj_2 sont deux doublets, vecteurs ou chaînes qui résident au même emplacement de la mémoire (section 3.5).
- obj_1 et obj_2 sont des procédures dont les marques d'emplacement mémoire sont les mêmes (section 4.1.4).

La procédure `eqv?` retourne `#f` si :

- obj_1 et obj_2 ont des types distincts (section 3.4).
- l'un de obj_1 et obj_2 est `#t` mais l'autre est `#f`.
- obj_1 et obj_2 sont des symboles, mais

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                              ⟹  #f
```

- l'un de obj_1 ou obj_2 est un nombre exact et l'autre un nombre inexact.
- obj_1 et obj_2 sont des nombres pour lesquels la procédure `=` retourne `#f`.
- obj_1 et obj_2 sont des caractères pour lesquels la procédure `char=?` retourne `#f`.
- l'un de obj_1 ou obj_2 est la liste vide mais pas l'autre.
- obj_1 et obj_2 sont deux doublets, vecteurs ou chaînes qui résident à des emplacements distincts de la mémoire.
- obj_1 et obj_2 sont des procédures qui se comportent différemment sur certains arguments.

```
(eqv? 'a 'a)                    ⟹  #t
(eqv? 'a 'b)                    ⟹  #f
(eqv? 2 2)                      ⟹  #t
(eqv? '() '())                  ⟹  #t
(eqv? 100000000 100000000)      ⟹  #t
(eqv? (cons 1 2) (cons 1 2))    ⟹  #f
```

```
(eqv? (lambda () 1)
      (lambda () 2))          ⟹  #f
(eqv? #f 'nil)                ⟹  #f
(let ((p (lambda (x) x)))
  (eqv? p p))                 ⟹  #t
```

Les exemples suivants illustrent des cas où les règles précédentes ne spécifient pas complètement le comportement de `eqv?`. Tout ce que l'on peut dire dans de tels cas, c'est que la valeur retournée par `eqv?` est un booléen.

```
(eqv? "" "")                  ⟹  non spécifié
(eqv? '#() '#())              ⟹  non spécifié
(eqv? (lambda (x) x)
      (lambda (x) x))         ⟹  non spécifié
(eqv? (lambda (x) x)
      (lambda (y) y))         ⟹  non spécifié
```

Les exemples suivants montrent l'utilisation de `eqv?` avec des procédures à état local. `Gen-counter` doit retourner chaque fois une procédure distincte, puisque chaque procédure a son propre compteur interne. `Gen-loser` au contraire retourne des procédures équivalentes, puisque l'état local ne modifie pas la valeur ou les effets de bord des procédures.

```
(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-counter)))
  (eqv? g g))                 ⟹  #t
(eqv? (gen-counter) (gen-counter))
                              ⟹  #f
(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-loser)))
  (eqv? g g))                 ⟹  #t
(eqv? (gen-loser) (gen-loser))
                              ⟹  non spécifié

(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
         (g (lambda () (if (eqv? f g) 'both 'g)))
  (eqv? f g))
                              ⟹  non spécifié

(letrec ((f (lambda () (if (eqv? f g) 'f 'both)))
         (g (lambda () (if (eqv? f g) 'g 'both)))
  (eqv? f g))
                              ⟹  #f
```

Puisqu'il s'agit d'une erreur de modifier des objets constants (ceux qui sont retournés par les expressions littérales), les implémentations ont le droit, bien que ce ne soit pas obligatoire, de procéder à des partages de structures lorsque c'est approprié. Donc la valeur de `eqv?` sur les constantes dépend parfois de l'implémentation.

```
(eqv? '(a) '(a))              ⟹  non spécifié
(eqv? "a" "a")                ⟹  non spécifié
(eqv? '(b) (cdr '(a b)))      ⟹  non spécifié
(let ((x '(a)))
  (eqv? x x))                 ⟹  #t
```

Rationale : La définition précédente de `eqv?` relaxe les contraintes d'implémentation en ce qui concerne le traitement des procédures et des littéraux : les implémentations sont libres de détecter ou pas l'équivalence de deux procédures ou de deux littéraux, et de décider si elles partageront les représentations d'objets équivalents en utilisant le même pointeur ou schéma de bits pour les représenter.

(`eq?` *obj*$_1$ *obj*$_2$) procédure essentielle

`Eq?` est similaire à `eqv?` sauf dans certains cas dans lesquels `eq?` est capable de distinctions plus fines que `eqv?`.

On garantit que `Eq?` et `eqv?` auront le même comportement sur les symboles, les booléens, la liste vide, les doublets, et sur les chaînes ou vecteurs non vides. Le comportement de `eq?` sur les nombres et les caractères dépend de l'implémentation, mais il retournera toujours vrai ou faux, et ne retournera vrai que si `eqv?` retournerait vrai lui aussi. `Eq?` peut aussi ne pas se comporter de la même façon que `eqv?` sur les chaînes ou vecteurs vides.

```
(eq? 'a 'a)                   ⟹  #t
(eq? '(a) '(a))               ⟹  non spécifié
(eq? (list 'a) (list 'a))     ⟹  #f
(eq? "a" "a")                 ⟹  non spécifié
(eq? "" "")                   ⟹  non spécifié
(eq? '() '())                 ⟹  #t
(eq? 2 2)                     ⟹  non spécifié
(eq? #\A #\A)                 ⟹  non spécifié
(eq? car car)                 ⟹  #t
(let ((n (+ 2 3)))
  (eq? n n))                  ⟹  non spécifié
(let ((x '(a)))
  (eq? x x))                  ⟹  #t
(let ((x '#()))
  (eq? x x))                  ⟹  #t
(let ((p (lambda (x) x)))
  (eq? p p))                  ⟹  #t
```

Rationale : Il est en général possible d'implémenter `eq?` de manière bien plus efficace que `eqv?`. Par exemple, comme une simple comparaison de pointeurs au lieu d'une opération plus compliquée. L'une des raisons est qu'il peut ne pas être possible de comparer deux nombres avec `eqv?` en temps constant, alors que `eq?` implémenté sous la forme d'une comparaison de pointeurs se fera en temps constant. `Eq?` peut être utilisé comme `eqv?` dans les applications utilisant des procédures pour implémenter des objets à états locaux puisqu'elle suit les mêmes contraintes que `eqv?`.

`(equal? obj1 obj2)` procédure essentielle

`Equal?` compare récursivement les contenus des doublets, des vecteurs et des chaînes, et applique `eqv?` sur les autres objets comme les nombres et les symboles. Disons que les objets sont généralement `equal?` s'ils s'impriment de la même façon. `Equal?` peut ne pas terminer si ses arguments sont des structures circulaires.

```
(equal? 'a 'a)                ==>  #t
(equal? '(a) '(a))            ==>  #t
(equal? '(a (b) c)
        '(a (b) c))           ==>  #t
(equal? "abc" "abc")          ==>  #t
(equal? 2 2)                  ==>  #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))   ==>  #t
(equal? (lambda (x) x)
        (lambda (y) y))       ==>  non spécifié
```

6.3. Doublets et listes

Un *doublet* (ou comme on dit en Lisp une *paire pointée*) est une structure d'article à deux champs nommés car et cdr (pour des raisons historiques). Les doublets sont créés par la procédure `cons`. On accède aux champs `car` et `cdr` avec les procédures `car` et `cdr`. On modifie les champs `car` et `cdr` avec les procédures `set-car!` et `set-cdr!`.

Les doublets sont utilisés d'abord pour représenter les listes. Une liste peut être définie récursivement soit comme la liste vide, soit comme un doublet dont le cdr est une liste. Plus précisément, l'ensemble des listes est défini comme le plus petit ensemble X tel que

- La liste vide est dans X.
- Si L est dans X, alors tout doublet dont le champ cdr contient L est encore dans X.

Les objets dans les champs car des doublets successifs d'une liste sont les éléments de cette liste. Par exemple, une liste à deux éléments est un doublet dont le car est le premier élément et dont le cdr est un doublet dont le car est le second élément et dont le cdr est vide. La longueur d'une liste est le nombre de ses éléments.

La liste vide est un objet spécial ayant un type propre (ce n'est pas un doublet) ; elle n'a aucun élément et sa longueur est 0.

Note : Les définitions précédentes impliquent qu'une liste est de longueur finie et se termine par la liste vide.

La notation la plus générale (représentation externe) pour les doublets Scheme est la notation “pointée” `(`c_1 `.` c_2`)` où c_1 est la valeur du champ car et c_2 la valeur du champ cdr. Par exemple, `(4 . 5)` est la représentation externe d'un doublet dont le car est 4 et dont le cdr est 5. Notez que `(4 . 5)` est bien la représentation externe d'un doublet, et pas d'une expression qui évalue à un doublet.

Une notation plus linéaire peut être utilisée pour les listes: les éléments de la liste sont simplement regroupés entre parenthèses et séparés par des espaces. La liste vide s'écrit `()`. Par exemple,

```
(a b c d e)
```

et

```
(a . (b . (c . (d . (e . ())))))
```

sont deux notations équivalentes pour une liste de symboles.

Un chaînage de doublets ne se terminant pas par la liste vide est appelé *liste impropre*. Remarquez qu'une liste impropre n'est pas une liste. Les notations linéaire et pointée peuvent se combiner pour représenter les listes impropres :

```
(a b c . d)
```

est équivalente à

```
(a . (b . (c . d)))
```

Qu'un doublet donné soit une liste dépend du contenu du champ cdr. Lorsque la procédure `set-cdr!` est utilisée, un objet peut être une liste à un moment donné, mais plus par la suite :

```
(define x (list 'a 'b 'c))
(define y x)
y                  ==>  (a b c)
(list? y)          ==>  #t
(set-cdr! x 4)     ==>  non spécifié
x                  ==>  (a . 4)
(eqv? x y)         ==>  #t
y                  ==>  (a . 4)
(list? y)          ==>  #f
(set-cdr! x x)     ==>  non spécifié
(list? x)          ==>  #f
```

A l'intérieur des expressions littérales et des représentations des objets lus par `read`, les formes `'`⟨donnée⟩, `` ` ``⟨donnée⟩, `,`⟨donnée⟩ et `,@`⟨donnée⟩ dénotent des listes à deux éléments dont le premier élément est respectivement l'un des symboles `quote`, `quasiquote`, `unquote` et `unquote-splicing`. Le second élément est dans chaque cas ⟨donnée⟩. Cette convention est mise en place de sorte que des programmes Scheme arbitraires puissent être représentés par des listes. Ceci signifie, au vu de la grammaire Scheme, que chaque ⟨expression⟩ est aussi une ⟨donnée⟩ (cf. section 7.1.2). Entre autres, ceci autorise l'usage de la procédure `read` pour scanner des programmes Scheme. Voir la section 3.3.

`(pair? `*obj*`)` procédure essentielle

`Pair?` retourne `#t` si *obj* est un doublet, et retourne `#f` sinon.

```
(pair? '(a . b))          ⟹  #t
(pair? '(a b c))          ⟹  #t
(pair? '())               ⟹  #f
(pair? '#(a b))           ⟹  #f
```

`(cons `obj_1 obj_2`)` procédure essentielle

Retourne un doublet nouvellement alloué dont le car est obj_1 et dont le cdr est obj_2. Le doublet est garanti distinct (au sens de `eqv?`) de tout objet existant.

```
(cons 'a '())             ⟹  (a)
(cons '(a) '(b c d))      ⟹  ((a) b c d)
(cons "a" '(b c))         ⟹  ("a" b c)
(cons 'a 3)               ⟹  (a . 3)
(cons '(a b) 'c)          ⟹  ((a b) . c)
```

`(car `*doublet*`)` procédure essentielle

Retourne le contenu du champ car du *doublet*. C'est une erreur de demander le car d'une liste vide.

```
(car '(a b c))            ⟹  a
(car '((a) b c d))        ⟹  (a)
(car '(1 . 2))            ⟹  1
(car '())                 ⟹  erreur
```

`(cdr `*doublet*`)` procédure essentielle

Retourne le contenu du champ cdr du *doublet*. C'est une erreur de demander le cdr d'une liste vide.

```
(cdr '((a) b c d))        ⟹  (b c d)
(cdr '(1 . 2))            ⟹  2
(cdr '())                 ⟹  erreur
```

`(set-car! `*doublet obj*`)` procédure essentielle

Dépose *obj* dans le champ car du *doublet*. La valeur retournée par `set-car!` est non spécifiée.

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)          ⟹  non spécifié
(set-car! (g) 3)          ⟹  erreur
```

`(set-cdr! `*doublet obj*`)` procédure essentielle

Dépose *obj* dans le champ cdr du *doublet*. La valeur retournée par `set-cdr!` est non spécifiée.

`(caar `*doublet*`)` procédure essentielle
`(cadr `*doublet*`)` procédure essentielle
⋮ ⋮
`(cdddar `*doublet*`)` procédure essentielle
`(cddddr `*doublet*`)` procédure essentielle

Ces procédures sont des composées de `car` et `cdr`, où par exemple `caddr` pourrait être défini par

```
(define caddr (lambda (x) (car (cdr (cdr x))))).
```

Les 28 compositions jusqu'au niveau 4 sont fournies comme primitives.

`(null? `*obj*`)` procédure essentielle

Retourne `#t` si *obj* est la liste vide, et `#f` sinon.

`(list? `*obj*`)` procédure essentielle

Retourne `#t` si *obj* est une liste, et `#f` sinon. Par définition, toutes les listes sont de longueur finie et se terminent par la liste vide.

```
(list? '(a b c))      ⟹  #t
(list? '())           ⟹  #t
(list? '(a . b))      ⟹  #f
(let ((x (list 'a)))
  (set-cdr! x x)
  (list? x))          ⟹  #f
```

`(list `*obj* ...`)` procédure essentielle

Retourne une liste nouvellement allouée formée de ses arguments.

```
(list 'a (+ 3 4) 'c)      ⟹  (a 7 c)
(list)                    ⟹  ()
```

`(length `*liste*`)` procédure essentielle

Retourne la longueur de *liste*.

```
(length '(a b c))         ⟹  3
(length '(a (b) (c d e))) ⟹  3
(length '())              ⟹  0
```

`(append `*liste* ...`)` procédure essentielle

Retourne une liste formée des éléments de la première *liste* suivis de ceux des autres *listes*.

```
(append '(x) '(y))        ⟹  (x y)
(append '(a) '(b c d))    ⟹  (a b c d)
(append '(a (b)) '((c)))  ⟹  (a (b) (c))
```

La liste résultat est toujours nouvellement allouée, sauf qu'elle partage la structure de la dernière *liste* argument. En fait, le dernier argument peut être n'importe quel objet; s'il s'agit d'une liste impropre, le résultat sera une liste impropre.

```
(append '(a b) '(c . d))  ⟹  (a b c . d)
(append '() 'a)           ⟹  a
```

`(reverse `*liste*`)` procédure essentielle

Retourne une liste nouvellement allouée formée des éléments de *liste* en ordre inverse.

```
(reverse '(a b c))          ⟹  (c b a)
(reverse '(a (b c) d (e (f))))
          ⟹  ((e (f)) d (b c) a)
```

`(list-tail` *liste k*`)` procédure

Retourne la sous-liste de *liste* obtenue en omettant les k premiers éléments. Une définition possible de `list-tail` pourrait être :

```
(define list-tail
  (lambda (x k)
    (if (zero? k)
        x
        (list-tail (cdr x) (- k 1)))))
```

`(list-ref` *liste k*`)` procédure essentielle

Retourne l'élément numéro k de *liste*. Le car d'une liste est l'élément numéro 0.

```
(list-ref '(a b c d) 2)     ⟹  c
(list-ref '(a b c d)
          (inexact->exact (round 1.8)))
          ⟹  c
```

`(memq` *obj liste*`)` procédure essentielle
`(memv` *obj liste*`)` procédure essentielle
`(member` *obj liste*`)` procédure essentielle

Ces procédures retournent la première sous-liste de *liste* dont le car est *obj*, où les sous-listes de *liste* sont les listes non vides retournées par `(list-tail` *liste k*`)` pour k inférieur ou égal à la longueur de *liste*. Si *obj* n'est pas un élément de *liste*, alors `#f` (et non la liste vide) est retourné. `Memq` utilise `eq?` pour comparer *obj* aux éléments de *liste*, tandis que `memv` utilise `eqv?` et que `member` utilise `equal?`.

```
(memq 'a '(a b c))          ⟹  (a b c)
(memq 'b '(a b c))          ⟹  (b c)
(memq 'a '(b c d))          ⟹  #f
(memq (list 'a) '(b (a) c)) ⟹  #f
(member (list 'a)
        '(b (a) c))         ⟹  ((a) c)
(memq 101 '(100 101 102))   ⟹  non spécifié
(memv 101 '(100 101 102))   ⟹  (101 102)
```

`(assq` *obj aliste*`)` procédure essentielle
`(assv` *obj aliste*`)` procédure essentielle
`(assoc` *obj aliste*`)` procédure essentielle

Aliste (pour "liste d'associations") doit être une liste de doublets. Ces procédures cherchent le premier doublet dans *aliste* dont le car est *obj*, et retournent ce doublet. S'il n'en trouvent pas, le résultat est `#f`. `Assq` utilise `eq?` pour comparer *obj* avec les car des doublets de *aliste*, tandis que `assv` utilise `eqv?` et que `assoc` utilise `equal?`.

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)                 ⟹  (a 1)
(assq 'b e)                 ⟹  (b 2)
(assq 'd e)                 ⟹  #f
(assq (list 'a) '(((a)) ((b)) ((c))))
                            ⟹  #f
(assoc (list 'a) '(((a)) ((b)) ((c))))
                            ⟹  ((a))
(assq 5 '((2 3) (5 7) (11 13)))
                            ⟹  non spécifié
(assv 5 '((2 3) (5 7) (11 13)))
                            ⟹  (5 7)
```

Rationale : Bien qu'ils soient d'ordinaire utilisés en tant que prédicats, `memq`, `memv`, `member`, `assq`, `assv` et `assoc` n'ont pas de point d'interrogation dans leurs noms car ils retournent des valeurs plus utiles que de simples `#t` ou `#f`.

6.4. Symboles

Les symboles sont des objets dont l'utilité repose sur le fait que deux symboles sont identiques (au sens de `eqv?`) si et seulement si leurs noms s'épellent de la même façon. C'est exactement la propriété requise pour représenter les identificateurs dans les programmes, et ainsi la plupart des implémentations de Scheme les utilisent de manière interne dans ce but. Les symboles sont utiles pour bien d'autres applications; par exemple, ils peuvent être utilisés comme les valeurs des types énumérés de Pascal.

Les règles d'écriture des symboles sont les mêmes que pour les identificateurs ; voir les sections 2.1 et 7.1.1.

On garantit que tout symbole qui a été retourné comme partie d'une expression littérale, ou lu par la procédure `read` et écrit par la procédure `write`, sera relu comme un symbole identique (au sens de `eqv?`). La procédure `string->symbol` cependant, peut créer des symboles pour lesquels cette invariance write/read peut être mise en défaut parce que leurs noms contiennent des caractères spéciaux ou des lettres qui ne sont pas dans la casse standard.

Note : Quelques implémentations de Scheme ont une facilité dite "slashification" pour garantir l'invariance write/read pour tous les symboles, mais historiquement l'utilisation la plus importante de cette facilité a été de compenser l'absence de vraies chaînes de caractères.

Certaines implémentations ont aussi des "symboles non internés", qui mettent en défaut l'invariance write/read même dans le cas d'implémentations avec slashification, et engendrent aussi des exceptions à la règle qui veut que deux symboles soient les mêmes si et seulement si leurs noms s'épellent de la même façon.

`(symbol?` *obj*`)` procédure essentielle

Retourne `#t` si *obj* est un symbole et `#f` sinon.

