Cours/LiSP/4

Principes d'implantation de Scheme et Lisp

Christian Queinnec

PARACAMPLUS

Du même éditeur :

UPMC/LI101 : annales corrigées

UPMC/LI102 (épuisé) : éléments de cours

UPMC/LI362 : annales corrigées

Missions de Paracamplus :

Paracamplus a pour vocation de procurer, à tous les étudiants en université, des ouvrages (annales corrigées, compléments de cours) et des environnements informatiques de travail, bref, des ressources pédagogiques propres à permettre, accélérer et entretenir l'apprentissage des connaissances enseignées.

Les ouvrages des éditions Paracamplus bénéficient d'une conception nous permettant de vous les proposer à des coûts très abordables.

Découvrez-nous plus avant sur notre site **www.paracamplus.com**.

SARL Paracamplus
7, rue Viollet-le-Duc, 75009 Paris – France
ISBN 978-2-916466-03-3

Table des matières

Avis au lecteur

Quoique la littérature sur Lisp soit abondante et déjà largement accessible au public français, un ouvrage pourtant manquait que vous tenez entre les mains ! Le substrat mathématico-logique sur lequel est fondé Lisp/Scheme exige que ses utilisateurs modernes ne répugnent point à lire des programmes usant, voire abusant, de haute technologie, c'est-à-dire de fonctionnelles d'ordre très largement supérieur et/ou de continuations. Les concepts de demain sont élaborés sur ces bases ; les méconnaître rendrait l'avenir plus inquiétant[1]. Expliquer ces entités, leur genèse, leurs variantes est un point que ce livre tente de couvrir en moult détails. Le folklore nous enseigne que si le lispien moyen connaît la valeur de chacune des constructions qu'il emploie, il en ignore généralement le coût. Cet ouvrage comble aussi cette lacune par une étude approfondie de la sémantique et de l'implantation canonique des différents traits de Lisp tels que sédimentés par plus de trente ans d'histoire.

Lisp est un langage plaisant en lequel de nombreux problèmes fondamentaux, non triviaux, peuvent être simplement étudiés. Lisp est, avec ML qui privilégie le typage et l'absence d'effets de bord, le plus représentatif des langages applicatifs. Les concepts qu'illustre cette classe de langages doivent absolument être maîtrisés par les étudiants et les informaticiens d'aujourd'hui ou de demain[2]. Fondés sur la notion de fonction que des siècles de mathématique ont lentement mûrie, les langages applicatifs sont omniprésents en informatique sous des formes variées (*shell* et flots d'UN⋆X, langage d'extension d'édition de texte en Emacs ou de dessin assisté sous AutoCAD ...). Ne pas reconnaître ces modèles, c'est méconnaître la composabilité de leurs éléments primitifs et donc se limiter à un mot à mot pénible pour l'écriture de programmes sans architecture.

Public

Cet ouvrage s'adresse :

[1] La première édition de ce livre date de 1994 et les adjonctions à cette préface apparaissent dans ces notes de bas de page. Depuis 1994, Perl, Python, C# se sont mis à proposer des fermetures, les classes internes de Java sont également des sortes de fermetures. Le terme de continuation est apparu dans le contexte des systèmes d'exploitation [DBRD91]. Le moteur Rhino implantant JavaScript procure également des continuations.

[2] Les modèles d'exécution des langages dynamiques tels que JavaScript, Perl, Python, Ruby, Scheme mais aussi Java sont macroscopiquement assez proches.

– aux étudiants de master qui ont à étudier l'implantation de langages (applicatifs ou pas) par interprétation et/ou compilation ;
– à tous les programmeurs (et computistes) en Lisp ou Scheme qui pourront comprendre finement le coût ou la subtilité des constructions qu'ils emploient afin de s'améliorer dans leur art et produire des programmes plus efficaces et plus portables ;
– aux nombreux amoureux des langages de programmation qui y trouveront de multiples pistes de réflexion sur leur langage favori.

Philosophie

Le présent ouvrage s'inspire de cours professés en DEA ITCP, en DESS GLA de l'université Pierre et Marie Curie (Paris 6) ainsi qu'en enseignement d'approfondissement à l'École Polytechnique. Il prend typiquement la suite d'un cours d'initiation à un langage applicatif (Lisp, Scheme, ML ...), qui culmine généralement par une description du langage en lui-même. L'ouvrage a pour ambition de très largement couvrir la sémantique et l'implantation d'interprètes ou de compilateurs pour des langages applicatifs. Il présente en effet pas moins de douze interprètes et deux compilateurs (vers du code-octet et vers le langage C) sans oublier un système d'objets (dérivé du populaire MEROON). Contrairement à de nombreux livres, cet ouvrage ne néglige aucunement des aspects aussi importants que la réflexivité, l'introspection, l'évaluation dynamique sans oublier les macros, tous phénomènes essentiels de la famille des dialectes de Lisp.

La facture de ce livre s'inspire principalement de deux ouvrages : *Anatomy of Lisp* [All78] qui couvrait l'implantation de Lisp des années soixante-dix et *Operating System Design : The XINU Approach* [Com84] dont les programmes ne celaient aucun détail et, par là même, s'acquéraient la totale confiance des lecteurs. Nous avons donc souhaité réaliser un ouvrage précis dont le thème central est la sémantique des langages applicatifs et de Scheme en particulier. Au travers de multiples implantations explorant différents aspects, la construction d'un tel système est entièrement explicitée. La plupart des grandes variations schismatiques qu'ont pu connaître les langages applicatifs est analysée, démontée, implantée et comparée. Tous les détails, même les plus infimes, sont divulgués. Cela permet au lecteur de ne point se bloquer par manque d'informations, le conforte dans l'idée que l'on ne lui cache rien, le met, enfin, à pied d'œuvre pour expérimenter par lui-même les concepts manipulés puisque tous les programmes de ce livre peuvent être récupérés électroniquement.

Structure

L'ouvrage est formé de deux parties. La première est une progression partant de l'implantation d'un interprète naïf pour Lisp et débouchant sur la sémantique de Scheme. Le fil conducteur est le besoin de préciser, d'affiner et de définir successivement les espaces de noms (Lisp_1, Lisp_2 ...), les continuations (et les multiples formes de contrôle associées), les affectations et les écritures dans les structures de données.

Chapitre	Signature
1	`(eval exp env)`
2	`(eval exp env fenv)`
	`(eval exp env fenv denv)`
	`(eval exp env denv)`
3	`(eval exp env cont)`
4	`(eval e r s k)`
5	`((meaning e) r s k)`
6	`((meaning e sr) r k)`
	`((meaning e sr tail?) k)`
	`((meaning e sr tail?))`
7	`(run (meaning e sr tail?))`
10	`(->C (meaning e sr))`

FIG. 1 – Signatures (approximatives) des interprètes et compilateurs

Cette lente augmentation du langage défini s'accompagne d'une régression du langage de définition qui s'appauvrit jusqu'à ne plus être qu'une sorte de λ-calcul. La description obtenue à ce stade est alors convertie en son équivalent dénotationnel par une simple curryfication. La nécessité de ce parcours quasi initiatique de recherche de précision dans le langage décrit nous semble, après plus de six ans d'enseignement, une très bonne approche de la sémantique dénotationnelle qui surgit ainsi justifiée et non parachutée.