```
(symbol? 'foo)              ⟹  #t
(symbol? (car '(a b)))      ⟹  #t
(symbol? "bar")             ⟹  #f
(symbol? 'nil)              ⟹  #t
(symbol? '())               ⟹  #f
(symbol? #f)                ⟹  #f
```

(symbol->string *symbole***)** procédure essentielle

Retourne le nom de *symbole* en tant que chaîne. Si le symbole fait partie d'un objet retourné comme valeur d'une expression littérale (section 4.1.2) ou par un appel à la procédure `read`, et si son nom contient des caractères alphabétiques, alors la chaîne retournée contiendra des caractères dans la casse préférée par défaut de l'implémentation ; certaines préfèrent les minuscules, d'autres les majuscules. Si le symbole a été retourné par `string->symbol`, la casse des caractères dans la chaîne retournée sera la même que la casse dans la chaîne qui a été passée à `string->symbol`. C'est une erreur d'appliquer des procédures de mutation comme `string-set!` à des chaînes retournées par cette procédure.

Les exemples suivants supposent que la casse par défaut de l'implémentation est le bas de casse (minuscules):

```
(symbol->string 'flying-fish)
                            ⟹  "flying-fish"
(symbol->string 'Martin)    ⟹  "martin"
(symbol->string
   (string->symbol "Vanessa"))
                            ⟹  "Vanessa"
```

(string->symbol *chaîne***)** procédure essentielle

Retourne le symbole dont le nom est *chaîne*. Cette procédure peut créer des symboles dont les noms contiennent des caractères spéciaux ou des lettres dans la casse non standard, mais c'est en général une mauvaise idée de créer de tels symboles, car dans certaines implémentations de Scheme, il ne pourront pas être relus. Voir `symbol->string`.

Les exemples qui suivent supposent que la casse standard de l'implémentation est le bas de casse:

```
(eq? 'mISSISSIppi 'mississippi)
          ⟹  #t
(string->symbol "mISSISSIppi")
          ⟹  the symbol with name "mISSISSIppi"
(eq? 'bitBlt (string->symbol "bitBlt"))
          ⟹  #f
(eq? 'JollyWog
     (string->symbol
       (symbol->string 'JollyWog)))
          ⟹  #t
(string=? "K. Harper"
          (symbol->string
            (string->symbol "K. Harper")))
          ⟹  #t
```

6.5. Nombres

Le calcul nuérique a été traditionnellement négligé par la communauté Lisp. Jusqu'à Common Lisp, il n'y avait pas de stratégie mûrement pensée pour organiser le calcul numérique, et à l'exception du système MacLisp [61], peu d'efforts furent faits pour exécuter du code numérique efficacement. Ce rapport prend acte de l'excellent travail des membres du comité Common Lisp et accepte plusieurs de leurs recommandations. En un certain sens, ce rapport simplifie et généralise leurs propositions d'une manière cohérente avec les buts de Scheme.

Il est important de faire une distinction entre les nombres mathématiques, les nombres Scheme qui essayent de les modéliser, les représentations en machine utilisées pour implémenter les nombres Scheme, et les notations utilisées pour écrire des nombres. Ce rapport utilise les types *number*, *complex*, *real*, *rational*, et *integer* pour parler à la fois des nombres mathématiques et des nombres Scheme. Les termes *fixnum* et *flonum* feront référence aux représentations machine en virgule fixe ou flottante.

6.5.1. Types numériques

Mathématiquement, les nombres peuvent être organisés en une tour de sous-types dans laquelle chaque niveau est un sous-ensemble du niveau au-dessus:

number
complex
real
rational
integer

Par exemple, 3 est un entier. Donc 3 est aussi un rationnel, un réel et un complexe. Il en est de même pour les nombres Scheme qui modélisent 3. Pour les nombres Scheme, ces types sont définis par les prédicats `number?`, `complex?`, `real?`, `rational?` et `integer?`.

Il n'y a pas de relation simple entre le type d'un nombre et sa représentation à l'intérieur d'un ordinateur. Bien que la plupart des implémentations de Scheme offrent au moins deux représentations distinctes de 3, ces différentes représentations dénotent le même entier.

Les opérations numériques de Scheme traitent les nombres comme des données abstraites, de manière aussi indépendante que possible de leurs représentations. Bien qu'une implémentation de Scheme puisse utiliser les représentations en virgule fixe ou flottante (ou autre) pour les nombres, ceci ne doit pas être visible au programmeur occasionnel qui rédige des programmes simples.

Il est nécessaire, cependant, de distinguer les nombres qui sont représentés de manière exacte de ceux qui ne le sont pas. Par exemple, les indices dans les structures de données doivent être connus exactement, de même que les coefficients d'un polynôme dans un système de calcul formel. D'un autre côté, les résultats de mesures sont intrinsèquement inexacts, et les nombres irrationnels peuvent être approchés par des rationnels, donc par des approximations inexactes. Dans le but de débusquer les utilisations de nombres inexacts là où des nombres exacts sont requis, Scheme distingue explicitement les nombres exacts et les nombres inexacts. Cette distinction est orthogonale aux types numériques.

6.5.2. Exactitude

Les nombres Scheme sont *exacts* ou *inexacts*. Un nombre est exact s'il a été écrit sous la forme d'une constante exacte, ou s'il a été obtenu comme résultat d'opérations exactes portant sur des nombres exacts. Un nombre est inexact s'il a été écrit sous la forme d'une constante inexacte, s'il a été obtenu à partir d'ingrédients inexacts, ou s'il provient d'opérations inexactes. Donc l'inexactitude est une propriété contagieuse pour un nombre.

Si deux implémentations produisent des résultats exacts pour un calcul dans lequel n'intervient aucun résultat intermédiaire inexact, les deux résultats seront mathématiquement équivalents. Ceci n'est pas vrai en général pour les calculs qui font intervenir des nombres inexacts à cause de méthodes approchées comme l'arithmétique en virgule flottante, mais il est du ressort de chaque implémentation de rendre le résultat aussi proche que possible du résultat mathématique idéal.

Les opérations rationnelles comme `+` devraient toujours produire des résultats exacts en présence d'arguments exacts. Si l'opération est incapable de produire un résultat exact, alors elle peut soit prévenir d'une violation de restriction d'implémentation, soit silencieusement convertir le résultat en une valeur inexacte. Voir la section 6.5.3.

A l'exception de `inexact->exact`, les opérations décrites dans cette section doivent en général retourner des résultats inexacts en présence d'arguments inexacts. Une opération peut, cependant, retourner un résultat exact si elle peut prouver que la valeur du résultat n'est pas modifiée par l'inexactitude de ses arguments. Par exemple, la multiplication d'un nombre quelconque par un 0 exact doit produire un 0 exact, même si l'autre argument est inexact.

6.5.3. Restriction d'implémentation

Les implémentations de Scheme ne sont pas tenues à implémenter la tour complète des sous-types donnée à la section 6.5.1, mais elles doivent implémenter un sous-ensemble cohérent compatible à la fois avec les buts de l'implémentation et l'esprit du langage Scheme. Par exemple, une implémentation dans laquelle tous les nombres seraient des réels pourrait encore s'avérer tout-à-fait utile.

Les implémentations peuvent aussi proposer seulement un intervalle limité de nombres d'un type donné, suivant les contraintes de cette section. L'intervalle des nombres exacts d'un certain type peut être différent de l'intervalle des nombres inexacts du même type. Par exemple, une implémentation qui utilise les nombres flottants pour représenter tous ses nombres réels inexacts peut supporter un intervalle quasiment non borné d'entiers et de rationnels exacts, tout en limitant le domaine des réels inexacts (et donc le domaine des entiers et rationnels inexacts) au domaine dynamique du format en virgule flottante.

De plus, la distance entre les entiers inexacts et rationnels représentables peut s'agrandir dans une telle implémentation au fur et à mesure que l'on s'approche des limites de l'intervalle.

Une implémentation de Scheme doit supporter les entiers exacts dans le domaine des nombres pouvant être utilisés comme indices de listes, de vecteurs, de chaînes ou pouvant provenir d'un calcul de listes, vecteurs ou chaînes. Les procédures `length`, `vector-length` et `string-length` doivent retourner un entier exact, et c'est une erreur d'utiliser autre chose qu'une entier exact comme indice. En outre, tout entier constant à l'intérieur du domaine des indices, s'il est exprimé avec la syntaxe d'un entier exact, sera vraiment lu comme un entier exact, quelles que soient les restrictions qu'une implémentation pourrait apporter à l'extérieur de ce domaine. Finalement, les procédures ci-dessous retourneront toujours un entier exact à condition que leurs arguments soient des entiers exacts, et le résultat mathématiquement espéré sera représentable comme un entier exact de l'implémentation :

```
+          -            *
quotient   remainder    modulo
max        min          abs
numerator  denominator  gcd
lcm        floor        ceiling
truncate   round        rationalize
expt
```

Les implémentations sont encouragées (mais pas obligées) à supporter les entiers et rationnels exacts d'une taille pratiquement illimitée, et d'implémenter les procédures ci-dessus et la procédure `/` de telle sorte qu'elles retournent toujours des résultats exacts lorsqu'on leur passe des arguments exacts. Si l'une de ces procédures est incapable de fournir un résultat exact avec des arguments exacts, alors elle doit ou bien signaler une restriction d'implémentation ou bien silencieusement convertir son résultat en un nombre inexact. Une telle conversion pourra provoquer une erreur plus tard.

Une implémentation peut utiliser la virgule flottante ainsi que d'autres stratégies de représentation approchée des nombres inexacts. Ce rapport recommande, mais n'impose pas, aux implémentations qui utilise la virgule flottante de suivre les standards correspondants IEEE 32-bits et 64-bits, et que les implémentations qui utiliseraient d'autres représentations fassent au moins aussi bien que ces standards [48].

En particulier, les implémentations qui utilise la virgule flottante doivent suivre ces règles : un résultat en flonum doit être représenté avec au moins autant de précision que celles utilisées pour exprimer les arguments inexacts de cette opération. Il est souhaitable, mais non requis, que les opérations potentiellement inexactes, comme `sqrt`, lorsqu'on les applique à des arguments exacts, produisent des réponses exactes si possible (par exemple, la racine carrée de l'entier exact 4 est l'entier exact 2). Dans le cas où une telle opération prendrait un nombre exact pour produire un nombre inexact (comme avec `sqrt`), et si le résultat est représenté comme un flonum, alors le format flonum disponible le plus précis doit être utilisé; mais si le résultat est représenté d'une autre façon alors la représentation doit avoir au moins autant de précision que le format flonum disponible le plus précis.

Bien que Scheme autorise une grande variété de notations écrites, une implémentation particulière peut ne supporter qu'une partie de ces notations. Par exemple, une implémentation dans laquelle tous les nombres sont réels n'a pas besoin de supporter les notations cartésiennes et polaires pour les nombres complexes. Si une implémentation rencontre une constante numérique exacte qu'elle ne peut pas représenter comme un nombre exact, alors elle peut ou bien signaler une restriction d'implémentation, ou bien silencieusement représenter la constante par un nombre inexact.

6.5.4. Syntaxe des constantes numériques

La syntaxe des représentations écrites pour les nombres est décrite formellement à la section 7.1.1.

Un nombre peut être écrit en binaire, octal, décimal ou hexadécimal en utilisant un préfixe de base, respectivement `#b` (binary), `#o` (octal), `#d` (décimal), et `#x` (hexadécimal). En l'absence d'un tel préfixe, le système décimal est pris par défaut.

Une constante numérique peut être spécifiée exacte ou inexacte par un préfixe, à savoir `#e` ou `#i`. Un préfixe d'exactitude peut apparaître avant ou après un préfixe de base. En l'absence de préfixe d'exactitude, la représentation écrite d'une constante numérique sera inexacte dans le cas où elle comporte un point décimal, ou un exposant, ou un caractère "`#`" à la place d'un chiffre. Sinon, elle sera exacte.

Dans les systèmes munis de nombres inexacts de diverses précisions, il peut être utile de spécifier la précision d'une constante. A cet effet, les constantes numériques peuvent s'écrire avec un marqueur d'exposant qui indique la précision souhaitée de la représentation inexacte. Les lettres `s`, `f`, `d`, `l` spécifient une précision *short*, *single*, *double* et *long* respectivement. (Lorsqu'il existe moins de quatre représentations inexactes internes, les quatre tailles sont remplacées par celles qui sont disponibles. Par exemple, une implémentation avec deux représentations internes peuvent associer *short* et *single*, ainsi que *double* et *long*.) De plus, le marqueur d'exposant `e` spécifie la précision par défaut de l'implémentation. Celle-ci est au moins aussi précise que *double*, mais les implémentations voudront peut-être laisser l'utilisateur régler cette précision par défaut.

```
3.14159265358979F0
            Arrondir en single — 3.141593
0.6L0
            Etendre en long — .600000000000000
```

6.5.5. Opérations numériques

Le lecteur est renvoyé à la section 1.3.3 pour les conventions sur la notation des arguments suivant leurs types. Les exemples de cette section supposent que les constantes numériques écrites en notation exacte sont représentées comme des nombres exacts. Quelques exemples supposent aussi que certaines constantes numériques écrites en notation inexacte peuvent être représentées sans perte de précision; les constantes inexactes ont été choisies de telle sorte que cette hypothèse s'avère vraie dans les implémentations qui utilisent des flonums pour représenter les nombres inexacts.

`(number?` *obj*`)` procédure essentielle
`(complex?` *obj*`)` procédure essentielle
`(real?` *obj*`)` procédure essentielle
`(rational?` *obj*`)` procédure essentielle
`(integer?` *obj*`)` procédure essentielle

Ces prédicats de types numériques peuvent être appliqués à n'importe quels arguments, même non numériques. Ils retournent `#t` si l'objet est du type concerné, et `#f` sinon. En général, si un prédicat de type est vrai pour un nombre, alors tous les prédicats "au-dessus" sont vrais aussi pour ce nombre. Ou encore, si un prédicat de type est faux pour un nombre, tous les prédicats "au-dessous" sont faux pour ce nombre.

Si z est un nombre complexe inexact, alors `(real?` z`)` est vrai si et seulement si `(zero? (imag-part` z`))` est vrai. Si x est un nombre réel inexact, alors `(integer?` x`)` est vrai si et seulement si `(=` x `(round` x`))`.

```
(complex? 3+4i)          ⟹  #t
(complex? 3)             ⟹  #t
(real? 3)                ⟹  #t
(real? -2.5+0.0i)        ⟹  #t
(real? #e1e10)           ⟹  #t
(rational? 6/10)         ⟹  #t
(rational? 6/3)          ⟹  #t
(integer? 3+0i)          ⟹  #t
(integer? 3.0)           ⟹  #t
(integer? 8/4)           ⟹  #t
```

Note : Le comportement de ces prédicats de type sur les nombres inexacts n'est pas fiable, puisque toute perte de précision peut modifier le résultat.

Note : Dans certaines implémentations, la procédure `rational?` sera la même que `real?`, et la procédure `complex?` sera la même que `number?`, mais des implémentations sortant de l'ordinaire peuvent être capables de représenter quelques nombres irrationnels exactement ou peuvent étendre le système numérique pour supporter certains nombres non complexes.

(exact? z**)** procédure essentielle
(inexact? z**)** procédure essentielle

Ces prédicats numériques procurent des tests pour vérifier l'exactitude d'une quantité. Tout nombre Scheme vérifie un seul de ces deux prédicats.

(= z_1 z_2 z_3 ...**)** procédure essentielle
(< x_1 x_2 x_3 ...**)** procédure essentielle
(> x_1 x_2 x_3 ...**)** procédure essentielle
(<= x_1 x_2 x_3 ...**)** procédure essentielle
(>= x_1 x_2 x_3 ...**)** procédure essentielle

Ces procédures retournent `#t` si leurs arguments sont respectivement : égaux, strictement croissants, strictement décroissants, croissants ou décroissants. Ces prédicats sont transitifs.

Note : Les implémentations traditionnelles de ces prédicats dans les langages de type Lisp ne sont pas transitives.

Note : Bien que ce ne soit pas une erreur de comparer des nombres inexacts avec ces prédicats, les résultats peuvent être non fiables à cause d'une légère perte de précision pouvant altérer le résultat; ceci est particulièrement vrai pour `zero?` et `=`. En cas de doute, consultez votre analyste (numéricien).

(zero? z**)** procédure essentielle
(positive? x**)** procédure essentielle
(negative? x**)** procédure essentielle
(odd? n**)** procédure essentielle
(even? n**)** procédure essentielle

Ces prédicats numériques testent une propriété particulière d'un nombre, et retournent `#t` ou `#f`. Voir la note ci-dessus.

(max x_1 x_2 ...**)** procédure essentielle
(min x_1 x_2 ...**)** procédure essentielle

Ces procédures retournent le plus grand ou le plus petit de leurs arguments.

```
(max 3 4)        ⟹  4     ; exact
(max 3.9 4)      ⟹  4.0   ; inexact
```

Note : Si l'un au moins des arguments est inexact, le résultat sera inexact (sauf si l'implémentation peut prouver que la perte de précision n'est pas assez grande pour altérer le résultat, ce qui n'est possible que dans des implémentations sortant de l'ordinaire). Si `min` ou `max` sont utilisés pour comparer des nombres d'exactitudes diverses, et si la valeur numérique du résultat ne peut pas être représentée sous la forme d'un nombre inexact sans perte de précision, alors la procédure peut signaler une restriction d'implémentation.

(+ z_1 ...**)** procédure essentielle
(* z_1 ...**)** procédure essentielle

Ces procédures retournent la somme ou le produit de leurs arguments.

```
(+ 3 4)     ⟹  7
(+ 3)       ⟹  3
(+)         ⟹  0
(* 4)       ⟹  4
(*)         ⟹  1
```

(- z_1 z_2**)** procédure essentielle
(- z**)** procédure essentielle
(- z_1 z_2 ...**)** procédure
(/ z_1 z_2**)** procédure essentielle
(/ z**)** procédure essentielle
(/ z_1 z_2 ...**)** procédure

Avec au moins deux arguments, ces procédures retournent la différence ou le quotient de leurs arguments, associées à gauche. Avec un argument, elles en retournent l'opposé ou l'inverse.