La seconde partie est un voyage inverse. Partant de la sémantique dénotationnelle et en quête d'efficacité, nous abordons l'interprétation rapide (par prétraitement des parties statiques), puis ce conditionnement (cette précompilation) est mis en œuvre pour un compilateur de code-octet. Sont alors traitées, dans ce cadre séparant bien la préparation des programmes de leur exécution : l'évaluation dynamique (`eval`), les aspects réflexifs (environnements de première classe, interprète auto-interprétable et tour d'interprètes) et les sémantiquement redoutables macros. Pour sacrifier à la mode, un second compilateur vers le langage C est présenté. Enfin le livre se clôt par l'implantation d'un système d'objets, objets qui sont justement utilisés pour décrire finement l'implantation de certains interprètes et de certains compilateurs.

La répétition est mère de la pédagogie. La multiplicité des interprètes présentés, écrits dans des styles volontairement variés (naïf, par objets, par fermetures, dénotationnel ...) couvre l'essentiel des techniques utilisées pour l'implantation des langages applicatifs et permet d'exciter la réflexion des lecteurs quant aux différences qu'ils présentent entre eux. C'est la (re)connaissance de ces différences, esquissées en figure 1, qui fonde la compréhension intime de ce que sont un langage et son implantation : Lisp n'est pas une de ces implantations en particulier, c'est une famille de dialectes qui tous composent leur propre cocktail à partir des traits présentés.

De manière générale, les chapitres forment des entités plutôt autonomes d'environ quarante pages et sont accompagnés d'exercices, dont les corrigés sont donnés en fin d'ouvrage. La bibliographie contient non seulement des références historiques (permettant d'apprécier l'évolution de Lisp depuis 1960) mais aussi des références aux

travaux actuels de recherches.

Coda

Si ce livre se veut divertissant et instructif tout à la fois, il n'est pas nécesssairement simple à lire. Certains sujets ne dévoilent leur beauté que pressés de près et seulement si l'ardeur mise à leur siège est à la mesure de leur complexité. L'étude des langages de programmation est une ascèse qui ne se peut apprendre sans au moins quelques outils théoriques comme le λ-calcul ou la sémantique dénotationnelle. Le dessein de cet ouvrage est d'amener le lecteur à ces connaissances suivant un ordre harmonieux qui ne supprime pas pour autant tout effort de sa part.

Il va de soi que cet ouvrage nécessite quelques connaissances préalables de Lisp ou de Scheme : le lecteur doit connaître les quelque trente fonctions de base qui suffisent pour débuter et être à même de déchiffrer sans trop d'efforts des programmes récursifs. Le langage adopté pour cet ouvrage est Scheme dont un abrégé apparaît plus loin, augmenté d'une couche d'objets, nommée MEROON, qui n'est utilisée que lorsqu'il s'agira d'exhiber les problèmes de représentation et d'implantation. Tous ces programmes ont été testés et tournent sous Scheme. Ils ne devraient pas poser de problèmes de portage aux lecteurs ayant assimilé ce livre !

Il ne me reste plus qu'à remercier les organismes qui m'ont procuré les machines (Apple Mac SE30 puis Sony News 3260[3]) et les moyens qui m'ont permis de composer cet ouvrage : l'École Polytechnique, l'Institut National de Recherche en Informatique et Automatique (INRIA–Rocquencourt) et le Greco-PRC de Programmation du Centre National de la Recherche Scientifique (CNRS)[4]. Je remercie aussi tous ceux qui ont concouru à l'élaboration de ce livre par tous les moyens dont ils disposaient et qu'ils ont su me permettre d'apprécier. Que soient donc remerciés Sophie Anglade, Josy Baron, Jérôme Chailloux, Marc Feeley, Jean-Marie Geffroy, Christian Jullien, Jean-Jacques Lacrampe, Michel Lemaître, Luc Moreau, Jean-François Perrot, Daniel Ribbens, Bernard Serpette, Manuel Serrano, Pierre Weis ainsi que ma muse préférée, Claire N***. Bien entendu les erreurs, qui certainement (et bien malheureusement) subsistent, sont miennes.

Notations

Les extraits de programmes apparaissent dans `cette police de caractères qui fait irrésistiblement penser aux machines à écrire de jadis`. Il est possible que certaines parties soient en italiques pour attirer l'œil sur des variantes concentrées en cet endroit.

La notation « $\rightarrow$ » exprime la relation « a pour valeur » tandis que le signe « $\equiv$ » exprime l'équivalence, c'est-à-dire la relation « a même valeur que ». Quand l'évaluation d'une forme est détaillée, une barre verticale désigne l'environnement dans lequel

[3] et, depuis 1994, divers PC et un PowerBook.

[4] sans oublier l'UPMC (université Pierre et Marie Curie), son service de formation permanente et le LIP6 (laboratoire d'informatique de Paris 6).

l'expression immédiatement précédente doit être considérée. Toutes ces notations apparaissent dans l'exemple qui suit :

```
(let ((a (+ b 1)))
  (let ((f (lambda () a)))
    (foo (f) a) ) )|              ; foo a pour valeur la fonction
                   | b→ 3         ; de création de paires c'est-à-dire la valeur
                   | foo≡ cons    ; de la variable globale cons.
≡ (let ((f (lambda () a)))
      (foo (f) a) )|
                   | a→ 4
                   | b→ 3
                   | foo≡ cons
≡ (foo (f) a)|
             | a→ 4
             | b→ 3
             | foo≡ cons
             | f≡ (lambda () a)|
                                 a→ 4
→ (4 . 4)
```

Tous les programmes sont écrits en *broken english*, la langue des mots-clés de Lisp, y compris les messages d'erreur qui ressemblent à ceux que l'on peut aujourd'hui observer.

Nous utiliserons çà et là quelques fonctions non standard en Scheme, comme `gensym` qui permet de créer des symboles garantis nouveaux, c'est-à-dire différents de tout symbole précédemment connu. Nous utilisons aussi, au chapitre 10, `format` et `pp` pour imprimer. Ces fonctions existent dans la plupart des implantations de Lisp ou Scheme.

Certains exemples ne se peuvent comprendre que par rapport à un dialecte particulier comme COMMON LISP, Dylan, EULISP, IS-Lisp, Le-Lisp[5], Scheme, etc. Dans ce cas, le nom du dialecte est apposé en regard de l'exemple comme dans ce qui suit :

```
(defdynamic fooncall                                        IS-Lisp
  (lambda (one :rest others)
    (funcall one others) ) )
```

Afin de faciliter la traversée de ce livre et lorsque sont évoquées des discussions passées ou à venir, un signe indicateur [☞ **p.**] mentionne la page concernée. C'est aussi le cas de variantes proposées en exercices dont le numéro est evoqué comme suit [☞ **Ex.**]. Enfin un index permet de retrouver toutes les définitions de fonctions citées. [☞ **p. 473**]

Enfin nous avons gardé aux fonctions le genre féminin. Ainsi la fameuse `car` retourne-t-elle le premier élément d'une liste non vide.

Petit précis de Scheme

Ce bref résumé n'a pas l'ambition d'un livre totalement dédié à l'apprentissage du langage Scheme comme le sont excellement et en anglais [AG85, Dyb87, SF89] ou en français [Cha96, Huf96, Blo00, BDP+04]. Il n'expose que les grands traits de

[5] Le-Lisp est une marque déposée de l'Institut National de Recherche en Informatique et Automatique (INRIA).

ce dialecte dont nous userons dans toute la suite, suite qui vous conduira à un plus haut niveau de compréhension de ce même dialecte par un auto-amorçage du plus bel effet. Le document de référence est le *Revised revised revised revised revised Report on Scheme* ou R^5RS [KCR98].

Scheme permet de manipuler des symboles, des caractères, des chaînes de caractères, des listes, des nombres, des booléens, des vecteurs, des flux et des fonctions (ou procédures). Tous ces types de données ont leur prédicat associé : `symbol?`, `char?`, `string?`, `pair?`, `number?`, `boolean?`, `vector?` et `procedure?`. Ils ont aussi, quand approprié, des sélecteurs et modificateurs correspondants comme `string-ref`, `string-set!`, `vector-ref`, `vector-set!` et pour les listes : `car`, `cdr`, `set-car!` et `set-cdr!`. Les sélecteurs `car` et `cdr` peuvent être composés, ainsi `cadr` désigne le second terme d'une liste.

Ces valeurs peuvent être implicitement nommées et créées par simple mention : c'est le cas des symboles (ou identificateurs), des caractères (préfixé de `#\` comme dans `#\Z` ou `#\space`), des chaînes (lorsqu'encloses entre guillemets), des listes dûment parenthésées, des nombres, des booléens (`#t` et `#f`) et des vecteurs (avec la syntaxe `#(do re mi)`). Ces valeurs peuvent être dynamiquement construites ou converties entre elles, lorsque cela est possible, grâce notamment à `string->symbol`, `int->char`, `string`, `make-string`, `cons`, `list`, `vector` et `make-vector`.

Les entrées/sorties sont l'objet des fonctions `read` qui lit une expression, `display` qui l'imprime et `newline` qui permet de passer à la ligne.

La séquence (ou forme `begin`) permet de regrouper une suite d'expressions à évaluer séquentiellement, par exemple :

```
(begin (display 1) (display 2) (newline))
```

Plusieurs formes conditionnelles existent dont la plus simple est la forme *if—then—else—* classique notée, en Scheme, (`if` *`condition alors sinon`*). Pour traiter des aiguillages à plus de deux voies, on trouve `cond` et `case`. La forme `cond` contient une suite de clauses débutant par une expression booléenne ; celles-ci sont tour à tour évaluées jusqu'à ce qu'une retourne Vrai (c'est-à-dire non Faux, non `#f`), les formes qui suivent cette expression booléenne ayant réussi sont alors évaluées et leur résultat devient celui de la forme `cond` tout entière. Voici un exemple de forme `cond` dans laquelle figure un comportement par défaut introduit par le mot-clé `else` :

```
(cond ((eq? x 'flip) 'flop)
      ((eq? x 'flop) 'flip)
      (else (list x "neither flip nor flop")) )
```

La forme `case` a comme premier paramètre une forme qui fournit une clé que l'on recherche ensuite dans toutes les clauses qui suivent et dont chacune précise la ou les clés qui la déclenche. Lorsqu'une clé adaptée est trouvée, les formes associées sont évaluées et leur résultat devient celui de la forme `case` tout entière. Voici la forme `cond` de l'exemple précédent mise à la sauce de `case` :

```
(case x
  ((flip) 'flop)
  ((flop) 'flip)
  (else (list x "neither flip nor flop")) )
```

Les fonctions sont définies par la forme `lambda`. Juste après le mot-clé `lambda`, se situent les variables de la fonction suivies d'expressions indiquant comment se calcule celle-ci. Les variables peuvent être modifiées par affectation, notée `set!`. Les

littéraux sont introduits par `quote`. Les formes `let`, `let*` et `letrec` introduisent toutes des variables locales dont la valeur d'initialisation est diversement calculée.

La forme `define` permet de définir des valeurs nommées de tout genre. On usera de la facilité d'écriture qu'offrent les `define` internes ainsi que de la syntaxe non essentielle où le nom de la fonction à définir est stipulé par sa forme d'appel. En voici un exemple :

```
(define (rev l)
  (define nil '())
  (define (reverse l r)
     (if (pair? l) (reverse (cdr l) (cons (car l) r)) r) )
  (reverse l nil) )
```

Cet exemple pourrait aussi être écrit, sans syntaxe inessentielle :

```
(define rev
  (lambda (l)
    (letrec ((nil '())
             (reverse
               (lambda (l r)
                 (if (pair? l)
                     (reverse (cdr l) (cons (car l) r))
                     r ) ) ))
      (reverse l nil) ) ) )
```

Ici s'achèvent ces quelques rappels de Scheme.

Programmes

Les programmes figurant dans ce livre, les divers interprètes et compilateurs, le système d'objets et les tests associés sont disponibles via Internet en :

`http://www-spi.lip6.fr/~queinnec/Books/LiSP-2ndEdition.tgz`

L'adresse électronique de l'auteur est `Christian.Queinnec@lip6.fr`.

Lectures recommandées

Comme nous supposons Scheme connu, on se référera aux classiques [AG85, SF89, ML95, HKK99, FFFK03] et, en français, [Cha96, Huf96, Blo00, BDP+04] sans oublier les exercices de [MQRS99, BDP+06]. On pourra s'armer aussi, pour mieux profiter de cet ouvrage, des manuels de référence de COMMON LISP [Ste90], Dylan [App92b], EULISP [PE92], IS-Lisp [ISO94], Le-Lisp, OakLisp [LP88], Scheme [KCR98] et T [RAM84]. Enfin, on consultera [BG94] pour un large tour d'horizon sur les langages de programmation en général.

Post-préface

Cet ouvrage est une nouvelle édition corrigée du livre intitulé « Les langages Lisp » paru chez InterÉditions en 1994. Que soient remerciés tous les lecteurs ayant fait pression, tout au long de ces dernières années, pour obtenir sa ré-édition !

1
Fondements de l'interprétation

Ce chapitre présente l'interprète de base sur lequel nous fonderons la plus grande part de cet ouvrage. Résolument simple, il s'apparente à Scheme plus qu'à Lisp mais expliquera celui-ci grâce à celui-là. Seront successivement discutés : les articulations de cet interprète, le fameux couple formé d'`eval` et d'`apply`, puis les qualités attendues des environnements et des fonctions. Enfin, diverses variantes seront esquissées procurant autant de pistes que poursuivront les prochains chapitres. Ces variantes sont autant de doutes qui devraient révéler à l'intrépide lecteur les abysses qu'il côtoie. Cet interprète et ses variations sont écrits en Scheme naturel sans restriction linguistique.

La littérature sur Lisp n'a que rarement résisté au plaisir narcissique de donner la définition de Lisp en Lisp. Ce pli, qu'a marqué pour la première fois le manuel de référence de Lisp 1.5 [MAE+62], a été très largement imité ; citons seulement [Rib69], [Gre77], [Que82], [Cay83], [Cha80], [SJ93] et en langue étrangère [Rey72], [Gor75], [SS75], [All78], [McC78b], [Lak80], [Hen80], [BM84], [Cli84], [FW84], [dRS84], [AG85], [R3R86], [Mas86], [Dyb87], [WH88], [Kes88], [LF88], [Dil88], [Kam90], [BDP+04] parmi tant d'autres !

Ces évaluateurs sont très variés à la fois par le langage qu'ils définissent et par celui qu'ils utilisent pour ce faire mais surtout, par le but poursuivi : l'évaluateur proposé dans [Lak80] montre comment sont immergés des objets et concepts graphiques en Lisp ; l'évaluateur de [BM84] tient à jour un paramètre représentant la taille de l'évaluation.