```
(- 3 4)     ⟹  -1
(- 3 4 5)   ⟹  -6
(- 3)       ⟹  -3
(/ 3 4 5)   ⟹  3/20
(/ 3)       ⟹  1/3
```

(abs x**)** procédure essentielle

`Abs` retourne la valeur absolue de son argument.

```
(abs -7)    ⟹  7
```

(quotient n_1 n_2**)** procédure essentielle
(remainder n_1 n_2**)** procédure essentielle
(modulo n_1 n_2**)** procédure essentielle

Ces procédures implémentent la division entière. Pour des entiers positifs n_1 et n_2, si n_3 et n_4 sont des entiers tels que $n_1 = n_2 n_3 + n_4$ et $0 \leq n_4 < n_2$, alors

```
(quotient n1 n2)    ⟹  n3
(remainder n1 n2)   ⟹  n4
(modulo n1 n2)      ⟹  n4
```

Pour des entiers n_1 et n_2 avec n_2 non nul,

```
(= n₁ (+ (* n₂ (quotient n₁ n₂))
         (remainder n₁ n₂)))
                          ⟹  #t
```

pourvu que tous les nombres ci-dessus soient exacts.

La valeur retournée par `quotient` a toujours le signe du produit de ses arguments. `Remainder` et `modulo` diffèrent sur les arguments négatifs—le `remainder` est nul ou a le signe du dividende, tandis que le `modulo` a toujours le signe du diviseur:

```
(modulo 13 4)              ⟹  1
(remainder 13 4)           ⟹  1

(modulo -13 4)             ⟹  3
(remainder -13 4)          ⟹  -1

(modulo 13 -4)             ⟹  -3
(remainder 13 -4)          ⟹  1

(modulo -13 -4)            ⟹  -1
(remainder -13 -4)         ⟹  -1

(remainder -13 -4.0)       ⟹  -1.0  ; inexact
```

`(gcd` n_1 `...)` procédure essentielle
`(lcm` n_1 `...)` procédure essentielle

Ces procédures retournent le PGCD (en anglais LCD) ou le PPCM (en anglais LCM) de leurs arguments. Le résultat est toujours ≥ 0.

```
(gcd 32 -36)               ⟹  4
(gcd)                      ⟹  0
(lcm 32 -36)               ⟹  288
(lcm 32.0 -36)             ⟹  288.0  ; inexact
(lcm)                      ⟹  1
```

`(numerator` q`)` procédure
`(denominator` q`)` procédure

Ces procédures retournent le numérateur ou le dénominateur de q ; le résultat est calculé comme si la fraction était irréductible, avec un dénominateur positif. Le dénominateur de 0 est égal à 1.

```
(numerator (/ 6 4))        ⟹  3
(denominator (/ 6 4))      ⟹  2
(denominator
  (exact->inexact (/ 6 4))) ⟹ 2.0
```

`(floor` x`)` procédure essentielle
`(ceiling` x`)` procédure essentielle
`(truncate` x`)` procédure essentielle
`(round` x`)` procédure essentielle

Ces procédures retournent des entiers. `Floor` retourne le plus grand entier $\leq x$, tandis que `ceiling` retourne le plus petit entier $\geq x$. `Truncate` retourne l'entier le plus proche de x dont la valeur absolue n'est pas plus grande que la valeur absolue de x. `Round` retourne l'entier le plus proche de x, en arrondissant à un entier impair si x est au milieu de deux entiers consécutifs.

Rationale : `Round` arrondit à l'impair pour suivre le standard de virgule flottante IEEE.

Note : Si l'argument de l'une de ces procédures est inexact, alors le résultat sera inexact. Si une valeur exacte est nécessaire, le résultat devrait être passé à la procédure `inexact->exact`.

```
(floor -4.3)               ⟹  -5.0
(ceiling -4.3)             ⟹  -4.0
(truncate -4.3)            ⟹  -4.0
(round -4.3)               ⟹  -4.0

(floor 3.5)                ⟹  3.0
(ceiling 3.5)              ⟹  4.0
(truncate 3.5)             ⟹  3.0
(round 3.5)                ⟹  4.0  ; inexact

(round 7/2)                ⟹  4    ; exact
(round 7)                  ⟹  7
```

`(rationalize` x y`)` procédure

`Rationalize` retourne le nombre rationnel *le plus simple* à une distance de x ne dépassant pas y. Un nombre rationnel r_1 est *plus simple* qu'un nombre rationnel r_2 si $r_1 = p_1/q_1$ et $r_2 = p_2/q_2$ (irréductibles) et $|p_1| \leq |p_2|$ et $|q_1| \leq |q_2|$. Ainsi $3/5$ est plus simple que $4/7$. Bien que cette relation d'ordre ne soit que partielle (considérez $2/7$ et $3/5$), tout intervalle contient un rationnel plus simple que tout autre rationnel de cet intervalle (entre $2/7$ et $3/5$, le plus simple est $2/5$). Notez que $0 = 0/1$ est le rationnel le plus simple de tous.

```
(rationalize
  (inexact->exact .3) 1/10) ⟹ 1/3    ; exact
(rationalize .3 1/10)       ⟹ #i1/3  ; inexact
```

`(exp` z`)` procédure
`(log` z`)` procédure
`(sin` z`)` procédure
`(cos` z`)` procédure
`(tan` z`)` procédure
`(asin` z`)` procédure
`(acos` z`)` procédure
`(atan` z`)` procédure
`(atan` y x`)` procédure

Ces procédures font partie de toute implémentation qui supporte les nombres réels ; elles calculent les fonctions transcendantes usuelles. `Log` calcule le logarithme népérien de z (pas celui en base 10). `Asin`, `acos` et `atan` calculent l'arc sinus ($\sin^{-1}$), l'arc cosinus ($\cos^{-1}$) et l'arc tangente ($\tan^{-1}$). Avec deux arguments, `(atan` *x y*`)` calcule `(angle (make-rectangular` *x y*`))` (voir ci-dessous), même dans les implémentations qui ne supportent pas les nombres complexes généraux.

En général, les fonctions mathématiques log, asin, acos et atan sont à valeurs multiples. Pour $x \neq 0$ réel, la valeur de $\log x$ est définie comme celle dont la partie imaginaire est dans $]-\pi,\pi]$. $\log 0$ n'est pas défini. La valeur de $\log z$ pour z complexe est définie par la formule :

$$\log z = \log |z| + i \arg z$$

Avec log ainsi défini, les valeurs de $\sin^{-1}$, $\cos^{-1}$ et $\tan^{-1}$ sont définies par :

$$\sin^{-1} z = -i \log(iz + \sqrt{1-z^2})$$
$$\cos^{-1} z = \pi/2 - \sin^{-1} z$$
$$\tan^{-1} z = (\log(1+iz) - \log(1-iz))/(2i)$$

Les spécifications ci-dessus suivent [82], qui à son tour cite [59] ; référez-vous à ces sources pour des discussions plus détaillées sur les branches des fonctions multiformes, les conditions aux frontières et l'implémentation de ces fonctions. Lorsque cela est possible, ces procédures retournent un résultat réel à partir d'un argument réel.

`(sqrt` *z*`)` procédure

Retourne la racine carrée principale de z. Le résultat aura ou bien une partie réelle positive, ou bien une partie réelle nulle et une partie imaginaire positive ou nulle.

`(expt` z_1 z_2`)` procédure

Retourne z_1 élevé à la puissance z_2:

$$z_1{}^{z_2} = e^{z_2 \log z_1}$$

0^0 est défini comme étant égal à 1.

`(make-rectangular` x_1 x_2`)` procédure
`(make-polar` x_3 x_4`)` procédure
`(real-part` *z*`)` procédure
`(imag-part` *z*`)` procédure
`(magnitude` *z*`)` procédure
`(angle` *z*`)` procédure

Ces procédures font partie de toute implémentation qui supportent les nombres complexes. Supposons que x_1, x_2, x_3, et x_4 soient des nombres réels et que z soit un nombre complexe tel que :

$$z = x_1 + x_2 i = x_3 \cdot e^{i x_4}$$

Alors `make-rectangular` et `make-polar` retournent z, `real-part` retourne x_1, `imag-part` retourne x_2, `magnitude` retourne x_3, et `angle` retourne x_4. Dans le cas de `angle`, dont la règle précédente n'assure pas l'unicité, la valeur retournée sera la seule dans $]-\pi,\pi]$.

Rationale : `Magnitude` coïncide avec `abs` pour un argument réel, mais `abs` doit être présent dans toute implémentation, alors que `magnitude` ne sert que dans les implémentations qui supportent les nombres complexes.

`(exact->inexact` *z*`)` procédure
`(inexact->exact` *z*`)` procédure

`Exact->inexact` retourne une représentation inexacte de z. La valeur retournée est le nombre inexact numériquement le plus proche de l'argument. Si un argument exact n'a pas d'équivalent inexact raisonnablement proche, alors une restriction d'implémentation peut être signalée.

`Inexact->exact` retourne une représentation exacte de z. La valeur retournée est le nombre exact numériquement le plus proche de l'argument. Si un argument inexact n'a pas d'équivalent exact raisonnablement proche, alors une restriction d'implémentation peut être signalée.

Ces procédures implémentent la bijection naturelle entre les entiers exacts et inexacts à l'intérieur du domaine supporté par l'implémentation. Voir la section 6.5.3.

6.5.6. Les entrées/sorties numériques

`(number->string` *nombre*`)` procédure essentielle
`(number->string` *nombre base*`)` procédure essentielle

Base peut être un entier exact, parmi 2, 8, 10, or 16. S'il est omis, *base* vaut 10 par défaut. La procédure `number->string` prend un nombre et une base et retourne une représentation externe de ce nombre dans la base donnée sous forme d'une chaîne de caractères, de sorte que

```
(let ((nombre nombre)
      (base base))
  (eqv? number
        (string->number (number->string nombre
                                        base)
                        base)))
```

soit vrai. Si *nombre* est inexact, si la base est 10, et si l'expression précédente peut être satisfaite par un résultat qui contient un point décimal, alors le résultat contiendra un point décimal et sera exprimé avec le minimum de chiffres (sans exposant ni 0 en tête) nécessaires pour rendre l'expression précédente vraie [88, 10] ; sinon le format du résultat est non spécifié.

Le résultat retourné par `number->string` ne contient jamais un préfixe de base explicite.

Note : Le cas d'erreur peut survenir seulement lorsque *nombre* n'est pas un nombre complexe ou est un nombre complexe de partie réelle ou imaginaire non rationnelle.

Rationale : Si *nombre* est un nombre inexact représenté par un flonum, et que la base est 10, alors l'expression précédente est normalement satisfaite par un résultat contenant un point décimal. Le cas non spécifié autorise des infinités, NaNs, et représentations sans nombres flottants.

`(string->number` *chaîne*`)` procédure essentielle
`(string->number` *chaîne base*`)` procédure essentielle

Retourne un nombre dans la représentation la plus précise de la *chaîne*. *Base* doit être un entier exact, parmi 2, 8, 10 ou 16. Si elle est fournie, *base* est une base par défaut qui peut être remplacée par un préfixe de base explicite dans *chaîne* (ex: `"#o177"`). Si *base* est omis, alors la base par défaut est 10. Si *chaîne* n'est pas une notation numérique syntaxiquement valide, alors `string->number` retourne `#f`.

```
(string->number "100")        ⟹  100
(string->number "100" 16)     ⟹  256
(string->number "1e2")        ⟹  100.0
(string->number "15##")       ⟹  1500.0
```

Note : Bien que `string->number` soit une procédure essentielle, une implémentation peut restreindre son domaine de l'une des manières suivantes. `String->number` a le droit de retourner `#f` lorsque *chaîne* contient un préfixe de base explicite. Si tous les nombres d'une implémentation sont réels, alors `string->number` a le droit de retourner `#f` lorsque *chaîne* utilise des notations complexes cartésiennes ou polaires. Si tous les nombres sont des entiers, alors `string->number` a le droit de retourner `#f` en cas d'utilisation d'une notation fractionnaire. Si tous les nombres sont exacts, alors `string->number` a le droit de retourner `#f` en cas d'utilisation d'un exposant ou d'un préfixe d'exactitude, ou si un `#` apparaît à la place d'un chiffre. Si tous les nombres inexacts sont des entiers, alors `string->number` a le droit de retourner `#f` en cas d'utilisation d'un point décimal.

6.6. Caractères

Les caractères sont des objets qui représentent les caractères imprimés tels les lettres et les chiffres. Les caractères sont écrits en utilisant la notation `#\`⟨caractère⟩ or `#\`⟨nom de caractère⟩. Par exemple:

```
#\a         ; lettre minuscule
#\A         ; lettre majuscule
#\(         ; parenthèse ouvrante
#\          ; caractère espace
#\space     ; le même en mieux
#\newline   ; caractère newline
```

La casse est significative dans `#\`⟨caractère⟩, mais pas dans `#\`⟨nom de caractère⟩. Si ⟨caractère⟩ dans `#\`⟨caractère⟩ est alphabétique, alors le caractère suivant ⟨caractère⟩ doit être un délimiteur tel que l'espace ou une parenthèse. Cette règle résoud le cas ambigü où, par exemple, la suite de caractères "`#\space`" pourrait être prise soit comme une représentation du caractère espace, soit comme une représentation du caractère "`#\s`" suivie d'une représentation du symbole "`pace`."

Les caractères écrits dans la notation `#\` sont auto-évaluants : il n'y a donc pas besoin de les quoter dans un programme.

Certaines des procédures qui opèrent sur les caractères ignorent la différence entre minuscule et majuscule. Les procédures qui ignore la casse ont "`-ci`" (pour "case insensitive") dans leurs noms.

`(char?` *obj*`)` procédure essentielle

Retourne `#t` si *obj* est un caractère, et `#f` sinon.

`(char=?` $char_1$ $char_2$`)` procédure essentielle
`(char<?` $char_1$ $char_2$`)` procédure essentielle
`(char>?` $char_1$ $char_2$`)` procédure essentielle
`(char<=?` $char_1$ $char_2$`)` procédure essentielle
`(char>=?` $char_1$ $char_2$`)` procédure essentielle

Ces procédures munissent l'ensemble des caractères d'un ordre total. Il est garanti qu'avec cet ordre :

- Les majuscules sont ordonnées. Par exemple : `(char<? #\A #\B)` retourne `#t`.
- Les minuscules sont ordonnées. Par exemple : `(char<? #\a #\b)` retourne `#t`.
- Les chiffres sont ordonnés. Par exemple : `(char<? #\0 #\9)` retourne `#t`.
- Ou bien tous les chiffres précèdent toutes les lettres majuscules, ou vice-versa.
- Ou bien tous les chiffres précèdent toutes les lettres minuscules, ou vice-versa.

Une implémentation particulière peut généraliser ces procédures de sorte qu'elles puissent accepter plus de deux arguments, comme avec les prédicats numériques.

`(char-ci=?` $char_1$ $char_2$`)` procédure essentielle
`(char-ci<?` $char_1$ $char_2$`)` procédure essentielle
`(char-ci>?` $char_1$ $char_2$`)` procédure essentielle
`(char-ci<=?` $char_1$ $char_2$`)` procédure essentielle
`(char-ci>=?` $char_1$ $char_2$`)` procédure essentielle

Ces procédures sont similaires à `char=?` etc, mais elles traitent de la même manière les lettres majuscules et minuscules. Par exemple, `(char-ci=? #\A #\a)` retourne `#t`. Certaines implémentations peuvent généraliser ces procédures pour qu'elles puissent accepter plus de deux arguments, comme avec les prédicats numériques.

`(char-alphabetic? ` *char*`)` procédure essentielle
`(char-numeric? ` *char*`)` procédure essentielle
`(char-whitespace? ` *char*`)` procédure essentielle
`(char-upper-case? ` *lettre*`)` procédure essentielle
`(char-lower-case? ` *lettre*`)` procédure essentielle

Ces procédures retournent `#t` si leurs arguments sont alphabétiques, numériques, blancs, majuscules, ou minuscules, respectivement, sinon elles retournent `#f`. Les remarques suivantes, spécifiques au code ASCII, ne sont là que comme guide : les caractères alphabétiques sont les 52 lettres minuscules et majuscules. Les caractères numériques sont les dix chiffres décimaux. Les caractères blancs sont l'espace, la tabulation, le saut de ligne, le saut de page et le retour chariot.

`(char->integer ` *char*`)` procédure essentielle
`(integer->char ` *n*`)` procédure essentielle

Etant donné un caractère, `char->integer` retourne une représentation entière exacte du caractère. Etant donné un entier exact qui est l'image d'un caractère par `char->integer`, `integer->char` retourne ce caractère. Ces procédures implémentent des bijections entre l'ensemble des caractères ordonné par `char<=?` et une certaine partie des entiers ordonnée par `<=`. C'est-à-dire, si

```
(char<=? a b) ⟹ #t   et   (<= x y) ⟹ #t
```

et x et y sont dans le domaine de `integer->char`, alors

```
(<= (char->integer a)
    (char->integer b))        ⟹  #t

(char<=? (integer->char x)
         (integer->char y)) ⟹  #t
```

`(char-upcase ` *char*`)` procédure essentielle
`(char-downcase ` *char*`)` procédure essentielle

Ces procédures retournent un caractère $char_2$ tel que `(char-ci=? ` *char* $char_2$`)`. En outre, si *char* est alphabétique, alors le résultat de `char-upcase` est en majuscule et le résultat de `char-downcase` est en minuscule.

6.7. Chaînes de caractères

Les chaînes (string) sont des suites de caractères. Les chaînes sont écrites sous la forme de suites de caractères entre des guillemets ("). Un guillemet peut être écrit à l'intérieur d'une chaîne en le faisant précéder d'un backslash (\), comme dans

```
"Le mot \"recursion\" a plusieurs sens."
```

Un backslash peut être écrit à l'intérieur d'une chaîne en le faisant précéder d'un autre backslash. Scheme ne spécifie pas l'effet d'un backslash à l'intérieur d'une chaîne s'il n'est pas suivi par un guillemet ou un backslash.

Une chaîne constante peut tenir sur plusieurs lignes, mais le contenu exact d'une telle chaîne n'est pas spécifié.

La *longueur* d'une chaîne est le nombre de ses caractères. Ce nombre est un entier naturel qui est fixé lorsque la chaîne est créée. Les *indices valides* d'une chaîne sont les entiers naturels exacts inférieurs à la longueur de la chaîne. Le premier caractère d'une chaîne a pour indice 0, le second a pour indice 1, etc.

Des phrases comme "les caractères de *chaîne* commençant à l'indice *début* et finissant à l'indice *fin*" signifient que l'indice *début* est compris et que l'indice *fin* est exclus. Donc si *début* et *fin* sont égaux, cela signifie une sous-chaîne vide, et si *début* est nul et si *fin* est la longueur de la *chaîne*, cela signifie la chaîne complète.

Certaines procédures opérant sur les chaînes ignorent la différence entre majuscules et minuscules. Les versions qui ignorent la casse ont "`-ci`" (pour "case insensitive") dans leurs noms.

`(string? ` *obj*`)` procédure essentielle

Retourne `#t` si *obj* est une chaîne, et sinon retourne `#f`.

`(make-string ` *k*`)` procédure essentielle
`(make-string ` *k char*`)` procédure essentielle

`Make-string` retourne une chaîne nouvellement allouée de longueur *k*. Si *char* est donné, alors tous les éléments de la chaîne sont initialisés à *char*, sinon le contenu de *string* est non spécifié.

`(string ` *char* ...`)` procédure essentielle

Retourne une chaîne nouvellement allouée composée des arguments.

`(string-length ` *chaîne*`)` procédure essentielle

Retourne le nombre de caractères de la *chaîne*.

`(string-ref ` *chaîne k*`)` procédure essentielle

k doit être un indice valide de *chaîne*. `String-ref` retourne le caractère *k* de la *chaîne* en utilisant une indexation partant de l'indice 0.

`(string-set! ` *chaîne k char*`)` procédure essentielle

k doit être un indice valide de *chaîne*, et *char* doit être un caractère. `String-set!` dépose *char* dans l'élément *k* de la *chaîne* et retourne une valeur non spécifiée.