Le *langage de définition* n'est pas indifférent. Que l'affectation et les opérateurs chirurgicaux (`set-car !`, `set-cdr !` etc.) soient autorisés augmente la richesse du langage de définition et permet ainsi de diminuer la taille (en nombre de lignes) des descriptions, voire de simuler précisément le *langage décrit* en termes rappelant les instructions de bas niveau de la machine (éventuellement virtuelle). Inversement, la description utilise plus de concepts. Restreindre le langage de définition complique la tâche mais diminue les risques de divergences sémantiques. Même si la taille de la description augmente, le langage décrit est plus précis et, de ce fait, mieux partagé.

La figure 1.1 classe quelques interprètes représentatifs en fonction de la complexité du langage de définition (en abscisse) et du langage défini (en ordonnée). La progression des connaissances y apparaît clairement : des problèmes de plus en plus

complexes sont attaqués avec des moyens cherchant à se restreindre. Le présent ouvrage correspond au vecteur orienté qui part d'un Lisp très riche implantant Scheme pour arriver au λ-calcul implantant un Lisp très riche.

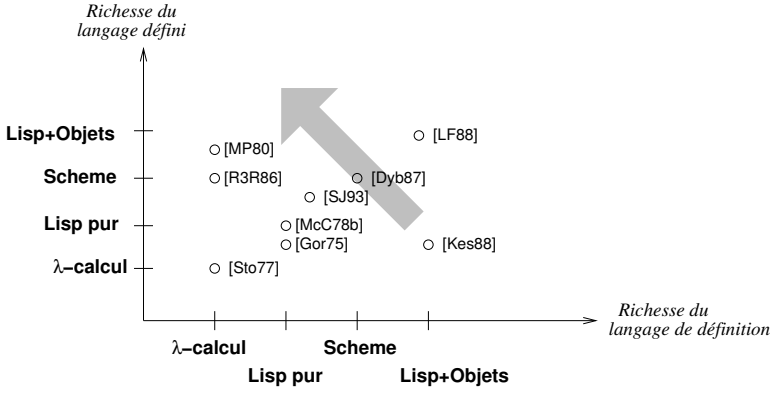


FIG. 1.1 – Classification de quelques évaluateurs

1.1 Évaluation

L'essentiel d'un interprète pour Lisp se concentre en une unique fonction autour de laquelle tournoient quelques autres plus utilitaires. Cette fonction, nommée `eval`, prend un programme en argument et délivre en sortie sa valeur.

La présence d'un évaluateur explicite est un trait caractéristique de Lisp. Son existence n'est pas due au hasard mais est plutôt le fruit d'un dessein précis. Un langage est universel s'il a la puissance d'une machine de Turing, ce qui, compte tenu de la rudimentarité de celle-ci (une mémoire de cellules binaires, une tête de lecture/écriture mobile), n'est pas difficile à obtenir ; il est probablement plus délicat de concevoir un langage utile qui ne soit pas universel.

La fameuse thèse de Church dit que toute fonction calculable peut s'écrire dans tout langage universel. Un système Lisp peut être comparé à une fonction prenant des programmes en entrée et retournant leur valeur en sortie. L'existence de tels systèmes prouve leur calculabilité : un système Lisp peut donc être écrit en un langage universel. *Ergo* la fonction `eval` peut être écrite en Lisp et, plus généralement, le comportement de Fortran peut être décrit en Fortran, etc.

Ce qui rend Lisp unique est qu'une explicitation non triviale d'`eval` est d'une taille raisonnable, de une à vingt pages environ[1] suivant le niveau de détail souhaité. Cette propriété est le résultat d'un effort de conception important pour régulariser le

[1] La définition de ce chapitre mesure environ 150 lignes tandis que celle (pédagogique) de [BDP+04] mesure (commentaires compris) 688 lignes.

langage, supprimer les cas particuliers et surtout établir une syntaxe presque abstraite tant est auguste sa simplicité.

De nombreuses et intéressantes propriétés découlent de la présence d'`eval` et de sa définissabilité en Lisp même.

- Apprendre Lisp peut s'effectuer par la lecture d'un manuel de référence (qui expose les fonctions par thèmes) ou bien par l'étude de la fonction `eval`. Le défaut de cette dernière approche est que pour lire `eval`, il faut connaître Lisp, ce qui était le résultat souhaité de l'opération et non son prérequis. En fait, il suffit de connaître le sous-ensemble de Lisp utilisé par `eval`. D'autre part, le langage que définit `eval` est plus dépouillé en ce sens qu'il ne procure que l'essence du langage réduit aux seules formes spéciales et fonctions primordiales.
  Disposer de deux approches interagissantes pour l'apprentissage de Lisp est un indéniable atout que procure Lisp.
- Que la définition d'`eval` soit disponible en Lisp même permet d'immerger simplement l'environnement de programmation au sein du langage et fournit à moindre coût : traceur, metteur au point et autre évaluateur réversible [Lie87]. En effet, l'écriture de ces outils de contrôle d'évaluation n'est qu'une décoration du code d'`eval` pour, par exemple, imprimer les appels fonctionnels, demander à l'utilisateur son indice de satisfaction afin d'éventuellement poursuivre l'évaluation, mémoriser des résultats intermédiaires, etc.
  Ces qualités ont pendant longtemps assuré la suprématie de Lisp en matière d'environnement de programmation. Encore aujourd'hui, qu'`eval` soit définissable permet que soient expérimentés facilement de nouveaux modèles de metteur au point.
- Enfin, `eval` est un outil de programmation. Cet outil est controversé puisqu'il implique au sein d'une application écrite en Lisp et usant d'`eval`, de conserver un interprète ou un compilateur entier, mais surtout de renoncer à de nombreuses optimisations. L'emploi d'`eval` n'est donc pas sans conséquence. Il se justifie dans certains cas, notamment lorsque Lisp sert à la définition et la mise en œuvre d'un langage de programmation incrémentielle.
  Outre ce coût important, la sémantique d'`eval` n'est pas claire, ce qui a justifié sa mise à l'écart de la définition officielle de Scheme dans le R3RS [R3R86] ; `eval` apparaît néanmoins dans le R5RS [KCR98]. **[☞ p. 257]**

1.2 Évaluateur fondamental

Il est possible de distinguer dans un programme les *variables libres* des *variables liées*. Une variable est *libre* lorsqu'aucune forme liante (`lambda`, `let` ...) ne la qualifie ; dans le cas contraire les variables sont dites *liées*. Comme son nom l'indique, une variable libre est affranchie de toute contrainte ; sa valeur peut être n'importe quoi. Il importe donc, pour connaître la valeur d'un fragment de programme comportant des variables libres, que soient connues les valeurs de ces mêmes variables libres. On nomme *environnement*, la structure de données associant variables et valeurs. La

fonction `evaluate`[2] est donc binaire, prend un programme accompagné d'un environnement et retourne une valeur.

```
(define (evaluate exp env) ...)
```

1.3 Évaluation des atomes

Une importante caractéristique de Lisp est que les programmes sont *représentés* par les expressions mêmes du langage. Mais comme toute représentation suppose un certain *codage*, il est nécessaire d'expliciter comment sont représentés les programmes. Les principales conventions sont qu'une variable est représentée par le symbole éponyme et qu'une application fonctionnelle est représentée par une liste dont le premier terme représente la fontion à appliquer et dont les termes suivants représentent les arguments qui seront soumis à cette fonction.