```
(define (f) (make-string 3 #\*))
(define (g) "***")
(string-set! (f) 0 #\?)        ⟹ non spécifié
(string-set! (g) 0 #\?)        ⟹ erreur
(string-set! (symbol->string 'immutable)
             0
             #\?)              ⟹ erreur
```

`(string=?` *chaîne*$_1$ *chaîne*$_2$`)` procédure essentielle
`(string-ci=?` *chaîne*$_1$ *chaîne*$_2$`)` procédure essentielle

Retourne `#t` si les deux chaînes sont de même longueur et contiennent les mêmes caractères aux mêmes positions, sinon retourne `#f`. `String-ci=?` ne fait pas de différence entre majuscules et minuscules, au contraire de `string=?`.

`(string<?` *chaîne*$_1$ *chaîne*$_2$`)` procédure essentielle
`(string>?` *chaîne*$_1$ *chaîne*$_2$`)` procédure essentielle
`(string<=?` *chaîne*$_1$ *chaîne*$_2$`)` procédure essentielle
`(string>=?` *chaîne*$_1$ *chaîne*$_2$`)` procédure essentielle
`(string-ci<?` *chaîne*$_1$ *chaîne*$_2$`)` procédure essentielle
`(string-ci>?` *chaîne*$_1$ *chaîne*$_2$`)` procédure essentielle
`(string-ci<=?` *chaîne*$_1$ *chaîne*$_2$`)` procédure essentielle
`(string-ci>=?` *chaîne*$_1$ *chaîne*$_2$`)` procédure essentielle

Ces procédures sont les extensions lexicographiques aux chaînes des ordres correspondants sur les caractères. Par exemple, `string<?` est l'ordre lexicographique sur les chaînes induit par l'ordre `char<?` sur les caractères. Si deux chaînes ne sont pas de même longueur, mais sont les mêmes jusqu'à la longueur de la chaîne la plus courte, cette dernière est considérée comme lexicographiquement inférieure à la chaîne la plus longue.

Les implémentations peuvent généraliser ces procédures ainsi que `string=?` et `string-ci=?` pour leur faire accepter plus de deux arguments, comme avec les prédicats numériques correspondants.

`(substring` *chaîne début fin*`)` procédure essentielle

Chaîne doit être une chaîne, et *début* et *fin* doivent être des entiers exacts satisfaisant

$$0 \leq \textit{début} \leq \textit{fin} \leq (\texttt{string-length}\ \textit{chaîne}).$$

`Substring` retourne une chaîne nouvellement allouée formée des caractères de *chaîne* commençant à l'indice *début* (inclus) et terminant à l'indice *fin* (exclus).

`(string-append` *chaîne* ...`)` procédure essentielle

Retourne une chaîne nouvellement allouée dont les caractères forment la concaténation des chaînes données.

`(string->list` *chaîne*`)` procédure essentielle
`(list->string` *Lchars*`)` procédure essentielle

`String->list` retourne une liste nouvellement allouée des caractères qui forment la chaîne donnée. `List->string` retourne une chaîne nouvellement allouée formée des caractères dans la liste *Lchars*. `String->list` et `list->string` sont inverses vis-à-vis de `equal?`.

`(string-copy` *chaîne*`)` procédure

Retourne une copie nouvellement allouée de la *chaîne*.

`(string-fill!` *chaîne char*`)` procédure

Dépose *char* dans chaque élément de la *chaîne* et retourne une valeur non spécifiée.

6.8. Vecteurs

Les vecteurs sont des structures hétérogènes dont les éléments sont indexés par des entiers. Un vecteur occupe typiquement moins d'espace qu'une liste de même longueur, et le temps moyen d'accès à un élément arbitraire est typiquement moindre dans le cas d'un vecteur qu'avec une liste.

La longueur d'un vecteur est le nombre de ses éléments. Ce nombre est un entier naturel fixé à la création du vecteur. Les *indices valides* d'un vecteur sont les entiers naturels exacts inférieurs à la longueur du vecteur. Le premier élément d'un vecteur est d'indice nul, et le dernier élément a pour indice un de moins que la longueur du vecteur.

Les vecteurs s'écrivent avec la notation `#(`*obj* ...`)`. Par exemple, un vecteur de longueur 3 contenant le nombre zéro à l'indice 0, la liste `(1 2 3 4)` à l'indice 1, et la chaîne `"Cindy"` à l'indice 2 peut être écrit de la manière suivante:

```
#(0 (1 2 3 4) "Cindy")
```

Notez que ceci est la représentation externe d'un vecteur, pas une expression évaluant à un vecteur. Comme les listes constantes, un vecteur constant doit être quoté :

```
'#(0 (1 2 3 4) "Cindy")
        ⟹ #(0 (1 2 3 4) "Cindy")
```

`(vector?` *obj*`)` procédure essentielle

Retourne `#t` si *obj* est un vecteur, sinon retourne `#f`.

`(make-vector` *k*`)` procédure essentielle
`(make-vector` *k obj*`)` procédure

Retourne un vecteur nouvellement alloué de *k* éléments. Si un second argument est donné, chaque élément est initialisé à *obj*, sinon le contenu initial de chaque élément est non spécifié.

`(vector` *obj* ...`)` procédure essentielle

Retourne un vecteur nouvellement alloué dont les éléments contiennent les arguments donnés. Analogue à `list`.

```
(vector 'a 'b 'c)          ⟹  #(a b c)
```

(vector-length *vecteur***)** procédure essentielle

Retourne le nombre d'éléments du *vecteur*.

(vector-ref *vecteur k***)** procédure essentielle

k doit être un indice valide du *vecteur*. **Vector-ref** retourne le contenu de l'élément d'indice *k* du *vecteur*.

```
(vector-ref '#(1 1 2 3 5 8 13 21)
            5)
          ⟹  8
(vector-ref '#(1 1 2 3 5 8 13 21)
            (inexact->exact
              (round (* 2 (acos -1)))))
          ⟹ 13
```

(vector-set! *vecteur k obj***)** procédure essentielle

k doit être un indice valide de *vecteur*. **Vector-set!** dépose *obj* dans l'élément d'indice *k* du *vecteur*. La valeur retournée par **vector-set!** est non spécifiée.

```
(let ((vec (vector 0 '(2 2 2 2) "Cindy")))
  (vector-set! vec 1 '("Claudia" "Claudia"))
  vec)
          ⟹  #(0 ("Claudia" "Claudia") "Cindy")

(vector-set! '#(0 1 2) 1 "foo")
          ⟹  erreur  ; vecteur constant
```

(vector->list *vecteur***)** procédure essentielle
(list->vector *liste***)** procédure essentielle

Vector->list retourne une liste nouvellement allouée des objets contenus dans les éléments du *vecteur*. **List->vector** retourne un vecteur nouvellement alloué initialisé aux éléments de la *liste*.

```
(vector->list '#(dah dah didah))
          ⟹  (dah dah didah)
(list->vector '(dididit dah))
          ⟹  #(dididit dah)
```

(vector-fill! *vecteur obj***)** procédure

Dépose *obj* dans chaque élément du *vecteur*. La valeur retournée par **vector-fill!** est non spécifiée.

6.9. Procédures de contrôle

Ce chapitre décrit diverses procédures primitives permettant de contrôler le déroulement de l'exécution d'un programme de manière spéciale. Le prédicat **procédure?** est décrit ici.

(procedure? *obj***)** procédure essentielle

Retourne **#t** si *obj* est une procédure, sinon retourne **#f**.

```
(procedure? car)              ⟹  #t
(procedure? 'car)             ⟹  #f
(procedure? (lambda (x) (* x x)))
                              ⟹  #t
(procedure? '(lambda (x) (* x x)))
                              ⟹  #f
(call-with-current-continuation procedure?)
                              ⟹  #t
```

(apply *proc args***)** procédure essentielle
(apply *proc* arg_1 ... *args***)** procédure

Proc doit être une procédure et *args* doit être une liste. La première forme (essentielle) appelle *proc* avec les éléments de *args* comme arguments effectifs. La seconde forme est une généralisation de la première qui appelle *proc* avec les éléments de la liste **(append (list** arg_1 ...**)** *args***)** comme arguments effectifs.

```
(apply + (list 3 4))          ⟹  7

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))

((compose sqrt *) 12 75)      ⟹  30
```

(map *proc* $liste_1$ $liste_2$ ...**)** procédure essentielle

Les *listes* doivent être des listes, et *proc* doit être une procédure prenant autant d'arguments qu'il y a de *listes*. S'il y a plus d'une *liste*, elle doivent être de la même longueur. **Map** applique *proc* tout à tour aux éléments des *listes* de même position et retourne une liste des résultats dans l'ordre, de la gauche vers la droite. Par contre, l'ordre dynamique dans lequel *proc* est appliqué aux éléments des *listes* est non spécifié.

```
(map cadr '((a b) (d e) (g h)))
          ⟹  (b e h)

(map (lambda (n) (expt n n))
     '(1 2 3 4 5))
          ⟹  (1 4 27 256 3125)

(map + '(1 2 3) '(4 5 6))     ⟹  (5 7 9)
```

```
(let ((compteur 0))
  (map (lambda (bidon)
         (set! compteur (+ compteur 1))
         compteur)
       '(a b c)))             ⟹  non spécifié
```

`(for-each` *proc list*$_1$ *list*$_2$ ...`)` procédure essentielle

Les arguments de `for-each` sont analogues à ceux de `map`, mais `for-each` invoque *proc* pour ses effets de bord plutôt que pour ses valeurs. Contrairement à `map`, `for-each` va invoquer *proc* sur les éléments des *liste*s dans l'ordre, du premier élément au dernier, et la valeur retournée par `for-each` est non spécifiée.

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
              (vector-set! v i (* i i)))
            '(0 1 2 3 4))
  v)                          ⟹  #(0 1 4 9 16)
```

`(force` *promesse*`)` procédure

Force la valeur de *promesse* (voir `delay`, section 4.2.5). Si aucune valeur n'a déjà été calculée pour la promesse, une valeur est calculée et retournée. La valeur de la promesse est "mémorisée" de telle sorte que si elle est forcée une seconde fois, la valeur déjà calculée est retournée.

```
(force (delay (+ 1 2)))     ⟹  3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))
                            ⟹  (3 3)

(define un-flot
  (letrec ((next
            (lambda (n)
              (cons n (delay (next (+ n 1)))))))
    (next 0)))
(define head car)
(define tail
  (lambda (flot) (force (cdr flot))))

(head (tail (tail un-flot)))
                            ⟹  2
```

`Force` et `delay` s'utilisent surtout dans des programmes écrits dans un style fonctionnel. Les exemples qui suivent ne devraient pas être considérés comme suivant un bon style de programmation, mais ils illustrent le fait qu'une seule valeur est calculée pour une promesse, peu importe le nombre de fois qu'elle est forcée.

```
(define compteur 0)
(define p
  (delay (begin (set! compteur (+ compteur 1))
                (if (> compteur x)
                    compteur
                    (force p)))))
(define x 5)
p                           ⟹  une promesse
(force p)                   ⟹  6
p                           ⟹  encore une promesse
(begin (set! x 10)
       (force p))           ⟹  6
```

Voici une implémentation possible de `delay` et `force`. Les promesses sont implémentées ici comme des procédures sans arguments, et `force` invoque simplement son argument :

```
(define force
  (lambda (objet)
    (objet)))
```

Nous définissons l'expression

```
(delay ⟨expression⟩)
```

comme ayant la même signification que l'appel de procédure

```
(make-promesse (lambda () ⟨expression⟩)),
```

où `make-promesse` est définie comme suit :

```
(define make-promesse
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()
        (if result-ready?
            result
            (let ((x (proc)))
              (if result-ready?
                  result
                  (begin (set! result-ready? #t)
                         (set! result x)
                         result))))))))
```

Rationale : Une promesse peut faire référence à sa propre valeur, comme dans le dernier exemple ci-dessus. Forcer une telle promesse peut obliger à forcer la promesse une seconde fois avant que la valeur du premier forçage ait été calculée. Ceci complique la définition de `make-promesse`.

Certaines implémentations supportent diverses extensions:

- Invoquer `force` sur un objet qui n'est pas une promesse peut simplement retourner cet objet.
- Il peut n'exister aucun moyen par lequel une promesse puisse être opérationnellement distinguée de sa valeur forcée. C'est-à-dire que des expressions comme la suivante peuvent évaluer soit à `#t` ou à `#f`, suivant l'implémentation :

```
(eqv? (delay 1) 1)          ⟹  non spécifié
(pair? (delay (cons 1 2)))  ⟹  non spécifié
```

- Certaines implémentations peuvent implémenter du "forçage implicite", où la valeur d'une promesse est forcée par des procédures primitives comme `cdr` et `+`:

```
(+ (delay (* 3 7)) 13)      ⟹  34
```

`(call-with-current-continuation` *proc*`)`
procédure essentielle

Proc doit être une procédure d'un argument. La procédure `call-with-current-continuation` (ou `call/cc`) photographie la continuation courante (voir le rationale ci-dessous) sous la forme d'une "procédure d'échappement" et la passe en argument à *proc*. La procédure d'échappement est une procédure Scheme d'un argument telle que, si on lui passe plus tard une valeur, ignorera la continuation courante à ce moment-là et passera cette valeur à la continuation qui était active lorsque la procédure d'échappement a été créée.

La procédure d'échappement qui est passée à *proc* a une durée de vie illimitée tout comme les autres procédures de Scheme. Elle peut être stockée dans des variables ou des structures de données et peut être invoquée autant de fois que l'on veut.

Les exemples qui suivent montrent seulement les utilisations les plus courantes de `call/cc`. Si les programmes réels étaient aussi simples que ces exemples, il n'y aurait pas besoin d'une procédure aussi puissante que `call/cc`.

```
(call/cc
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #t))                     ⟹  -3

(define list-length
  (lambda (obj)
    (call/cc
      (lambda (return)
        (letrec ((r
                  (lambda (obj)
                    (cond ((null? obj) 0)
                          ((pair? obj)
                           (+ (r (cdr obj)) 1))
                          (else (return #f))))))
          (r obj))))))

(list-length '(1 2 3 4))     ⟹  4

(list-length '(a b . c))     ⟹  #f
```

Rationale : Une utilisation courante de `call/cc` concerne les sorties structurées, non locales, de boucles ou de corps de procédure, mais en fait `call/cc` est extrèmement utile pour implémenter une grande variété de structures de contrôle avancées.

Lorsqu'une expression Scheme est évaluée, il y a une *continuation* attendant la valeur de l'expression. La continuation représente un futur entier (par défaut) du calcul. Si l'expression est évaluée au toplevel, par exemple, alors la continuation prendrait le résultat, l'écrirait sur l'écran, attendrait l'expression suivante, l'évaluerait, et ainsi de suite. La plupart du temps, la continuation inclut des actions spécifiées par du code utilisateur, comme dans une continuation qui prendrait le résultat, le multiplierait par la valeur d'une variable locale, ajouterait sept, et rendrait la réponse à la continuation du toplevel pour affichage. Normalement ces continuations sont cachées derrière la scène et les programmeurs n'y pensent pas beaucoup. Dans de rares occasions, cependant, un programmeur peut avoir besoin d'utiliser des continuations explicitement. `Call/cc` permet aux programmeurs Scheme de le faire en créant une procédure qui se comporte comme la continuation courante.

La plupart des langages de programmation incorporent une ou plusieurs constructions d'échappement pour des usages spéciaux, avec des noms comme `exit`, `return`, ou même `goto`. En 1965, cependant, Peter Landin [54] inventa un opérateur nommé `J` d'échappement général. John Reynolds [68] décrivit un mécanisme plus simple mais aussi puissant en 1972. La forme spéciale `catch` décrite par Sussman et Steele dans le rapport de 1975 sur Scheme est exactement la mme que la construction de Reynolds, bien que son nom provienne d'une construction moins générale de MacLisp. Plusieurs implémenteurs Scheme remarquèrent que la pleine puissance de la construction `catch` pouvait provenir d'une procédure plutôt que d'une construction syntaxique spéciale, et le nom `call-with-current-continuation` émergea en 1982. Ce nom est descriptif, mais on trouve diverses opinions quant aux mérites de ce nom à rallonge, et la plupart des gens utilisent plutôt le synonyme `call/cc`.

6.10. Les entrées-sorties

6.10.1. Ports

Les ports représentent les organes d'entrée et de sortie. Pour Scheme, un port d'entrée est un objet Scheme qui peut fournir des caractères à la demande, tandis qu'un port de sortie est un objet Scheme qui peut accepter des caractères.

`(call-with-input-file` *chaîne proc*`)`
procédure essentielle
`(call-with-output-file` *chaîne proc*`)`
procédure essentielle

Proc doit être une procédure à un argument, et *chaîne* doit représenter le nom d'un fichier. Pour `call-with-input-file`, le fichier doit déjà exister ; pour `call-with-output-file`, l'effet est non spécifié si le fichier existe déjà. Ces procédures invoquent *proc* avec un argument : le port obtenu en ouvrant le fichier en question en entrée ou en sortie. Si le fichier ne peut pas être ouvert, une erreur est signalée. Si la procédure termine, le port est automatiquement fermé et la valeur obtenue par la procédure est retournée. Si la procédure ne termine pas, le port ne sera pas fermé automatiquement sauf s'il est possible de prouver qu'il ne sera plus jamais utilisé pour une opération de lecture ou d'écriture.

Rationale : Puisque les procédures d'échappement de Scheme ont une durée de vie illimitée, il est possible de s'échapper de la continuation courante, mais d'y revenir plus tard. Si les implémentations avaient le droit de fermer le port lors d'un échappement de la continuation courante, il serait impossible d'écrire du code portable contenant à la fois `call-with-current-continuation` et `call-with-input-file` ou `call-with-output-file`.

`(input-port?` *obj*`)` procédure essentielle
`(output-port?` *obj*`)` procédure essentielle

Retourne `#t` si *obj* est un port d'entrée ou de sortie respectivement, sinon retourne `#f`.

`(current-input-port)` procédure essentielle
`(current-output-port)` procédure essentielle

Retourne le port d'entrée ou de sortie courant.

`(with-input-from-file` *chaîne proc*`)` procédure
`(with-output-to-file` *chaîne proc*`)` procédure

Proc doit être une procédure sans argument, et *chaîne* doit être une chaîne nommant un fichier. Pour `with-input-from-file`, le fichier doit déjà exister; pour `with-output-to-file`, l'effet est non spécifié si le fichier existe déjà. Le fichier est ouvert en lecture ou en écriture, un port d'entrée ou de sortie lui est connecté qui devient la valeur par défaut de `current-input-port` ou `current-output-port`, et *proc* est invoquée sans arguments. Lorsque *proc* termine, le port est fermé et le port par défaut précédent est restauré. `With-input-from-file` et `with-output-to-file` retournent la valeur obtenue par *proc*. Si une procédure d'échappement est utilisée pour s'échapper de la continuation de ces procédures, leur comportement dépend de l'implémentation.

`(open-input-file` *chaîne*`)` procédure essentielle

Prend une *chaîne* nommant un fichier existant et retourne un port d'entrée capable de fournir des caractères à partir du fichier. Si le fichier ne peut pas être ouvert, une erreur est signalée.

`(open-output-file` *chaîne*`)` procédure essentielle

Prend une *chaîne* nommant un fichier de sortie devant être créé et retourne un port de sortie capable d'écrire des caractères dans un nouveau fichier ayant ce nom. Si le fichier ne peut pas être ouvert, une erreur est signalée. Si un fichier de même nom existe déjà, l'effet est non spécifié.

`(close-input-port` *port*`)` procédure essentielle
`(close-output-port` *port*`)` procédure essentielle

Ferme le fichier associé à *port*, rendant ainsi le *port* incapable de fournir ou d'accepter des caractères. Ces routines n'ont aucun effet si le fichier a déjà été fermé. La valeur de retour est non spécifiée.

6.10.2. Lecture

`(read)` procédure essentielle
`(read` *port*`)` procédure essentielle

`Read` convertit des représentations externes d'objets Scheme en les objets eux-mêmes. Autrement dit, il s'agit d'un analyseur pour le non terminal ⟨donnée⟩ (voir les sections ?? et 6.3). `Read` retourne le prochain objet analysable sur le *port* donné, mettant à jour le *port* en le faisant pointer vers le premier caractère suivant la fin de la représentation externe de l'objet.

Si une fin de fichier est rencontrée durant la lecture avant d'avoir trouvé un caractère débutant un objet, un objet fin de fichier est retourné. Le port reste ouvert, et toute tentative ultérieure de lecture retournera aussi l'objet fin de fichier. Si une fin de fichier est rencontrée après le début de la représentation externe d'un objet mais si cette représentation est incomplète, et donc non analysable, une erreur est signalée.

L'argument *port* peut être omis, auquel cas il vaudra par défaut celui retourné par `current-input-port`. C'est une erreur d'essayer de lire dans un port fermé.

`(read-char)` procédure essentielle
`(read-char` *port*`)` procédure essentielle

Retourne le prochain caractère disponible sur le *port* d'entrée, mettant à jour le *port* en le faisant pointer vers le caractère suivant. S'il n'y a plus de caractère disponible, un objet fin de fichier est retourné. *Port* peut être omis, auquel cas il vaudra par défaut celui retourné par `current-input-port`.

`(peek-char)` procédure essentielle
`(peek-char` *port*`)` procédure essentielle

Retourne le prochain caractère disponible sur le *port*, sans mettre à jour le *port* en le faisant pointer vers le caractère suivant. S'il n'y a plus de caractère disponible, un objet fin de fichier est retourné. *Port* peut être omis, auquel cas il vaudra par défaut celui retourné par `current-input-port`.

Note : La valeur retournée par un appel à `peek-char` est la même que celle qui aurait été retournée par un appel à `read-char` avec le même *port*. La seule différence est que l'appel suivant à `read-char` ou `peek-char` sur ce *port* retournera la valeur retournée par l'appel précédent à `peek-char`. En particulier, un appel à `peek-char` sur un port interactif sera bloqué dans l'attente d'une entrée à partir du moment où un appel à `read-char` aurait été bloqué.

`(eof-object? `*obj*`)` procédure essentielle

Retourne `#t` si *obj* est un objet fin de fichier, sinon retourne `#f`. L'ensemble précis des objets fin de fichier varie suivant les implémentations, mais aucun objet fin de fichier ne peut être lu par `read`.

`(char-ready?)` procédure
`(char-ready? `*port*`)` procédure

Retourne `#t` si un caractère est prêt sur le *port* d'entrée et retourne `#f` sinon. Si `char-ready` retourne `#t` alors l'opération `read-char` suivant sur le *port* donné ne se bloquera pas. Si le *port* est en fin de fichier, alors `char-ready?` retourne `#t`. *Port* peut être omis, auquel cas il vaudra par défaut celui retourné par `current-input-port`.

Rationale : `Char-ready?` existe afin de permettre à un programme d'accepter des caractères de ports interactifs sans se retrouver bloqué dans l'attente d'une entrée. Tout éditeur de texte associé à de tels ports doit s'assurer que les caractères dont l'existence a été garantie par `char-ready?` ne peuvent pas être effacés. Si `char-ready?` devait retourner `#f` en fin de fichier, on ne pourrait pas distinguer un port en fin de fichier d'un port interactif n'ayant aucun caracatère prêt.

6.10.3. Ecriture

`(write `*obj*`)` procédure essentielle
`(write `*obj port*`)` procédure essentielle

Ecrit une représentation écrite de *obj* sur le *port* donné. Les chaînes qui apparaissent dans la représentation écrite sont entourées de guillemets, et à l'intérieur de ces chaînes les caractères backslash et guillemet sont précédés de backslashs. `Write` retourne une valeur non spécifiée. L'argument *port* peut être omis, auquel cas il vaudra par défaut celui retourné par `current-output-port`.

`(display `*obj*`)` procédure essentielle
`(display `*obj port*`)` procédure essentielle

Ecrit une représentation de *obj* sur le *port* donné. Les chaînes qui apparaissent dans la représentation écrite ne sont pas entourées de guillemets, et à l'intérieur de ces chaînes aucun caractère n'est précédé de backslash. Les objets de type caractères apparaissent dans la représentation comme s'ils avaient été écrits par `write-char` au lieu de `write`. `Display` retourne une valeur non spécifiée. L'argument *port* peut être omis, auquel cas il vaudra par défaut celui retourné par `current-output-port`.

Rationale : `Write` sert à produire des écritures lisibles par la machine tandis que `display` sert à produire des écritures lisibles par l'homme. Les implémentations qui permettent la "slashification" à l'intérieur des symboles voudront probablement écrire par `write` mais pas par `display` pour slashifier des caractères bizarres dans les symboles.

`(newline)` procédure essentielle
`(newline `*port*`)` procédure essentielle

Ecrit une fin de ligne sur *port*. Comment ceci est réalisé dépend du système d'exploitation utilisé. Retourne une valeur non spécifiée. L'argument *port* peut être omis, auquel cas il vaudra par défaut celui retourné par `current-output-port`.

`(write-char `*char*`)` procédure essentielle
`(write-char `*char port*`)` procédure essentielle

Ecrit le caractère *char* (pas une représentation externe du caractère) sur le *port* donné et retourne une valeur non spécifiée. L'argument *port* peut être omis, auquel cas il vaudra par défaut celui retourné par `current-output-port`.

6.10.4. Interface système

Les questions d'interface système tombent en général en-dehors du domaine de ce rapport. Cependant, les opérations suivantes sont suffisamment importantes pour les décrire ici.

`(load `*chaîne*`)` procédure essentielle

Chaîne doit être le nom d'un fichier contenant du source Scheme. La procédure `load` lit des expressions et des définitions à partir du fichier et les évalue en séquence. On ne spécifie pas si les résultats des expressions doivent être affichés. La procédure `load` ne modifie pas les valeurs retournées par `current-input-port` et `current-output-port`. `Load` retourne une valeur non spécifiée.

`(transcript-on `*chaîne*`)` procédure
`(transcript-off)` procédure

Chaîne doit être le nom d'un fichier devant être créé. L'effet de `transcript-on` consiste à ouvrir le fichier nommé en écriture, et à faire écho dans ce fichier de l'interaction entre l'utilisateur et le système Scheme. Cet écho se termine par un appel à `transcript-off`, qui ferme le fichier de transcript. Un transcript au plus peut être ouvert à un instant donné, bien que certaines implémentations puissent relâcher cette contrainte. Les valeurs retournées par ces procédures ne sont pas spécifiées.

7. Syntaxe et sémantique formelles

Ce chapitre formalise ce qui a déjà été décrit informellement dans les chapitres précédents de ce rapport.

7.1. Syntaxe formelle

Cette section décrit une syntaxe formelle de Scheme dans une notation BNF étendue. La syntaxe du langage entier est fournie, y-compris les traits non essentiels.

Tous les espaces dans la grammaire font partie du méta-langage. La casse n'est pas significative ; par exemple, `#x1A` et `#X1a` sont équivalents. ⟨vide⟩ dénote la chaîne vide.

On utilisera les extensions suivantes de la BNF pour rendre les descriptions plus concises : ⟨chose⟩* signifie zéro ou plus occurences de ⟨chose⟩; et ⟨chose⟩$^+$ signifie au moins une ⟨chose⟩.

7.1.1. Structure lexicale

Cette section décrit comment les tokens individuels (identificateurs, nombres, etc) sont formés à partir de suites de caractères. Les sections suivantes décrivent comment les expressions et les programmes sont formés à partir de suites de tokens.

⟨Espace intertoken⟩ peut se trouver d'un côté ou de l'autre d'un token, mais pas à l'intérieur d'un token.

Les tokens qui nécessitent une terminaison implicite (identificateurs, nombres, caractères, et le point) peuvent se terminer par n'importe quel ⟨délimiteur⟩, mais pas nécessairement par autre chose.

```
⟨token⟩ ⟶ ⟨ident⟩ | ⟨bool⟩ | ⟨nombre⟩ | ⟨char⟩
     | ⟨string⟩ | ( | ) | #( | ' | ` | , | ,@ | .
⟨delimiteur⟩ ⟶ ⟨blanc⟩ | ( | ) | " | ;
⟨blanc⟩ ⟶ ⟨espace ou newline⟩
⟨commentaire⟩ ⟶ ; ⟨tous les char suivants
                         jusqu'a la fin de ligne⟩
⟨atmosphere⟩ ⟶ ⟨blanc⟩ | ⟨commentaire⟩
⟨espace intertoken⟩ ⟶ ⟨atmosphere⟩*

⟨ident⟩ ⟶ ⟨initial⟩ ⟨suivant⟩*
     | ⟨ident particulier⟩
⟨initial⟩ ⟶ ⟨lettre⟩ | ⟨initial special⟩
⟨lettre⟩ ⟶ a | b | c | ... | z
⟨initial special⟩ ⟶ ! | $ | % | & | * | / | : | < | =
     | > | ? | ~ | _ | ^
⟨suivant⟩ ⟶ ⟨initial⟩ | ⟨chiffre⟩
     | ⟨suivant special⟩
⟨chiffre⟩ ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨suivant special⟩ ⟶ . | + | -
⟨ident particulier⟩ ⟶ + | - | ...
⟨mot-cle syntaxique⟩ ⟶ ⟨mot-cle d'expression⟩
     | else | => | define
     | unquote | unquote-splicing
⟨mot-cle d'expression⟩ ⟶ quote | lambda | if
     | set! | begin | cond | and | or | case
     | let | let* | letrec | do | delay
     | quasiquote

⟨variable⟩ ⟶ ⟨tout ⟨ident⟩ qui n'est pas
                aussi un ⟨mot-cle syntaxique⟩⟩

⟨bool⟩ ⟶ #t | #f
⟨char⟩ ⟶ #\ ⟨tout char⟩
     | #\ ⟨nom de char⟩
⟨nom de char⟩ ⟶ space | newline

⟨string⟩ ⟶ "⟨element de string⟩*"
⟨element de string⟩ ⟶ ⟨tout char autre que " ou \⟩
     | \" | \\

⟨nombre⟩ ⟶ ⟨num 2⟩| ⟨num 8⟩ | ⟨num 10⟩ | ⟨num 16⟩
```

Les règles suivantes pour ⟨num R⟩, ⟨complex R⟩, ⟨real R⟩, ⟨ureal R⟩, ⟨uinteger R⟩, et ⟨prefixe R⟩ sont valables pour $R = 2, 8, 10,$ et 16. Il n'y a aucune règle pour ⟨decimal 2⟩, ⟨decimal 8⟩, and ⟨decimal 16⟩, ce qui signifie que les nombres contenant un point décimal ou un exposant doivent être en base 10.

```
⟨num R⟩ ⟶ ⟨prefixe R⟩ ⟨complex R⟩
⟨complex R⟩ ⟶ ⟨real R⟩ | ⟨real R⟩@⟨real R⟩
     | ⟨real R⟩+⟨ureal R⟩i | ⟨real R⟩-⟨ureal R⟩i
     | ⟨real R⟩+i | ⟨real R⟩-i
     | +⟨ureal R⟩i | -⟨ureal R⟩i | +i | -i
⟨real R⟩ ⟶ ⟨signe⟩ ⟨ureal R⟩
⟨ureal R⟩ ⟶ ⟨uinteger R⟩
     | ⟨uinteger R⟩ / ⟨uinteger R⟩
     | ⟨decimal R⟩
⟨decimal 10⟩ ⟶ ⟨uinteger 10⟩ ⟨suffixe⟩
     | . ⟨chiffre 10⟩+ #* ⟨suffixe⟩
     | ⟨chiffre 10⟩+ . ⟨chiffre 10⟩* #* ⟨suffixe⟩
     | ⟨chiffre 10⟩+ #+ . #* ⟨suffixe⟩
⟨uinteger R⟩ ⟶ ⟨chiffre R⟩+ #*
⟨prefixe R⟩ ⟶ ⟨base R⟩ ⟨exactitude⟩
     | ⟨exactitude⟩ ⟨base R⟩

⟨suffixe⟩ ⟶ ⟨vide⟩
     | ⟨marque d'exposant⟩ ⟨signe⟩ ⟨chiffre 10⟩+
⟨marque d'exposant⟩ ⟶ e | s | f | d | l
⟨signe⟩ ⟶ ⟨vide⟩ | + | -
⟨exactitude⟩ ⟶ ⟨vide⟩ | #i | #e
⟨base 2⟩ ⟶ #b
⟨base 8⟩ ⟶ #o
⟨base 10⟩ ⟶ ⟨vide⟩ | #d
⟨base 16⟩ ⟶ #x
⟨chiffre 2⟩ ⟶ 0 | 1
⟨chiffre 8⟩ ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
⟨chiffre 10⟩ ⟶ ⟨chiffre⟩
```

⟨chiffre 16⟩ ⟶ ⟨chiffre 10⟩ | a | b | c | d | e | f

7.1.2. Représentations externes

⟨Donnee⟩ est ce que la procédure `read` (section 6.10.2) analyse avec succès. Notez qu'une string qui s'analyse comme une ⟨expression⟩ s'analyse aussi comme une ⟨donnee⟩.

⟨donnee⟩ ⟶ ⟨donnee simple⟩ | ⟨donnee composee⟩
⟨donnee simple⟩ ⟶ ⟨bool⟩ | ⟨nombre⟩ | ⟨char⟩
  | ⟨string⟩ | ⟨symbol⟩
⟨symbol⟩ ⟶ ⟨ident⟩
⟨donnee composee⟩ ⟶ ⟨list⟩ | ⟨vector⟩
⟨list⟩ ⟶ (⟨donnee⟩*) | (⟨donnee⟩+ . ⟨donnee⟩)
  | ⟨abbreviation⟩
⟨abbreviation⟩ ⟶ ⟨prefixe d'abbrev⟩ ⟨donnee⟩
⟨prefixe d'abbrev⟩ ⟶ ' | ` | , | ,@
⟨vector⟩ ⟶ #(⟨donnee⟩*)

7.1.3. Expressions

⟨expression⟩ ⟶ ⟨variable⟩
  | ⟨litteral⟩
  | ⟨appel de procedure⟩
  | ⟨lambda expression⟩
  | ⟨conditionnelle⟩
  | ⟨affectation⟩
  | ⟨expression derivee⟩

⟨litteral⟩ ⟶ ⟨quotation⟩ | ⟨auto-evaluant⟩
⟨auto-evaluant⟩ ⟶ ⟨bool⟩ | ⟨nombre⟩
  | ⟨char⟩ | ⟨string⟩
⟨quotation⟩ ⟶ '⟨donnee⟩ | (quote ⟨donnee⟩)
⟨appel de procedure⟩ ⟶ (⟨operateur⟩ ⟨operande⟩*)
⟨operateur⟩ ⟶ ⟨expression⟩
⟨operande⟩ ⟶ ⟨expression⟩

⟨lambda expression⟩ ⟶ (lambda ⟨formels⟩ ⟨corps⟩)
⟨formels⟩ ⟶ (⟨variable⟩*) | ⟨variable⟩
  | (⟨variable⟩+ . ⟨variable⟩)
⟨corps⟩ ⟶ ⟨definition⟩* ⟨sequence⟩
⟨sequence⟩ ⟶ ⟨commande⟩* ⟨expression⟩
⟨commande⟩ ⟶ ⟨expression⟩

⟨conditionnelle⟩ ⟶ (if ⟨test⟩ ⟨sivrai⟩ ⟨sifaux⟩)
⟨test⟩ ⟶ ⟨expression⟩
⟨sivrai⟩ ⟶ ⟨expression⟩
⟨sifaux⟩ ⟶ ⟨expression⟩ | ⟨vide⟩

⟨affectation⟩ ⟶ (set! ⟨variable⟩ ⟨expression⟩)

⟨expression derivee⟩ ⟶
  (cond ⟨clause de cond⟩+)
  | (cond ⟨clause de cond⟩* (else ⟨sequence⟩))
  | (case ⟨expression⟩
    ⟨clause de case⟩+)
  | (case ⟨expression⟩
    ⟨clause de case⟩*
    (else ⟨sequence⟩))
  | (and ⟨test⟩*)
  | (or ⟨test⟩*)
  | (let (⟨liaison⟩*) ⟨corps⟩)
  | (let ⟨variable⟩ (⟨liaison⟩*) ⟨corps⟩)
  | (let* (⟨liaison⟩*) ⟨corps⟩)
  | (letrec (⟨liaison⟩*) ⟨corps⟩)
  | (begin ⟨sequence⟩)
  | (do (⟨iteration spec⟩*)
      (⟨test⟩ ⟨sequence⟩)
    ⟨commande⟩*)
  | (delay ⟨expression⟩)
  | ⟨quasiquotation⟩

⟨clause de cond⟩ ⟶ (⟨test⟩ ⟨sequence⟩) | (⟨test⟩)
⟨clause de case⟩ ⟶ ((⟨donnee⟩*) ⟨sequence⟩)

⟨liaison⟩ ⟶ (⟨variable⟩ ⟨expression⟩)
⟨iteration spec⟩ ⟶ (⟨variable⟩ ⟨init⟩ ⟨pas⟩)
  | (⟨variable⟩ ⟨init⟩)
⟨init⟩ ⟶ ⟨expression⟩
⟨pas⟩ ⟶ ⟨expression⟩

7.1.4. Quasiquotations

La grammaire suivante pour les expressions quasiquote n'est pas context-free. Elle est présentée comme une recette pour engendrer un nombre infini de règles de production. Imaginez une copie des règles suivantes pour $D = 1, 2, 3, \ldots$. D mémorise le niveau d'imbrication.