Comme tout compilateur, `evaluate` débute par une analyse syntaxique de l'expression à évaluer pour en déduire ce qu'elle représente. Le titre de cette section est donc impropre puisqu'il ne s'agit pas littéralement d'évaluer des atomes mais des programmes dont la représentation est atomique. Il importe de bien distinguer le programme de sa représentation, le message du medium. La fonction `evaluate` travaille sur la représentation, en déduit l'intention escomptée, exécute enfin ce qui est demandé.

```
(define (evaluate exp env)
  (if (atom? exp)  ;(atom? exp) ≡ (not (pair? exp))
      ...
      (case (car exp)
        ...
        (else ...) ) ) )
```

Si une expression n'est pas une liste, ce peut être un symbole ou une donnée immédiate, comme un nombre ou une chaîne de caractères, etc. Lorsque l'expression est un symbole, l'expression représente une *variable* et sa valeur est celle que lui attribue l'environnement.

```
(define (evaluate exp env)
  (if (atom? exp)
      (if (symbol? exp) (lookup exp env) exp)
      (case (car exp)
        ...
        (else ...) ) ) )
```

La fonction `lookup`, explicitée plus loin, sait retrouver la valeur d'une variable dans un environnement. Notez que la fonction `lookup` a pour signature :

`(lookup` *variable environnement*`)` → *valeur*

Il y a donc eu une conversion implicite entre symbole et variable. Une écriture plus pointilleuse aurait été à la place de `(lookup exp env)` :

```
... (lookup (symbol->variable exp) env) ...
```

[2] Afin d'aider le lecteur à ne pas confondre l'évaluateur que nous décrivons avec la fonction `eval`, le plus souvent unaire, nous nommerons `evaluate` et `invoke` les fonctions `eval` et `apply`. Ces nouveaux noms faciliteront aussi la vie du lecteur qui souhaiterait expérimenter ces programmes.

Cette dernière écriture met bien en relief que le symbole, valeur de `exp`, doit être transmué en une variable et que, loin d'être l'identité, la fonction `symbol->variable`[3] convertit une entité syntaxique (un symbole) en une entité sémantique (une variable). Une variable, en effet, n'est rien d'autre qu'un objet imaginaire auquel le langage et le programmeur attachent un certain sens mais qui, pour des raisons matérielles, n'est manipulée que par le biais d'une représentation. La représentation a été choisie pour sa légèreté : `symbol->variable` est l'identité parce que Lisp possède le concept de symbole parmi ses types de base. D'autres représentations auraient pu être adoptées : une variable pourrait apparaître sous la forme d'une suite de caractères préfixée par un signe dollar. En ce dernier cas, la fonction de conversion `symbol->variable` aurait été moins simpliste.

Si une variable est un concept imaginaire, la fonction `lookup` ne saurait l'accepter en premier argument puisqu'elle ne sait opérer que sur des objets tangibles. Il faut donc encore une fois encoder la variable en une représentation, une clé, permettant à `lookup` de retrouver sa valeur dans l'environnement. Une écriture plus précise serait donc :

```
... (lookup (variable->key (symbol->variable exp)) env) ...
```

Mais la paresse naturelle des lispiens les pousse à adopter comme clé associée à une variable le symbole de même nom. Ainsi donc, `variable->key` n'est qu'un inverse de `symbol->variable`, la composition de ces deux fonctions n'étant que l'identité.

On a coutume, lorsque l'expression est atomique (lorsqu'il ne s'agit pas d'une paire pointée) et qu'elle n'est pas un symbole, de la considérer comme la représentation d'une constante qui est sa propre valeur. Cette idempotence est qualifiée d'*autoquote*. Un objet « autoquoté » n'a pas besoin d'être « quoté » et a pour valeur lui-même, voir [Cha94] pour un exemple. Là encore, ce choix n'est pas évident pour plusieurs raisons. Tous les objets atomiques ne se dénotent pas naturellement en eux-mêmes. La valeur de la chaîne de caractères `"a?b:c"` pourrait être d'appeler le compilateur C sur cette chaîne, puis d'exécuter le programme résultant et de réinjecter les résultats en Lisp. D'autres types d'objets semblent absolument rétifs au concept d'évaluation comme, par exemple, les fonctions. La variable `car` a pour valeur la fonction qui extrait d'une paire son fils gauche (nommons la *car*), fonction dont on sait, par ailleurs, l'intérêt. Mais le programme représenté par la fonction *car* elle-même, quelle valeur a-t-il ? Évaluer une fonction est en général l'indice d'une erreur qu'il vaut mieux détecter au plus tôt. Un autre exemple est la liste vide `()` qui pourrait suggérer, de par sa forme écrite, qu'elle est une application vide, c'est-à-dire une application fonctionnelle sans argument et dont on a même oublié de mentionner la fonction. Cette syntaxe est pas définie par la norme de Scheme et n'a, par conséquent, pas de valeur[4]. Il faut donc analyser soigneusement les expressions et ne rendre *autoquote* que les seules données qui le méritent, à savoir nombres, caractères et chaînes de caractères. **[☞ p. 7]** On pourrait ainsi écrire :

[3] Je n'aime personnellement pas les noms formés comme x`->`y pour indiquer une conversion car cet ordre complique la compréhension des compositions : `(`y`->`z`(`x`->`y`...))` est moins simple que `(`z`<-`y`(`y`<-`x`...))`. En revanche, x`->`y se lit plus simplement que y`<-`x. On voit là une des difficultés auxquelles se heurtent les concepteurs de langages.

[4] Ce qui n'empêche pas certaines implantations de lui donner une valeur !

```
(define (evaluate e env)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e) (string? e) (char? e)
                 (boolean? e) (vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
      ... ) )
```

On voit dans le fragment précédent apparaître le premier cas d'erreur possible. La plupart des systèmes Lisp ont un système d'exception personnel ; aussi est-il délicat d'écrire du code portable en ce domaine. Dans une situation erronée, nous ferons appel à `wrong`[5] avec, pour premier argument, une chaîne de caractères décrivant succinctement (et en *basic english*) la nature de l'erreur et, en arguments suivants, les objets explicitant les raisons de ce dysfonctionnement. Mentionnons toutefois que si les systèmes les plus rudimentaires se contentent d'émettre un message sybillin tel que `Bus error : core dumped` lorsqu'une erreur survient, d'autres abandonnent le calcul en cours et reviennent d'eux-mêmes dans la boucle fondamentale d'interaction. D'autres encore permettent d'associer à un calcul un gestionnaire d'exceptions qui se saisira de l'objet représentant l'erreur ou l'exception et qui décidera de la conduite à tenir. **[☞ p. 243]** Les machines-Lisp proposaient même des gestionnaires d'exceptions qui sont de véritables systèmes experts, analysant l'erreur et le code correspondant pour proposer les choix de correction les plus appropriés à l'utilisateur. Toute une gamme !

1.4 Évaluation des formes

Tout langage possède un certain nombre de formes syntaxiques de base intangibles, irredéfinissables et impervertissables. Ce sont, en Lisp, les *formes spéciales*. Elles sont représentées par des listes dont le premier terme est un symbole particulier appartenant à la liste des *opérateurs spéciaux*. Un dialecte de Lisp se caractérise par le jeu de formes spéciales qu'il arbore et sa bibliothèque de fonctions primitives, celles qui ne peuvent être écrites dans le langage même et qui ont des répercussions sémantiques profondes, comme, par exemple, `call/cc` en Scheme. Lisp n'est, d'une certaine manière, qu'un banal λ-calcul appliqué augmenté d'un jeu de formes spéciales. Mais le génie d'un Lisp particulier s'exprime par ce jeu. Scheme a choisi de minimiser le nombre d'opérateurs spéciaux (`quote`, `if`, `set !` et `lambda`) tandis que COMMON LISP (CLtL2 [Ste90]) en procure plus d'une trentaine circonscrivant les nombreux cas où de performantes générations de code véloce sont possibles.

Du fait du codage des formes spéciales, leur analyse syntaxique est simple et se fonde sur une analyse du premier terme des formes : un unique `case` suffit à la tâche. Lorsqu'une forme ne débute pas par un mot-clé, c'est par défaut une *application*[6] *fonctionnelle*. Pour l'instant nous ne retiendrons qu'un petit sous-ensemble de formes

[5] Notez que nous n'avons pas écrit « la fonction `wrong` ». Les techniques de récupération d'erreurs seront vues page 243.

[6] On peut distinguer l'*application fonctionnelle* où une fonction est appliquée à ses arguments d'une *application* qui est un programme autonome susceptible d'être lancé depuis un système d'exploitation. Les deux sens étant clairement différents, nous commettrons l'abus de parler seulement d'application.

spéciales générales (les chapitres suivants en introduiront de nouvelles plus spécialisées) à savoir : `quote`, `if`, `begin`, `set!` et `lambda`.

```
(define (evaluate e env)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e) (string? e) (char? e)
                 (boolean? e) (vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote)  (cadr e))
        ((if)     (if (evaluate (cadr e) env)
                      (evaluate (caddr e) env)
                      (evaluate (cadddr e) env) ))
        ((begin)  (eprogn (cdr e) env))
        ((set!)   (update! (cadr e) env (evaluate (caddr e) env)))
        ((lambda) (make-function (cadr e) (cddr e) env))
        (else     (invoke (evaluate (car e) env)
                          (evlis (cdr e) env) )) ) ) )
```

Pour alléger la définition précédente, l'analyse syntaxique a été réduite à son minimum et ne s'embarrasse pas de vérifier que les `if` sont vraiment ternaires[7], que les citations sont bien écrites, etc. Nous supposerons que les programmes analysés sont corrects.

1.4.1 Citation

La *citation*, ou forme spéciale `quote`, permet d'introduire une valeur qui, sans citation explicite, aurait pu être confondue avec une expression légale : un fragment de programme. La décision de représentation d'un programme sous la forme d'une expression du langage rend nécessaire, lorsque l'on souhaite citer une expression du langage, de la discriminer des programmes qui usurpent le même espace. D'autres choix syntaxiques auraient pu être faits qui auraient évité ce problème. Les M-expressions originellement prévues pour être la syntaxe normale des programmes écrits en Lisp [McC60] éliminaient ces problèmes mais interdisaient les macros qui se sont révélées un fabuleux outil d'extension de syntaxe : les M-expressions sont mortes peu après [McC78a]. La forme spéciale `quote` est donc le discriminateur entre programme et donnée.

La citation consiste à retourner en valeur le terme qui suit le mot-clé, ce qu'énonce fort brièvement le fragment :

```
... (case (car e)
     ((quote) (cadr e)) ...) ...
```

Y a-t-il une différence entre une citation implicite ou explicite, par exemple entre `33` et `'33` ou encore entre[8] `#(fa do sol)` et `'#(fa do sol)` ? La première comparaison porte sur des objets immédiats alors que la seconde porte sur des objets composites (quoique atomiques dans la terminologie lispienne). Il est possible d'imaginer des

[7] Les formes `if` ne sont pas nécessairement ternaires. Scheme et COMMON LISP acceptent les `if` binaires ou ternaires, EULISP et IS-Lisp uniquement les ternaires, Le-Lisp les `if` au moins binaires avec un `progn` (ou `begin` en Scheme) implicite sur l'alternant.

[8] Rappelons qu'en Scheme la notation `#(...)` représente la citation d'un vecteur.

sens divergents pour ces deux fragments. La citation explicite se contente de retourner comme valeur sa citation alors que `#(fa do sol)` pourrait, à chaque évaluation, retourner un nouvel exemplaire d'un vecteur à trois composantes initialisés par trois symboles particuliers. En d'autres mots, `#(fa do sol)` ne serait qu'une abréviation de `(vector 'fa 'do 'sol)` dont le comportement est fortement différent de `'#(fa do sol)` ainsi, du reste, que de `(vector fa do sol)`. Nous reviendrons plus tard sur le sens à accorder à la citation qui est loin d'être simple. **[☞ p. 133]**

1.4.2 Alternative

L'*alternative*, ou forme spéciale `if`, est considérée ici comme ternaire. C'est une structure de contrôle qui, après avoir évalué son premier argument (la *condition*), choisit, suivant la valeur obtenue, de retourner la valeur de son second (la *conséquence*) ou troisième (l'*alternant*) argument. C'est ce qu'exprime le fragment ci-dessous :

```
... (case (car e) ...
     ((if) (if (evaluate (cadr e) env)
               (evaluate (caddr e) env)
               (evaluate (cadddr e) env) )) ... ) ...
```

Cette programmation ne rend pas entièrement justice à la représentation des booléens. Comme le lecteur l'a certainement remarqué, nous mélangeons ici deux langages. Le premier est Scheme (ou du moins y ressemble singulièrement au point que l'on ne peut encore les distinguer) tandis que le second est aussi Scheme (ou du moins s'en approche). Le premier sert à implanter le second ; il y a donc entre eux le même rapport qu'il y a entre, par exemple, TEX [Knu84] et le langage de la première implantation de TEX : Pascal. La discussion sur le sens que revêt l'alternative peut aussi bien s'entendre comme définissant l'alternative du langage de définition que celle du langage en cours de définition.

La fonction `evaluate` retourne une valeur appartenant au langage en cours de définition. Cette valeur n'entretient a priori aucun rapport avec les booléens du langage de définition. Sachant que l'on reprend la convention que tout objet différent du booléen *faux* peut être considéré comme le booléen *vrai*, une programmation plus soigneuse serait donc :

```
... (case (car e) ...
     ((if) (if (not (eq? (evaluate (cadr e) env) the-false-value))
               (evaluate (caddr e) env)
               (evaluate (cadddr e) env) )) ... ) ...
```

On suppose bien sûr que la variable `the-false-value` a pour valeur la représentation (dans le langage de définition) du booléen *faux* du langage en cours de définition. Un large choix est ouvert pour représenter cette valeur et par exemple :

```
(define the-false-value (cons "false" "boolean"))
```

La comparaison d'une valeur quelconque au booléen *faux* étant effectuée par le comparateur physique `eq?`, une paire pointée fait donc très bien l'affaire et ne peut être confondue avec aucune autre valeur possible du langage en cours de définition.

La discussion n'est pas oiseuse. Le *sexe* du booléen *faux*, de la liste vide `()` et du symbole `NIL` défraie encore la chronique du monde lispien. La position la plus propre, indépendamment de toute volonté de préserver l'existant, est que *faux* est différent de `()`, qui n'est que la liste vide, et que tous deux n'ont rien à voir avec le symbole qui

s'épelle `N`, `I` et `L`. Cette position est celle de Scheme qui l'a adoptée quelques semaines avant d'être normalisé à l'IEEE [IEE91]. Là où cependant les choses se gâtent, c'est que `()` se prononce *nil* ! En Lisp traditionnel *faux*, `()` et `NIL` sont un seul et même symbole. En Le-Lisp, `NIL` est une variable dont la valeur est `()`, la liste vide étant assimilée au booléen *faux* et au symbole vide `||`.