⟨quasiquotation⟩ ⟶ ⟨quasiquotation 1⟩
⟨modele 0⟩ ⟶ ⟨expression⟩
⟨quasiquotation D⟩ ⟶ `⟨modele D⟩
  | (quasiquote ⟨modele D⟩)
⟨modele D⟩ ⟶ ⟨donnee simple⟩
  | ⟨modele de liste D⟩
  | ⟨modele de vecteur D⟩
  | ⟨unquotation D⟩
⟨modele de liste D⟩ ⟶ (⟨modele ou splice D⟩*)
  | (⟨modele or splice D⟩+ . ⟨modele D⟩)
  | '⟨modele D⟩
  | ⟨quasiquotation $D+1$⟩
⟨modele de vecteur D⟩ ⟶ #(⟨modele ou splice D⟩*)
⟨unquotation D⟩ ⟶ ,⟨modele $D-1$⟩
  | (unquote ⟨modele $D-1$⟩)
⟨modele ou splice D⟩ ⟶ ⟨modele D⟩
  | ⟨splicing unquotation D⟩
⟨splicing unquotation D⟩ ⟶ ,@⟨modele $D-1$⟩

| `(unquote-splicing` ⟨modele $D-1$⟩`)`

Dans les ⟨quasiquotation⟩s, un ⟨modele de liste D⟩ peut parfois être confondu avec soit une ⟨unquotation D⟩ ou une ⟨splicing unquotation D⟩. L'interprétation sous la forme d'une ⟨unquotation⟩ ou d'une ⟨splicing unquotation D⟩ est prioritaire.

7.1.5. Programmes et définitions

⟨programme⟩ ⟶ ⟨commande ou definition⟩*
⟨commande ou definition⟩ ⟶ ⟨commande⟩ | ⟨definition⟩
⟨definition⟩ ⟶ `(define` ⟨variable⟩ ⟨expression⟩`)`
  | `(define (`⟨variable⟩ ⟨parametres⟩`)` ⟨corps⟩`)`
  | `(begin` ⟨definition⟩*`)`
⟨parametres⟩ ⟶ ⟨variable⟩*
  | ⟨variable⟩+ `.` ⟨variable⟩

7.2. Sémantique formelle

Cette section fournit une sémantique dénotationnelle formelle des expressions primitives de Scheme et de certaines procédures primitives. Les concepts et notations utilisés sont décrits dans [92] ; voici un sommaire de la notation :

$\langle\ldots\rangle$	formation d'une suite
$s \downarrow k$	k-eme élément de la suite s ($k \geq 1$)
$\#s$	longueur d'une suite s
$s \S t$	concaténation des suites s et t
$s \dagger k$	saute les k premiers éléments de la suite s
$t \rightarrow a, b$	conditionnelle de McCarthy "if t then a else b"
$\rho[x/i]$	substitution "ρ où x remplace i"
x in $\mathtt{D}$	injection de x dans le domaine $\mathtt{D}$
$x \mid \mathtt{D}$	projection de x sur le domaine $\mathtt{D}$

La raison pour laquelle les continuations d'expressions prennent des suites de valeurs au lieu de valeurs uniques, est que ceci simplifie le traitement formel des appels de procédure et que cela facilite l'ajout des valeurs de retour multiples.

Le drapeau booléen associé aux doublets, vecteurs et chaînes vaudra vrai pour les objets mutables et faux pour les objets non mutables.

L'ordre d'évaluation dans un appel est non spécifié. Nous simulons ce fait en appliquant des permutations arbitraires *permute* et *unpermute*, qui doivent être inverses l'une de l'autre, aux arguments d'un appel avant et après qu'ils aient été évalués. Ceci n'est pas vraiment rigoureux car cela suggère, incorrectement, que l'ordre d'évaluation est constant à l'intérieur d'un programme (pour un nombre donné d'arguments), mais c'est une meilleure approximation de la sémantique souhaitée que celle qui fixerait une évaluation de gauche à droite.

L'allocateur de mémoire *new* dépend de l'implémentation, mais il doit satisfaire l'axiome suivant : si $new\,\sigma \in \mathtt{L}$, alors $\sigma\,(new\,\sigma \mid \mathtt{L}) \downarrow 2 = false$.

La définition de $\mathcal{K}$ est omise car une définition propre de $\mathcal{K}$ compliquerait la sémantique pour un résultat peu intéressant.

Si P est un programme dans lequel toutes les variables sont définies avant d'être référencées ou affectées, alors la signification de P est

$$\mathcal{E}[\![\texttt{((lambda (}\mathrm{I}^*\texttt{) }\mathrm{P}'\texttt{)}\ \langle\text{undefined}\rangle\ \ldots\texttt{)}]\!]$$

où I* est la suite des variables définies dans P, P′ est la suite des expressions obtenues en remplaçant chaque définition dans P par une affectation, ⟨undefined⟩ est une expression qui évalue à *undefined*, et $\mathcal{E}$ est la fonction sémantique qui donne un sens aux expressions.

7.2.1. Syntaxe abstraite

$\mathrm{K} \in \mathrm{Con}$	constantes, y-compris les quotations
$\mathrm{I} \in \mathrm{Ide}$	identificateurs (variables)
$\mathrm{E} \in \mathrm{Exp}$	expressions
$\Gamma \in \mathrm{Com} = \mathrm{Exp}$	commandes

Exp ⟶ K | I | $(\mathrm{E}_0\ \mathrm{E}^*)$
  | `(lambda` $(\mathrm{I}^*)\ \Gamma^*\ \mathrm{E}_0$`)`
  | `(lambda` $(\mathrm{I}^*$ `.` $\mathrm{I})\ \Gamma^*\ \mathrm{E}_0$`)`
  | `(lambda` $\mathrm{I}\ \Gamma^*\ \mathrm{E}_0$`)`
  | `(if` $\mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2$`)` | `(if` $\mathrm{E}_0\ \mathrm{E}_1$`)`
  | `(set!` I E`)`

7.2.2. Domaines

$\alpha \in \mathtt{L}$		emplacements
$\nu \in \mathtt{N}$		entiers naturels
$\mathtt{T}$	$= \{false,\ true\}$	booléens
$\mathtt{Q}$		symboles
$\mathtt{H}$		caractères
$\mathtt{R}$		nombres
$\mathtt{E_p}$	$= \mathtt{L} \times \mathtt{L} \times \mathtt{T}$	doublets
$\mathtt{E_v}$	$= \mathtt{L}^* \times \mathtt{T}$	vecteurs
$\mathtt{E_s}$	$= \mathtt{L}^* \times \mathtt{T}$	chaînes
$\mathtt{M}$	$= \{false,\ true,\ null,\ undefined,\ unspecified\}$	
$\phi \in \mathtt{F}$	$= \mathtt{L} \times (\mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C})$	valeurs procédurales
$\epsilon \in \mathtt{E}$	$= \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E_p} + \mathtt{E_v} + \mathtt{E_s} + \mathtt{M} + \mathtt{F}$	valeurs d'expressions
$\sigma \in \mathtt{S}$	$= \mathtt{L} \rightarrow (\mathtt{E} \times \mathtt{T})$	mémoires
$\rho \in \mathtt{U}$	$= \mathrm{Ide} \rightarrow \mathtt{L}$	environnements
$\theta \in \mathtt{C}$	$= \mathtt{S} \rightarrow \mathtt{A}$	continuations de commandes
$\kappa \in \mathtt{K}$	$= \mathtt{E}^* \rightarrow \mathtt{C}$	continuations d'expressions
$\mathtt{A}$		réponses
$\mathtt{X}$		erreurs

7.2.3. Fonctions sémantiques

$\mathcal{K} : \mathrm{Con} \rightarrow \mathtt{E}$
$\mathcal{E} : \mathrm{Exp} \rightarrow \mathtt{U} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\mathcal{E}^* : \mathrm{Exp}^* \rightarrow \mathtt{U} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\mathcal{C} : \mathrm{Com}^* \rightarrow \mathtt{U} \rightarrow \mathtt{C} \rightarrow \mathtt{C}$

Définition de $\mathcal{K}$ délibérément omise.

$$\mathcal{E}[\![\mathrm{K}]\!] = \lambda\rho\kappa\,.\,\mathit{send}\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa$$

$$\begin{array}{l}
\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\kappa\,.\,\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I})\\
\qquad\qquad\qquad (\mathit{single}(\lambda\epsilon\,.\,\epsilon = \mathit{undefined} \rightarrow\\
\qquad\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“variable indefinie”},\\
\qquad\qquad\qquad\qquad\qquad \mathit{send}\,\epsilon\,\kappa))
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(}\mathrm{E}_0\ \mathrm{E}^*\texttt{)}]\!] =\\
\quad \lambda\rho\kappa\,.\,\mathcal{E}^*(\mathit{permute}(\langle\mathrm{E}_0\rangle \mathbin{\S} \mathrm{E}^*))\\
\qquad\quad \rho\\
\qquad\quad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\,(\epsilon^* \downarrow 1)\,(\epsilon^* \dagger 1)\,\kappa)\\
\qquad\qquad\quad (\mathit{unpermute}\,\epsilon^*)))
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda}\ \texttt{(}\mathrm{I}^*\texttt{)}\ \Gamma^*\ \mathrm{E}_0\texttt{)}]\!] =\\
\quad \lambda\rho\kappa\,.\,\lambda\sigma\,.\\
\qquad \mathit{new}\,\sigma \in \mathtt{L} \rightarrow\\
\qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma \mid \mathtt{L},\\
\qquad\qquad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^* \rightarrow\\
\qquad\qquad\qquad\quad \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa'))\\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*))\\
\qquad\qquad\qquad\qquad\quad \epsilon^*,\\
\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“nb incorrect d'arguments”}\rangle\\
\qquad\qquad \texttt{in}\ \mathtt{E})\\
\qquad\qquad \kappa\\
\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L})\,\mathit{unspecified}\,\sigma),\\
\qquad\quad \mathit{wrong}\ \text{“saturation memoire”}\ \sigma
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda}\ \texttt{(}\mathrm{I}^*\ \texttt{.}\ \mathrm{I}\texttt{)}\ \Gamma^*\ \mathrm{E}_0\texttt{)}]\!] =\\
\quad \lambda\rho\kappa\,.\,\lambda\sigma\,.\\
\qquad \mathit{new}\,\sigma \in \mathtt{L} \rightarrow\\
\qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma \mid \mathtt{L},\\
\qquad\qquad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^* \rightarrow\\
\qquad\qquad\qquad\quad \mathit{tievalsrest}\\
\qquad\qquad\qquad\qquad (\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa'))\\
\qquad\qquad\qquad\qquad\qquad (\mathit{extends}\,\rho\,(\mathrm{I}^* \mathbin{\S} \langle\mathrm{I}\rangle)\,\alpha^*))\\
\qquad\qquad\qquad\qquad \epsilon^*\\
\qquad\qquad\qquad\qquad (\#\mathrm{I}^*),\\
\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“trop peu d'arguments”}\rangle\ \texttt{in}\ \mathtt{E})\\
\qquad\qquad \kappa\\
\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L})\,\mathit{unspecified}\,\sigma),\\
\qquad\quad \mathit{wrong}\ \text{“saturation memoire”}\ \sigma
\end{array}$$

$$\mathcal{E}[\![\texttt{(lambda}\ \mathrm{I}\ \Gamma^*\ \mathrm{E}_0\texttt{)}]\!] = \mathcal{E}[\![\texttt{(lambda}\ \texttt{(.}\ \mathrm{I}\texttt{)}\ \Gamma^*\ \mathrm{E}_0\texttt{)}]\!]$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\texttt{)}]\!] =\\
\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \rightarrow \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa))
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\texttt{)}]\!] =\\
\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \rightarrow \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{send}\,\mathit{unspecified}\,\kappa))
\end{array}$$

Ici et ailleurs, toute valeur d'expression autre que *undefined* peut être utilisée à la place de *unspecified*.

$$\begin{array}{l}
\mathcal{E}[\![\texttt{(set!}\ \mathrm{I}\ \mathrm{E}\texttt{)}]\!] =\\
\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(\mathit{single}(\lambda\epsilon\,.\,\mathit{assign}\,(\mathit{lookup}\,\rho\,\mathrm{I})\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \epsilon\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{send}\,\mathit{unspecified}\,\kappa)))
\end{array}$$

$$\mathcal{E}^*[\![\ ]\!] = \lambda\rho\kappa\,.\,\kappa\langle\ \rangle$$

$$\begin{array}{l}
\mathcal{E}^*[\![\mathrm{E}_0\ \mathrm{E}^*]\!] =\\
\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}(\lambda\epsilon_0\,.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\,(\lambda\epsilon^*\,.\,\kappa\,(\langle\epsilon_0\rangle \mathbin{\S} \epsilon^*))))
\end{array}$$

$$\mathcal{C}[\![\ ]\!] = \lambda\rho\theta\,.\,\theta$$

$$\mathcal{C}[\![\Gamma_0\ \Gamma^*]\!] = \lambda\rho\theta\,.\,\mathcal{E}[\![\Gamma_0]\!]\,\rho\,(\lambda\epsilon^*\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho\theta)$$

7.2.4. Fonctions auxilliaires

$$\begin{array}{l}
\mathit{lookup} : \mathtt{U} \rightarrow \mathrm{Ide} \rightarrow \mathtt{L}\\
\mathit{lookup} = \lambda\rho\mathrm{I}\,.\,\rho\mathrm{I}
\end{array}$$

$$\begin{array}{l}
\mathit{extends} : \mathtt{U} \rightarrow \mathrm{Ide}^* \rightarrow \mathtt{L}^* \rightarrow \mathtt{U}\\
\mathit{extends} =\\
\quad \lambda\rho\mathrm{I}^*\alpha^*\,.\,\#\mathrm{I}^* = 0 \rightarrow \rho,\\
\qquad\qquad\qquad \mathit{extends}\,(\rho[(\alpha^* \downarrow 1)/(\mathrm{I}^* \downarrow 1)])\,(\mathrm{I}^* \dagger 1)\,(\alpha^* \dagger 1)
\end{array}$$

$\mathit{wrong} : \mathtt{X} \rightarrow \mathtt{C}$ [dépend de l'implémentation]

$$\begin{array}{l}
\mathit{send} : \mathtt{E} \rightarrow \mathtt{K} \rightarrow \mathtt{C}\\
\mathit{send} = \lambda\epsilon\kappa\,.\,\kappa\langle\epsilon\rangle
\end{array}$$

$$\begin{array}{l}
\mathit{single} : (\mathtt{E} \rightarrow \mathtt{C}) \rightarrow \mathtt{K}\\
\mathit{single} =\\
\quad \lambda\psi\epsilon^*\,.\,\#\epsilon^* = 1 \rightarrow \psi(\epsilon^* \downarrow 1),\\
\qquad\qquad \mathit{wrong}\ \text{“nb incorrect de valeurs de retour”}
\end{array}$$

$\mathit{new} : \mathtt{S} \rightarrow (\mathtt{L} + \{\mathit{error}\})$ [dépend de l'implémentation]

$$\begin{array}{l}
\mathit{hold} : \mathtt{L} \rightarrow \mathtt{K} \rightarrow \mathtt{C}\\
\mathit{hold} = \lambda\alpha\kappa\sigma\,.\,\mathit{send}\,(\sigma\alpha \downarrow 1)\kappa\sigma
\end{array}$$

$$\begin{array}{l}
\mathit{assign} : \mathtt{L} \rightarrow \mathtt{E} \rightarrow \mathtt{C} \rightarrow \mathtt{C}\\
\mathit{assign} = \lambda\alpha\epsilon\theta\sigma\,.\,\theta(\mathit{update}\,\alpha\epsilon\sigma)
\end{array}$$

$$\begin{array}{l}
\mathit{update} : \mathtt{L} \rightarrow \mathtt{E} \rightarrow \mathtt{S} \rightarrow \mathtt{S}\\
\mathit{update} = \lambda\alpha\epsilon\sigma\,.\,\sigma[\langle\epsilon, \mathit{true}\rangle/\alpha]
\end{array}$$

$$\begin{array}{l}
\mathit{tievals} : (\mathtt{L}^* \rightarrow \mathtt{C}) \rightarrow \mathtt{E}^* \rightarrow \mathtt{C}\\
\mathit{tievals} =\\
\quad \lambda\psi\epsilon^*\sigma\,.\,\#\epsilon^* = 0 \rightarrow \psi\langle\ \rangle\sigma,\\
\qquad\qquad \mathit{new}\,\sigma \in \mathtt{L} \rightarrow \mathit{tievals}\,(\lambda\alpha^*\,.\,\psi(\langle\mathit{new}\,\sigma \mid \mathtt{L}\rangle \mathbin{\S} \alpha^*))\\
\qquad\qquad\qquad\qquad\qquad\qquad (\epsilon^* \dagger 1)\\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L})(\epsilon^* \downarrow 1)\sigma),\\
\qquad\qquad\quad \mathit{wrong}\ \text{“saturation memoire”}\,\sigma
\end{array}$$

$$\begin{array}{l}
\mathit{tievalsrest} : (\mathtt{L}^* \rightarrow \mathtt{C}) \rightarrow \mathtt{E}^* \rightarrow \mathtt{N} \rightarrow \mathtt{C}\\
\mathit{tievalsrest} =\\
\quad \lambda\psi\epsilon^*\nu\,.\,\mathit{list}\,(\mathit{dropfirst}\,\epsilon^*\nu)\\
\qquad\qquad\quad (\mathit{single}(\lambda\epsilon\,.\,\mathit{tievals}\,\psi\,((\mathit{takefirst}\,\epsilon^*\nu) \mathbin{\S} \langle\epsilon\rangle)))
\end{array}$$

$$\mathit{dropfirst} = \lambda ln\,.\,n = 0 \rightarrow l, \mathit{dropfirst}\,(l \dagger 1)(n-1)$$

$$\mathit{takefirst} = \lambda ln\,.\,n = 0 \rightarrow \langle\ \rangle, \langle l \downarrow 1\rangle \mathbin{\S} (\mathit{takefirst}\,(l \dagger 1)(n-1))$$

$$\begin{array}{l}
\mathit{truish} : \mathtt{E} \rightarrow \mathtt{T}\\
\mathit{truish} = \lambda\epsilon\,.\,\epsilon = \mathit{false} \rightarrow \mathit{false}, \mathit{true}
\end{array}$$

$\mathit{permute} : \mathrm{Exp}^* \rightarrow \mathrm{Exp}^*$ [dépend de l'implémentation]

$\mathit{unpermute} : \mathtt{E}^* \rightarrow \mathtt{E}^*$ [inverse de *permute*]

$$\begin{array}{l}
\mathit{applicate} : \mathtt{E} \rightarrow \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C}\\
\mathit{applicate} =\\
\quad \lambda\epsilon\epsilon^*\kappa\,.\,\epsilon \in \mathtt{F} \rightarrow (\epsilon \mid \mathtt{F} \downarrow 2)\epsilon^*\kappa, \mathit{wrong}\ \text{“mauvaise procédure”}
\end{array}$$

$$\begin{array}{l}
\mathit{onearg} : (\mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C}) \\
\mathit{onearg} = \\
\quad \lambda\zeta\epsilon^*\kappa \,.\, \#\epsilon^* = 1 \to \zeta(\epsilon^* \downarrow 1)\kappa, \\
\qquad\qquad \mathit{wrong}\ \text{“nb incorrect d'arguments”}
\end{array}$$

$$\begin{array}{l}
\mathit{twoarg} : (\mathtt{E} \to \mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C}) \\
\mathit{twoarg} = \\
\quad \lambda\zeta\epsilon^*\kappa \,.\, \#\epsilon^* = 2 \to \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\kappa, \\
\qquad\qquad \mathit{wrong}\ \text{“nb incorrect d'arguments”}
\end{array}$$

$$\begin{array}{l}
\mathit{list} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{list} = \\
\quad \lambda\epsilon^*\kappa \,.\, \#\epsilon^* = 0 \to \mathit{send}\,\mathit{null}\,\kappa, \\
\qquad\quad \mathit{list}(\epsilon^* \dagger 1)(\mathit{single}(\lambda\epsilon \,.\, \mathit{cons}\langle\epsilon^* \downarrow 1, \epsilon\rangle\kappa))
\end{array}$$

$$\begin{array}{l}
\mathit{cons} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{cons} = \\
\quad \mathit{twoarg}(\lambda\epsilon_1\epsilon_2\kappa\sigma \,.\, \mathit{new}\,\sigma \in \mathtt{L} \to \\
\qquad\qquad (\lambda\sigma' \,.\, \mathit{new}\,\sigma' \in \mathtt{L} \to \\
\qquad\qquad\qquad \mathit{send}(\langle \mathit{new}\,\sigma \mid \mathtt{L}, \mathit{new}\,\sigma' \mid \mathtt{L}, \mathit{true}\rangle \\
\qquad\qquad\qquad\quad \text{in } \mathtt{E}) \\
\qquad\qquad\qquad\quad \kappa \\
\qquad\qquad\qquad\quad (\mathit{update}(\mathit{new}\,\sigma' \mid \mathtt{L})\epsilon_2\sigma'), \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“saturation memoire”}\,\sigma') \\
\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L})\epsilon_1\sigma), \\
\qquad\qquad \mathit{wrong}\ \text{“saturation memoire”}\,\sigma)
\end{array}$$

$$\begin{array}{l}
\mathit{less} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{less} = \\
\quad \mathit{twoarg}(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to \\
\qquad\qquad \mathit{send}(\epsilon_1 \mid \mathtt{R} < \epsilon_2 \mid \mathtt{R} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\qquad \mathit{wrong}\ \text{“argument non numérique à <”})
\end{array}$$

$$\begin{array}{l}
\mathit{add} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{add} = \\
\quad \mathit{twoarg}(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to \\
\qquad\qquad \mathit{send}((\epsilon_1 \mid \mathtt{R} + \epsilon_2 \mid \mathtt{R}) \text{ in } \mathtt{E})\kappa, \\
\qquad\qquad \mathit{wrong}\ \text{“argument non numérique à +”})
\end{array}$$

$$\begin{array}{l}
\mathit{car} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{car} = \\
\quad \mathit{onearg}(\lambda\epsilon\kappa \,.\, \epsilon \in \mathtt{E}_\mathrm{p} \to \mathit{hold}(\epsilon \mid \mathtt{E}_\mathrm{p} \downarrow 1)\kappa, \\
\qquad\qquad \mathit{wrong}\ \text{“}\mathtt{car}\text{ attendait un doublet”})
\end{array}$$

$$\mathit{cdr} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \qquad [\text{similaire à } \mathit{car}]$$

$$\begin{array}{l}
\mathit{setcar} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{setcar} = \\
\quad \mathit{twoarg}(\lambda\epsilon_1\epsilon_2\kappa \,.\, \epsilon_1 \in \mathtt{E}_\mathrm{p} \to \\
\qquad\qquad (\epsilon_1 \mid \mathtt{E}_\mathrm{p} \downarrow 3) \to \mathit{assign}(\epsilon_1 \mid \mathtt{E}_\mathrm{p} \downarrow 1) \\
\qquad\qquad\qquad\qquad \epsilon_2 \\
\qquad\qquad\qquad\qquad (\mathit{send}\,\mathit{unspecified}\,\kappa), \\
\qquad\qquad \mathit{wrong}\ \text{“argument non mutable à }\mathtt{set\text{-}car!}\text{”}, \\
\qquad\qquad \mathit{wrong}\ \text{“}\mathtt{set\text{-}car!}\text{ attendait un doublet”})
\end{array}$$

$$\begin{array}{l}
\mathit{eqv} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{eqv} = \\
\quad \mathit{twoarg}(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{M} \wedge \epsilon_2 \in \mathtt{M}) \to \\
\qquad\qquad \mathit{send}(\epsilon_1 \mid \mathtt{M} = \epsilon_2 \mid \mathtt{M} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\quad (\epsilon_1 \in \mathtt{Q} \wedge \epsilon_2 \in \mathtt{Q}) \to \\
\qquad\qquad \mathit{send}(\epsilon_1 \mid \mathtt{Q} = \epsilon_2 \mid \mathtt{Q} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\quad (\epsilon_1 \in \mathtt{H} \wedge \epsilon_2 \in \mathtt{H}) \to \\
\qquad\qquad \mathit{send}(\epsilon_1 \mid \mathtt{H} = \epsilon_2 \mid \mathtt{H} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\quad (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to \\
\qquad\qquad \mathit{send}(\epsilon_1 \mid \mathtt{R} = \epsilon_2 \mid \mathtt{R} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\quad (\epsilon_1 \in \mathtt{E}_\mathrm{p} \wedge \epsilon_2 \in \mathtt{E}_\mathrm{p}) \to \\
\qquad\qquad \mathit{send}((\lambda p_1 p_2 \,.\, ((p_1 \downarrow 1) = (p_2 \downarrow 1) \wedge \\
\qquad\qquad\qquad\qquad (p_1 \downarrow 2) = (p_2 \downarrow 2)) \to \mathit{true}, \\
\qquad\qquad\qquad\qquad \mathit{false}) \\
\qquad\qquad\qquad (\epsilon_1 \mid \mathtt{E}_\mathrm{p}) \\
\qquad\qquad\qquad (\epsilon_2 \mid \mathtt{E}_\mathrm{p})) \\
\qquad\qquad\quad \kappa, \\
\qquad\quad (\epsilon_1 \in \mathtt{E}_\mathrm{v} \wedge \epsilon_2 \in \mathtt{E}_\mathrm{v}) \to \ldots, \\
\qquad\quad (\epsilon_1 \in \mathtt{E}_\mathrm{s} \wedge \epsilon_2 \in \mathtt{E}_\mathrm{s}) \to \ldots, \\
\qquad\quad (\epsilon_1 \in \mathtt{F} \wedge \epsilon_2 \in \mathtt{F}) \to \\
\qquad\qquad \mathit{send}((\epsilon_1 \mid \mathtt{F} \downarrow 1) = (\epsilon_2 \mid \mathtt{F} \downarrow 1) \to \mathit{true}, \mathit{false}) \\
\qquad\qquad\quad \kappa, \\
\qquad\qquad \mathit{send}\ \mathit{false}\ \kappa)
\end{array}$$

$$\begin{array}{l}
\mathit{apply} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{apply} = \\
\quad \mathit{twoarg}(\lambda\epsilon_1\epsilon_2\kappa \,.\, \epsilon_1 \in \mathtt{F} \to \mathit{valueslist}\langle\epsilon_2\rangle(\lambda\epsilon^* \,.\, \mathit{applicate}\,\epsilon_1\epsilon^*\kappa), \\
\qquad\qquad \mathit{wrong}\ \text{“mauvais arguments pour }\mathtt{apply}\text{”})
\end{array}$$

$$\begin{array}{l}
\mathit{valueslist} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{valueslist} = \\
\quad \mathit{onearg}(\lambda\epsilon\kappa \,.\, \epsilon \in \mathtt{E}_\mathrm{p} \to \\
\qquad\qquad \mathit{cdr}\langle\epsilon\rangle \\
\qquad\qquad\quad (\lambda\epsilon^* \,.\, \mathit{valueslist} \\
\qquad\qquad\qquad \epsilon^* \\
\qquad\qquad\qquad (\lambda\epsilon^* \,.\, \mathit{car}\langle\epsilon\rangle(\mathit{single}(\lambda\epsilon \,.\, \kappa(\langle\epsilon\rangle \mathbin{\S} \epsilon^*))))), \\
\qquad\quad \epsilon = \mathit{null} \to \kappa\langle\,\rangle, \\
\qquad\qquad \mathit{wrong}\ \text{“}\mathtt{values\text{-}list}\text{ attendait une liste”})
\end{array}$$

$$\begin{array}{l}
\mathit{cwcc} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \qquad [\mathtt{call\text{-}with\text{-}current\text{-}continuation}] \\
\mathit{cwcc} = \\
\quad \mathit{onearg}(\lambda\epsilon\kappa \,.\, \epsilon \in \mathtt{F} \to \\
\qquad\qquad (\lambda\sigma \,.\, \mathit{new}\,\sigma \in \mathtt{L} \to \\
\qquad\qquad\qquad \mathit{applicate}\,\epsilon \\
\qquad\qquad\qquad\qquad \langle\langle \mathit{new}\,\sigma \mid \mathtt{L}, \lambda\epsilon^*\kappa' \,.\, \kappa\epsilon^*\rangle \text{ in } \mathtt{E}\rangle \\
\qquad\qquad\qquad\qquad \kappa \\
\qquad\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L}) \\
\qquad\qquad\qquad\qquad\qquad \mathit{unspecified} \\
\qquad\qquad\qquad\qquad\qquad \sigma), \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“saturation memoire”}\ \sigma), \\
\qquad\qquad \mathit{wrong}\ \text{“mauvais argument procédural”})
\end{array}$$

7.3. Expressions dérivées

Cette section donne des règles de réécriture pour les expressions dérivées. Par application de ces règles, toute expression peut être réduite à une expression sémantiquement équivalente ne contenant que les expressions primitives (littérales, variables, appel de procédure, `lambda`, `if`, `set!`).

```
(cond (⟨test⟩ ⟨sequence⟩)
      ⟨clause₂⟩ ...)
  ≡  (if ⟨test⟩
         (begin ⟨sequence⟩)
         (cond ⟨clause₂⟩ ...))

(cond (⟨test⟩)
      ⟨clause₂⟩ ...)
  ≡  (or ⟨test⟩ (cond ⟨clause₂⟩ ...))
```

```
(cond (⟨test⟩ => ⟨recipient⟩)
      ⟨clause2⟩ ...)
   ≡  (let ((test-result ⟨test⟩)
            (thunk2 (lambda () ⟨recipient⟩))
            (thunk3 (lambda () (cond ⟨clause2⟩ ...))))
        (if test-result
            ((thunk2) test-result)
            (thunk3)))

(cond (else ⟨sequence⟩))
   ≡  (begin ⟨sequence⟩)

(cond)
   ≡  ⟨une expression retournant une valeur non spécifiée⟩

(case ⟨key⟩
  ((d1 ...) ⟨sequence⟩)
  ...)
   ≡  (let ((key ⟨key⟩)
            (thunk1 (lambda () ⟨sequence⟩))
            ...)
        (cond ((⟨memv⟩ key '(d1 ...)) (thunk1))
              ...))

(case ⟨key⟩
  ((d1 ...) ⟨sequence⟩)
  ...
  (else f1 f2 ...))
   ≡  (let ((key ⟨key⟩)
            (thunk1 (lambda () ⟨sequence⟩))
            ...
            (elsethunk (lambda () f1 f2 ...)))
        (cond ((⟨memv⟩ key '(d1 ...)) (thunk1))
              ...
              (else (elsethunk))))
```

où $\langle \text{memv} \rangle$ est une expression évaluant à la procédure `memv`.

```
(and)              ≡  #t
(and ⟨test⟩)       ≡  ⟨test⟩
(and ⟨test1⟩ ⟨test2⟩ ...)
   ≡  (let ((x ⟨test1⟩)
            (thunk (lambda () (and ⟨test2⟩ ...))))
        (if x (thunk) x))

(or)               ≡  #f
(or ⟨test⟩)        ≡  ⟨test⟩
(or ⟨test1⟩ ⟨test2⟩ ...)
   ≡  (let ((x ⟨test1⟩)
            (thunk (lambda () (or ⟨test2⟩ ...))))
        (if x x (thunk)))

(let ((⟨variable1⟩ ⟨init1⟩) ...)
  ⟨corps⟩)
   ≡  ((lambda (⟨variable1⟩ ...) ⟨corps⟩) ⟨init1⟩ ...)

(let* () ⟨corps⟩)
   ≡  ((lambda () ⟨corps⟩))
(let* ((⟨variable1⟩ ⟨init1⟩)
       (⟨variable2⟩ ⟨init2⟩)
       ...)
  ⟨corps⟩)
   ≡  (let ((⟨variable1⟩ ⟨init1⟩))
        (let* ((⟨variable2⟩ ⟨init2⟩)
               ...)
          ⟨corps⟩))

(letrec ((⟨variable1⟩ ⟨init1⟩)
         ...)
  ⟨corps⟩)
   ≡  (let ((⟨variable1⟩ ⟨undefined⟩)
            ...)
         (let ((⟨temp1⟩ ⟨init1⟩)
               ...)
           (set! ⟨variable1⟩ ⟨temp1⟩)
           ...)
         ⟨corps⟩)
```

où $\langle \text{temp}_1 \rangle$, $\langle \text{temp}_2 \rangle$, ... sont des variables, distinctes de $\langle \text{variable}_1 \rangle$, ..., qui ne sont pas libres dans les expressions $\langle \text{init} \rangle$ originales. Le second `let` de cette expansion n'est pas vraiment nécessaire, mais il permet de conserver la prorpiété que les expressions $\langle \text{init} \rangle$ sont évaluées dans un ordre arbitraire.

```
(begin ⟨sequence⟩)
   ≡  ((lambda () ⟨sequence⟩))
```

L'expansion alternative suivante pour `begin` ne se sert pas de la possibilité d'écrire plus d'une expression dans le corps d'une lambda expression. En tous cas, notez que ces règles s'appliquent seulement si $\langle \text{sequence} \rangle$ ne contient aucune définition.

```
(begin ⟨expression⟩)   ≡  ⟨expression⟩
(begin ⟨command⟩ ⟨sequence⟩)
   ≡  ((lambda (ignore thunk) (thunk))
       ⟨commande⟩
       (lambda () (begin ⟨sequence⟩)))
```

L'expansion suivante pour `do` est simplifiée par l'hypothèse qu'aucun $\langle \text{pas} \rangle$ n'est omis. Toute expression `do` dans laquelle un $\langle \text{pas} \rangle$ est omis peut être remplacée par une expression `do` équivalente dans laquelle la $\langle \text{variable} \rangle$ correspondante apparaît comme le $\langle \text{pas} \rangle$.

```
(do ((⟨variable1⟩ ⟨init1⟩ ⟨pas1⟩)
     ...)
    (⟨test⟩ ⟨sequence⟩)
  ⟨commande1⟩ ...)
   ≡  (letrec ((⟨loop⟩
                (lambda (⟨variable1⟩ ...)
                  (if ⟨test⟩
                      (begin ⟨sequence⟩)
                      (begin ⟨commande1⟩
                             ...
                             (⟨loop⟩ ⟨pas1⟩ ...))))))
        (⟨loop⟩ ⟨init1⟩ ...))
```

où $\langle \text{loop} \rangle$ est n'importe quelle variable distincte de $\langle \text{variable}_1 \rangle$, ..., et qui n'apparaît pas libre dans l'expression `do`.

```
(let ⟨variable₀⟩ ((⟨variable₁⟩ ⟨init₁⟩) ...)
  ⟨corps⟩)
   ≡  ((letrec ((⟨variable₀⟩ (lambda (⟨variable₁⟩ ...)
                                 ⟨corps⟩)))
          ⟨variable₀⟩)
       ⟨init₁⟩ ...)

(delay ⟨expression⟩)
   ≡  (⟨make-promesse⟩ (lambda () ⟨expression⟩))
```

où ⟨make-promesse⟩ est une expression évaluant à une procédure qui se comporte de manière adéquate vis-à-vis de la procédure `force` ; voir la section 6.9.

NOTES

Changement du langage

Cette section énumère les changements apportés à Scheme depuis la "Révision^3" [67] de ce rapport.