1.4.3 Séquence

La *séquence* permet de condenser en une seule entité syntaxique, une suite de formes à évaluer séquentiellement. Similaire au fameux `begin...end` des langages de la famille d'Algol, Scheme préfixe cette forme spéciale par `begin` là où les autres Lisps emploient plutôt `progn`, généralisation immédiate de `prog1`, `prog2`, ... L'évaluation en séquence d'une suite de formes sera sous-traitée à une fonction particulière : `eprogn`.

```
... (case (car e) ...
     ((begin) (eprogn (cdr e) env)) ... ) ...
```

```
(define (eprogn exps env)
  (if (pair? exps)
      (if (pair? (cdr exps))
          (begin (evaluate (car exps) env)
                 (eprogn (cdr exps) env) )
          (evaluate (car exps) env) )
      '() ) )
```

Le sens de la séquence est maintenant biblique. On remarquera toutefois, au sein de la définition de `eprogn`, l'appel récursif terminal à `evaluate` pour le traitement du terme final de la séquence. Le calcul du terme final d'une séquence s'effectue comme si, à lui seul, il remplaçait la séquence toute entière. La récursion terminale sera mieux caractérisée ultérieurement. **[☞ p. 98]**

Une autre remarque concerne le sens qu'il faut attribuer à `(begin)`. Nous avons ici défini que `(begin)` retournait la liste vide `()`. Mais pourquoi diantre la liste vide, pourquoi pas n'importe quoi d'autre : `libellule` ou même `(papillon)` ? Ce choix provient de l'héritage de Lisp où l'usage de rendre `nil` lorsque rien ne semble s'imposer de meilleur prévalait. Mais dans un monde où `nil`, `()` et le booléen *faux* sont distincts, que rendre ? Nous singulariserons le langage décrit dans ce chapitre pour que `(begin)` retourne la valeur de `empty-begin`, soit le nombre (presque arbitraire) `813`[Leb05] :

```
(define (eprogn exps env)
  (if (pair? exps)
      (if (pair? (cdr exps))
          (begin (evaluate (car exps) env)
                 (eprogn (cdr exps) env) )
          (evaluate (car exps) env) )
      empty-begin ) )
(define empty-begin 813)
```

Le problème naît de ce que l'implantation que nous définissons doit nécessairement retourner une valeur. Le langage peut, comme Scheme, n'attribuer aucun sens particulier à `(begin)`, ce qui peut s'interpréter de deux façons au moins : ou bien cette écriture est permise au sein d'une implantation particulière : c'est une extension qui doit donc retourner une valeur librement élue par l'implantation en question ; ou bien

l'écriture `(begin)` est illicite et donc erronée. Par voie de conséquence, il vaut mieux s'abstenir d'utiliser cette forme pour laquelle aucune garantie de légalité et donc de valeur n'existe. Quelques implantations ont un objet `#<unspecified>` qui peut servir à cet usage ainsi que, plus généralement, dans tous les cas où l'on ne sait ce que l'on peut retourner car rien ne semble approprié. Cet objet est souvent imprimable et ne doit point être confondu avec l'indéfini. **[☞ p. 56]**

La séquence n'a pas d'intérêt si le langage est purement fonctionnel, c'est-à-dire sans aucun effet de bord. À quoi sert d'évaluer un programme dont on n'a cure du résultat ? C'est cependant méconnaître les jeux vidéo où la vitesse de la machine excède les réflexes des joueurs. Pour ralentir la machine, on programme des boucles d'attente obtenant ainsi de magnifiques effets de bord sur le temps. Voilà donc à quoi sert `progn` dans le domaine des jeux vidéo programmés en langage fonctionnel pur. En présence d'opérations de lecture ou d'écriture classiques, qui sont des effets de bord sur les flux d'entrée et de sortie, la séquence trouve tout son intérêt car il est nettement plus clair de poser une question (par `display`) puis d'attendre la réponse (par `read`) que l'inverse. La séquence est alors la forme explicite qui permet de séquentialiser une suite d'évaluations. D'autres formes spéciales peuvent aussi introduire un certain ordre comme l'alternative :

`(if` α β β`)` $\equiv$ `(begin` α β`)`

Mais cette règle peut aussi être simulée par[9] :

`(begin` α β`)` $\equiv$ `((lambda (`*void*`)` β`)` α`)`

Cette dernière règle montre, s'il en était encore besoin, que `begin` n'est pas une forme spéciale vraiment nécessaire à Scheme puisqu'elle est émulable par l'appplication fonctionnelle qui impose que les arguments soient calculés avant le corps de la fonction invoquée : c'est la technique dite d'*appel par valeur*.

1.4.4 Affectation

Comme dans de nombreux autres langages, la valeur d'une variable peut être modifiée ; on dit alors qu'on affecte la variable. Comme il s'agit de modifier dans l'environnement la valeur de la variable, nous laisserons ce soin à la fonction `update!`[10] qui sera explicitée plus loin.

```
... (case (car e)
      ((set!) (update! (cadr e) env (evaluate (caddr e) env))) ...
```

L'*affectation* s'effectue en deux temps : la nouvelle valeur est calculée puis devient la nouvelle valeur de la variable. Notez que la variable n'est pas le résultat d'un calcul. De nombreuses variantes sémantiques existent pour l'affectation que l'on discutera aussi plus tard. **[☞ p. 105]** Sachez toutefois que la valeur d'une affectation n'est pas spécifiée en Scheme.

[9] La variable *void* ne doit pas être libre dans β, ce qui est trivialement vérifiée si *void* n'apparaît pas du tout dans β. C'est pourquoi l'on utilise souvent des `gensym` pour créer de nouvelles variables garanties inédites, voir l'exercice 1.11.

[10] Conformément à l'usage courant en Scheme, les fonctions qui font des effets de bord ont leur nom suffixé par un point d'exclamation.

1.4.5 Abstraction

Les fonctions (pour *procedure* dans le jargon Scheme) sont le résultat de l'évaluation de la forme spéciale `lambda` qui elle-même se nomme, par référence au λ-calcul, une *abstraction*. Nous déléguerons à `make-function` le soin de véritablement créer la fonction en lui fournissant tous les paramètres disponibles dont elle, `make-function`, pourrait avoir besoin et notamment la liste des variables, le corps de la fonction et l'environnement courant :

```
... (case (car e) ...
     ((lambda) (make-function (cadr e) (cddr e) env)) ...) ...
```

1.4.6 Application fonctionnelle

Lorsqu'une liste ne possède pas un opérateur spécial en premier terme, c'est alors une *application fonctionnelle* ou, par référence au λ-calcul, une *combinaison*. La fonction, obtenue par évaluation du premier terme, est appliquée à ses arguments obtenus par évaluation des termes suivants. C'est exactement ce que reflète :

```
... (case (car e) ...
        (else (invoke (evaluate (car e) env)
                      (evlis (cdr e) env) )) ) ...
```

La fonction utilitaire `evlis` retourne d'une liste d'expressions la liste des valeurs de ces mêmes expressions. Elle se définit simplement comme :

```
(define (evlis exps env)
  (if (pair? exps)
      (cons (evaluate (car exps) env)
            (evlis (cdr exps) env) )
      '() ) )
```

La fonction `invoke` a la charge d'appliquer son premier argument (une fonction, sauf anomalie) à son second argument (la liste des arguments de la fonction de premier argument) et de retourner en résultat la valeur de cette application. Afin d'éclaircir les multiples usages du mot « argument » dans la phrase précédente, observez que `invoke` est semblable à la classique `apply`, hors l'irruption explicite de l'environnement. Nous remettons à plus tard la représentation exacte des fonctions et environnements.