- Bien que les implémentations puissent étendre Scheme, elles doivent offrir un mode syntaxique qui n'ajoute aucun mot réservé et n'empêche aucun convention lexicale de Scheme.
- Les implémentations peuvent signaler les violations de restriction d'implémentation.
- Le fait que la liste vide compte pour vrai ou pour faux dans les expressions conditionnelles n'est plus spécifié. Il faut noter que le standard IEEE de Scheme exige que la liste vide compte pour vrai [49].
- Les ensembles définis par `boolean?`, `pair?`, `symbol?`, `number?`, `char?`, `string?`, `vector?`, et `procedure?` sont disjoints.
- Les variables liées par `lambda`, `let`, `letrec`, et `do` doivent être toutes distinctes.
- Les expressions `begin` emboîtées contenant des définitions sont traitées comme des suites de définitions.
- La procédure `eqv?` n'a plus à être vraie pour deux chaînes vides ou deux vecteurs vides.
- La syntaxe des constantes numériques a changée, et l'exactitude impliquée par chaque syntaxe a été spécifiée.
- La sémantique de plusieurs procédures numériques a été éclaircie.
- `Rationalize` est devenu restreint à deux arguments et sa spécification éclaircie.
- Les procédures `number->string` et `string->number` ont été modifiées.
- `Integer->char` exige maintenant un argument entier exact.
- Les spécification de la procédure `force` a été affaiblie. La spécification précédente n'était pas implémentable.
- Variables supprimées : `t`, `nil`.
- Procédures supprimées : `approximate`, `last-pair`.
- Procédures ajoutées : `list?`, `peek-char`.
- Syntaxes rendues essentielles : `case`, `and`, `or`, `quasiquote`.
- Procédures rendues essentielles :

`reverse`	`char-ci=?`	`make-string`
`max`	`char-ci<?`	`string-set!`
`min`	`char-ci>?`	`string-ci=?`
`modulo`	`char-ci<=?`	`string-ci<?`
`gcd`	`char-ci>=?`	`string-ci>?`
`lcm`	`char-alphabetic?`	`string-ci<=?`
`floor`	`char-numeric?`	`string-ci>=?`
`ceiling`	`char-whitespace?`	`string-append`
`truncate`	`char-lower-case?`	`open-input-file`
`round`	`char-upper-case?`	`open-output-file`
`number->string`	`char-upcase`	`close-input-port`
`string->number`	`char-downcase`	`close-output-port`

- Procédures acceptant désormais un nombre plus général d'arguments : `append`, `+`, `*`, `-` (un argument), `/` (un argument), `=`, `<`, `>`, `<=`, `>=`, `map`, `for-each`.
- Un système de macros a été ajouté en appendice à ce rapport, mais ne figure pas (encore) dans cette traduction en français, peu d'implémentations offrant les "macros R^4". Référez-vous à votre dialecte local.

EXEMPLE

`Integrate-system` intègre le système

$$y_k' = f_k(y_1, y_2, \ldots, y_n),\ k = 1, \ldots, n$$

d'équations différentielles par la méthode de Runge-Kutta.

Le paramètre `system-derivative` est une fonction qui prend un état du système (un vecteur de valeurs pour les variables d'état $y_1, \ldots, y_n$) et produit une dérivée du système (les valeurs $y_1', \ldots, y_n'$). Le paramètre `initial-state` fournit un état initial du système, et `h` est un essai initial pour la longueur du pas d'intégration.

La valeur retournée par `integrate-system` est un flot infini d'états du système.

```
(define integrate-system
  (lambda (system-derivative initial-state h)
    (let ((next (runge-kutta-4 system-derivative h)))
      (letrec ((states
                (cons initial-state
                      (delay (map-streams next
                                          states)))))
        states))))
```

`Runge-Kutta-4` prend une fonction `f`, qui produit une dérivée du système à partir d'un état du système. `Runge-Kutta-4` produit une fonction qui prend un état du système et produit un nouvel état du système.

```
(define runge-kutta-4
  (lambda (f h)
    (let ((*h (scale-vector h))
          (*2 (scale-vector 2))
          (*1/2 (scale-vector (/ 1 2)))
          (*1/6 (scale-vector (/ 1 6))))
      (lambda (y)
        ;; y is a system state
        (let* ((k0 (*h (f y)))
               (k1 (*h (f (add-vectors y (*1/2 k0)))))
               (k2 (*h (f (add-vectors y (*1/2 k1)))))
               (k3 (*h (f (add-vectors y k2)))))
          (add-vectors y
            (*1/6 (add-vectors k0
                               (*2 k1)
                               (*2 k2)
                               k3))))))))

(define elementwise
  (lambda (f)
    (lambda vectors
      (generate-vector
        (vector-length (car vectors))
        (lambda (i)
          (apply f
                 (map (lambda (v) (vector-ref  v i))
                      vectors)))))))

(define generate-vector
  (lambda (size proc)
    (let ((ans (make-vector size)))
      (letrec ((loop
                (lambda (i)
                  (cond ((= i size) ans)
                        (else
                         (vector-set! ans i (proc i))
                         (loop (+ i 1)))))))
        (loop 0)))))

(define add-vectors (elementwise +))

(define scale-vector
  (lambda (s)
    (elementwise (lambda (x) (* x s)))))
```

`Map-streams` est analogue à `map`: elle applique son premier argument (une procédure) à tous les arguments de son second argument (un flot).

```
(define map-streams
  (lambda (f s)
    (cons (f (head s))
          (delay (map-streams f (tail s))))))
```

Les flots infinis sont implémentés par des doublets dont le car contient le premier élément du flot et dont le cdr contient une promesse de retourner le reste du flot.

```
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))
```

Ce qui suit illustre l'utilisation de `integrate-system` en intégrant le système

$$C\frac{dv_C}{dt} = -i_L - \frac{v_C}{R}$$

$$L\frac{di_L}{dt} = v_C$$

qui modélise un oscillateur amorti.

```
(define damped-oscillator
  (lambda (R L C)
    (lambda (state)
      (let ((Vc (vector-ref state 0))
            (Il (vector-ref state 1)))
        (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
                (/ Vc L))))))

(define the-states
  (integrate-system
     (damped-oscillator 10000 1000 .001)
     '#(1 0)
     .01))
```

BIBLIOGRAPHIE ET RÉFÉRENCES

[1] Harold Abelson and Gerald Jay Sussman. Lisp: a language for stratified design. *BYTE* 13(2):207–218, February 1988.

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

[3] Norman Adams and Jonathan Rees. Object-oriented programming in Scheme. In *Proceedings of the 1988 Conference on Lisp and Functional Programming*, pages 277–288, August 1988.

[4] David H. Bartley and John C. Jensen. The implementation of PC Scheme. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 86–93.

[5] John Batali, Edmund Goodhue, Chris Hanson, Howie Shrobe, Richard M. Stallman, and Gerald Jay Sussman. The Scheme-81 architecture—system and chip. In *Proceedings, Conference on Advanced Research in VLSI*, pages 69–77. Paul Penfield, Jr., editor. Artech House, 610 Washington Street, Dedham MA, 1982.

[6] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 86–95.

[7] William Clinger. The Scheme 311 compiler: an exercise in denotational semantics. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364.

[8] William Clinger, editor. The revised revised report on Scheme, or an uncommon Lisp. MIT Artificial Intelligence Memo 848, August 1985. Also published as Computer Science Department Technical Report 174, Indiana University, June 1985.

[9] William Clinger. Semantics of Scheme. *BYTE* 13(2):221–227, February 1988.

[10] William Clinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 92–101. Proceedings published as *SIGPLAN Notices* 25(6), June 1990.

[11] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In *Algebraic Methods in Semantics*, pages 237–250. J. Reynolds, M. Nivat, editor. Cambridge University Press, 1985.

[12] William Clinger, Anne Hartheimer, and Eric Ost. Implementation strategies for continuations. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 124–131.

[13] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pages 155–162.

[14] Pavel Curtis and James Rauen. A module system for Scheme. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

[15] R. Kent Dybvig. *The Scheme Programming Language.* Prentice-Hall, 1987.

[16] R. Kent Dybvig and Daniel P. Friedman and Christopher T. Haynes. Expansion-passing style: a general macro mechanism. *Lisp and Symbolic Computation* 1(1):53–76, June 1988.

[17] R. Kent Dybvig and Robert Hieb. A variable-arity procedural interface. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 106–115.

[18] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Journal of Computer Languages* 14(2), pages 109–123, 1989.

[19] R. Kent Dybvig and Robert Hieb. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Notices Symposium on Principles and Practice of Parallel Programming*, pages 128–136, March 1990.

[20] Michael A. Eisenberg. Bochser: an integrated Scheme programming system. MIT Laboratory for Computer Science Technical Report 349, October 1985.

[21] Michael Eisenberg. Harold Abelson, editor. *Programming In Scheme.* Scientific Press, Redwood City, California, 1988.

[22] Michael Eisenberg, with William Clinger and Anne Hartheimer. Harold Abelson, editor. *Programming In MacScheme.* Scientific Press, San Francisco, 1990.

[23] Marc Feeley. Deux approches à l'implantation du language Scheme. M.Sc. thesis, Département d'Informatique et de Recherche Opérationelle, University of Montreal, May 1986.

[24] Marc Feeley and Guy LaPalme. Using closures for code generation. *Journal of Computer Languages* 12(1):47–66, 1987.

[25] Marc Feeley and James Miller. A parallel virtual machine for efficient Scheme compilation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

[26] Matthias Felleisen. Reflections on Landin's J-Operator: a partly historical note. *Journal of Computer Languages* 12(3/4):197–207, 1987.

[27] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, pages 193–219, August 1986.

[28] Matthias Felleisen and Daniel P. Friedman. A closer look at export and import statements. *Journal of Computer Languages* 11(1):29–37, 1986.

[29] Matthias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 314–345, January 1987.

[30] Matthias Felleisen and Daniel P. Friedman. A reduction semantics for imperative higher-order languages. In *Lecture Notes in Computer Science, Parallel Architectures and Languages Europe* 259:206–223, 1987. De Bakker, Nijman and Treleaven, editors. Springer-Verlag, Berlin.

[31] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. Reasoning with continuations. In *Proceedings of the Symposium on Logic in Computer Science*, pages 131–141. IEEE Computer Society Press, Washington DC, 1986.

[32] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science* 52:205–237, 1987.

[33] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce Duba. Abstract continuations: a mathematical semantics for handling functional jumps. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, July 1988.

[34] Carol Fessenden, William Clinger, Daniel P. Friedman, and Christopher Haynes. Scheme 311 version 4 reference manual. Indiana University Computer Science Technical Report 137, February 1983. Superseded by [39].

[35] John Franco and Daniel P. Friedman. Towards a facility for lexically scoped, dynamic mutual recursion in Scheme. *Journal of Computer Languages* 15(1):55–64, 1990.

[36] Daniel P. Friedman and Matthias Felleisen. *Le Petit LISPien.* Masson, Paris, 1991.

[37] Daniel P. Friedman and Christopher T. Haynes. Constraining control. In *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages*, pages 245–254. ACM, January 1985.

[38] Daniel P. Friedman, Christopher T. Haynes, and Eugene Kohlbecker. Programming with continuations. In *Program Transformation and Programming Environments,* pages 263–274. P. Pepper, editor. Springer-Verlag, 1984.

[39] D. Friedman, C. Haynes, E. Kohlbecker, and M. Wand. Scheme 84 interim reference manual. Indiana University Computer Science Technical Report 153, January 1985.

[40] Daniel P. Friedman and Mitchell Wand. Reification: reflection without metaphysics. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 348–355.

[41] Christopher T. Haynes. Logic continuations. In *Proceedings of the Third International Conference on Logic Programming,* pages 671–685. Springer-Verlag, July 1986.

[42] Christopher T. Haynes and Daniel P. Friedman. Engines build process abstractions. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming,* pages 18–24.

[43] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Journal of Computer Languages* 12(2):109–121, 1987.

[44] Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems* 9(4):582–598, October 1987.

[45] Christopher T. Haynes and Daniel P. Friedman and Mitchell Wand. Obtaining coroutines with continuations. *Journal of Computer Languages* 11(3/4):143–153, 1986.

[46] Peter Henderson. Functional geometry. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 179–187.

[47] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, June 1990. Proceedings published as *SIGPLAN Notices* 25(6), June 1990.

[48] *IEEE Standard 754-1985. IEEE Standard for Binary Floating-Point Arithmetic.* IEEE, New York, 1985.

[49] *IEEE Standard 1178-1990. IEEE Standard for the Scheme Programming Language.* IEEE, New York, 1991.

[50] Eugene Edmund Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, August 1986.

[51] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161.

[52] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233. ACM, June 1986. Proceedings published as *SIGPLAN Notices* 21(7), July 1986.

[53] David Kranz. *Orbit: An optimizing compiler for Scheme.* PhD thesis, Yale University, 1988.

[54] Peter Landin. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2):89–101, February 1965.

[55] Drew McDermott. An efficient environment allocation scheme in an interpreter for a lexically-scoped lisp. In *Conference Record of the 1980 Lisp Conference,* pages 154–162. Proceedings reprinted by ACM.

[56] MIT Department of Electrical Engineering and Computer Science. Scheme manual, seventh edition. September 1984.

[57] Steven S. Muchnick and Uwe F. Pleban. A semantic comparison of Lisp and Scheme. In *Conference Record of the 1980 Lisp Conference*, pages 56–64. Proceedings reprinted by ACM.

[58] Peter Naur et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM* 6(1):1–17, January 1963.

[59] Paul Penfield, Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings,* pages 248–256. ACM SIGAPL, San Francisco, September 1981. Proceedings published as *APL Quote Quad* 12(1), ACM, September 1981.

[60] Kent M. Pitman. Exceptional situations in Lisp. MIT Artificial Intelligence Laboratory Working Paper 268, February 1985.

[61] Kent M. Pitman. The revised MacLisp manual (Saturday evening edition). MIT Laboratory for Computer Science Technical Report 295, May 1983.

[62] Kent M. Pitman. Special forms in Lisp. In *Conference Record of the 1980 Lisp Conference*, pages 179–187. Proceedings reprinted by ACM.

[63] Uwe F. Pleban. *A Denotational Approach to Flow Analysis and Optimization of Scheme, A Dialect of Lisp.* PhD thesis, University of Kansas, 1980.

[64] Jonathan A. Rees. *Modular Macros.* M.S. thesis, MIT, May 1989.

[65] Jonathan A. Rees and Norman I. Adams IV. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122.

[66] Jonathan A. Rees, Norman I. Adams IV, and James R. Meehan. The T manual, fourth edition. Yale University Computer Science Department, January 1984.

[67] Jonathan Rees and William Clinger, editors. The revised^3 report on the algorithmic language Scheme. In *ACM SIGPLAN Notices* 21(12), pages 37–79, December 1986.

[68] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[69] Guillermo J. Rozas. Liar, an Algol-like compiler for Scheme. S. B. thesis, MIT Department of Electrical Engineering and Computer Science, January 1984.

[70] Olin Shivers. Control flow analysis in Scheme. *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 164–174. Proceedings published as *SIGPLAN Notices* 23(7), July 1988.

[71] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation* 3(1):67–99, January 1990.

[72] Brian C. Smith. Reflection and semantics in a procedural language. MIT Laboratory for Computer Science Technical Report 272, January 1982.

[73] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming.* MIT Press and McGraw-Hill, 1989.

[74] Amitabh Srivastava, Don Oxley, and Aditya Srivastava. An(other) integration of logic and functional programming. In *Proceedings of the Symposium on Logic Programming*, pages 254–260. IEEE, 1985.

[75] Richard M. Stallman. Phantom stacks—if you look too hard, they aren't there. MIT Artificial Intelligence Memo 556, July 1980.

[76] Guy Lewis Steele Jr. Lambda, the ultimate declarative. MIT Artificial Intelligence Memo 379, November 1976.

[77] Guy Lewis Steele Jr. Debunking the "expensive procedure call" myth, or procedure call implementations considered harmful, or lambda, the ultimate GOTO. In *ACM Conference Proceedings*, pages 153–162. ACM, 1977.

[78] Guy Lewis Steele Jr. Macaroni is better than spaghetti. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, pages 60–66. These proceedings were published as a special joint issue of *SIGPLAN Notices* 12(8) and *SIGART Newsletter* 64, August 1977.

[79] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[80] Guy Lewis Steele Jr. Compiler optimization based on viewing LAMBDA as RENAME + GOTO. In *AI: An MIT Perspective.* Patrick Henry Winston Richard Henry Brown, editor. MIT Press, 1980.

[81] Guy Lewis Steele Jr. An overview of Common Lisp. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 98–107.

[82] Guy Lewis Steele Jr. *Common Lisp: The Language.* Digital Press, Burlington MA, 1984.

[83] Guy Lewis Steele Jr. and Gerald Jay Sussman. Lambda, the ultimate imperative. MIT Artificial Intelligence Memo 353, March 1976.

[84] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, January 1978.

[85] Guy Lewis Steele Jr. and Gerald Jay Sussman. The art of the interpreter, or the modularity complex (parts zero, one, and two). MIT Artificial Intelligence Memo 453, May 1978.

[86] Guy Lewis Steele Jr. and Gerald Jay Sussman. Design of a Lisp-based processor. *Communications of the ACM* 23(11):628–645, November 1980.

[87] Guy Lewis Steele Jr. and Gerald Jay Sussman. The dream of a lifetime: a lazy variable extent mechanism. In *Conference Record of the 1980 Lisp Conference*, pages 163–172. Proceedings reprinted by ACM.

[88] Guy Lewis Steele Jr. and Jon L White. How to print floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 112–126. Proceedings published as *SIGPLAN Notices* 25(6), June 1990.

[89] Gerald Jay Sussman. Lisp, programming and implementation. In *Functional Programming and its Applications.* Darlington, Henderson, Turner, editor. Cambridge University Press, 1982.

[90] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349, December 1975.

[91] Gerald Jay Sussman, Jack Holloway, Guy Lewis Steele Jr., and Alan Bell. Scheme-79—Lisp on a chip. *IEEE Computer* 14(7):10–21, July 1981.

[92] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, 1977.

[93] Texas Instruments, Inc. *TI Scheme Language Reference Manual.* Preliminary version 1.0, November 1985.

[94] Steven R. Vegdahl and Uwe F. Pleban. The runtime environment for Screme, a Scheme implementation on the 88000. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 172–182, April 1989.

[95] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM* 27(1):174–180, 1978.

[96] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28. Proceedings available from ACM.

[97] Mitchell Wand. *Finding the source of type errors.* In *Conference Record of the Thirteenth Annual Symposium on Principles of Programming Languages*, pages 38–43, 1986.

[98] Mitchell Wand. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *Proceedings of the 1986 ACM Symposium on LISP and Functional Programming*, pages 298–307, August 1986.

[99] Mitchell Wand and Daniel P. Friedman. Compiling lambda expressions using continuations and factorizations. *Journal of Computer Languages* 3:241–263, 1978.

[100] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *Meta-Level Architectures and Reflection*, pages 111–134. P. Maes and D. Nardi, editor. Elsevier Sci. Publishers B.V. (North Holland), 1988.

INDEX ALPHABÉTIQUE DES DÉFINITIONS DE CONCEPTS, MOTS-CLÉ, ET PROCÉDURES

L'entrée principale de chaque terme, procédure, ou mot-clé est listée en premier, séparée des autres entrées par un point-virgule.