Retour sur `evaluate`

La description précédente est assez précise. Quelques fonctions utilitaires n'ont pas encore été explicitées comme `lookup` et `update!`, qui traitent des environnements, et, similairement, `make-function` et `invoke`, qui traitent des fonctions. Pourtant, de nombreuses questions trouvent déjà réponse. Par exemple, il est évident que le dialecte défini a un seul espace de noms, est mono-valeur (ou Lisp_1) **[☞ p. 29]** et que les objets fonctionnels y sont présents. En revanche, quel est l'ordre d'évaluation des arguments ?

L'ordre d'évaluation des arguments du Lisp défini est semblable à l'ordre d'évaluation des arguments du `cons` qui apparaît dans `evlis`. On pourrait cependant imposer l'ordre que l'on souhaite, par exemple gauche-droite, par une séquence explicite comme dans :

```
(define (evlis exps env)
```

```
(if (pair? exps)
    (let ((argument1 (evaluate (car exps) env)))
      (cons argument1 (evlis (cdr exps) env)) )
    '() ) )
```

Sans augmenter l'arsenal de traits utilisés dans le Lisp de définition[11], nous avons augmenté la précision de la description du Lisp défini. La première partie de cet ouvrage tendra à définir de plus en plus précisément un certain dialecte tout en restreignant de plus en plus le dialecte servant à la définition.

1.5 Représentation de l'environnement

L'environnement associe des valeurs à des variables. La structure de donnée traditionnelle de Lisp pour représenter de telles associations est la bien nommée *liste d'associations* ou *A-liste*. Nous allons donc représenter l'environnement comme une A-liste associant variables et valeurs. Pour simplifier, nous représenterons les variables par les symboles de même nom !

Les fonctions `lookup` et `update!` se définissent donc très aisément :

```
(define (lookup id env)
  (if (pair? env)
      (if (eq? (caar env) id)
          (cdar env)
          (lookup id (cdr env)) )
      (wrong "No such binding" id) ) )
```

On voit apparaître un second cas d'erreur qui surgit lorsque l'on désire connaître la valeur d'une variable sans définition. Là encore, on se contentera d'exprimer le trouble dans lequel l'interprète est plongé par un appel à `wrong`.

Dans les vieux temps et principalement lorsque les interprètes disposaient de très peu de mémoire[12], les implanteurs ont souvent favorisé un mode *autoquote* généralisé. Une variable sans valeur en avait quand même une et cette valeur était le symbole de même nom ! Triste retour des choses que d'avoir attaché tant d'importance à séparer symbole et variable et de voir revenir (au galop) l'infernale confusion.

S'il est pratique pour les implanteurs de ne jamais provoquer une erreur et donc d'offrir un monde idyllique d'où l'erreur est bannie, ce noble dessein n'a qu'un inconvénient : le but d'un programme n'est pas de n'être pas erroné mais de remplir son office. L'erreur est une sorte de grossier garde-fou qui, quand elle survient, démontre que le programme est en désaccord avec son intention. Il est de bon aloi que soient signalées le plus rapidement possibles les erreurs et les contextes erronés afin que celles-là soient au plus tôt corrigées. Le mode *autoquote* est donc un mauvais choix de conception car il permet à certaines situations de croupir occultement.

La fonction `update!` qui modifie l'environnement est susceptible de provoquer la même erreur : on ne peut modifier la valeur d'une variable inconnue. Nous reviendrons sur ce point quand nous parlerons d'environnement global.

```
(define (update! id env value)
```

[11] Comme on le sait, `let` est une macro s'expansant en une application fonctionnelle : `(let ((`x π_1`))` π_2`)` $\equiv$ `((lambda (`x`)` π_2`)` π_1`)`.

[12] La mémoire (avec les entrées/sorties), bien que de prix sans cesse décroissant, forme toujours la partie la plus chère d'un calculateur.

```
(if (pair? env)
    (if (eq? (caar env) id)
        (begin (set-cdr! (car env) value)
               value )
        (update! id (cdr env) value) )
    (wrong "No such binding" id) ) )
```

Le contrat de la fonction `update!` prévoyait qu'elle retourne une valeur qui deviendrait celle, finale, de la forme spéciale d'affectation. La valeur de l'affectation n'est pas définie en Scheme, ce qui signifie qu'un programme portable ne peut s'attendre à une valeur précise. Plusieurs possibilités s'offrent comme de retourner *(i)* la valeur qui vient d'être affectée (c'est ce qui est fait plus haut dans `update!`) *(ii)* l'ancien contenu de la variable (ce qui pose un léger problème à l'initialisation lorsque l'on donne une première valeur à la variable) *(iii)* un objet représentant ce qui n'est pas spécifié, sur lequel on ne peut faire grand chose, quelque chose comme `#<UFO>`[13] *(iv)* la valeur d'une forme à valeur non spécifiée comme l'est la forme `set-cdr!` en Scheme.

L'environnement peut être vu comme un type abstrait composite. On peut extraire ou modifier des sous-parties d'un environnement, ce sont les fonctions de sélection ou modification. Il reste à définir comment sont construits et enrichis les environnements.

L'environnement initial est vide, ce qui se représente simplement :

```
(define env.init '())
```

Nous veillerons plus loin à procurer un environnement standard un peu plus étoffé.

Lorsqu'une fonction est appliquée, un nouvel environnement est construit qui lie les variables de la fonction à leur valeur. La fonction `extend` étend un environnement `env` avec une liste de variables `variables` et une liste de valeurs `values`.

```
(define (extend env variables values)
  (cond ((pair? variables)
         (if (pair? values)
             (cons (cons (car variables) (car values))
                   (extend env (cdr variables) (cdr values)) )
             (wrong "Too less values") ) )
        ((null? variables)
             (if (null? values)
                 env
                 (wrong "Too much values") ) )
        ((symbol? variables) (cons (cons variables values) env)) ) )
```

La principale difficulté est qu'il faut analyser syntaxiquement toutes les formes possibles qu'une *<liste-de-variables>* peut posséder au sein d'une abstraction en Scheme[14]. Une *liste de variables* est représentée par une liste de symboles éventuellement pointée, c'est-à-dire terminée non pas par `()` mais par un symbole. Plus formellement, une liste de variables répond à la pseudo-grammaire suivante :

[13] *Unknown Flying Object* est l'équivalent du célèbre (en français) `**OVNI**` de Le-Lisp.

[14] Certains systèmes Lisp, comme COMMON LISP, ont cédé à la tentation d'enrichir la liste de variables de mot-clés divers (`&aux`, `&key`, `&rest` ...) compliquant singulièrement le mécanisme de liaison. D'autres généralisent la liaison des variables au filtrage [SJ93].

```
<liste-de-variables>  :=  ()
                      |   <variable>
                      |   ( <variable> . <liste-de-variables> )
          <variable>  ∈   Symbol
```

Lorsque l'on étend un environnement, il doit y avoir accord entre les noms et les valeurs. Usuellement, il doit y avoir autant de valeurs qu'il y a de variables, sauf si la liste de variables se termine par une variable *pointée* ou n-aire qui peut prendre toutes les valeurs superflues réunies en une liste. Deux erreurs peuvent donc être signalées suivant qu'il y a trop ou pas assez de valeurs.

1.6 Représentation des fonctions

La façon la plus simple de représenter des fonctions est peut-être d'utiliser des fonctions. Ne vous y méprenez pas : les deux emplois du mot « fonction » dans la phrase précédente renvoient à des entités différentes. Plus précisément donc : « La façon la plus simple de représenter des fonctions du langage en cours de définition est d'utiliser des fonctions du langage d'implantation. » La représentation ici choisie va minimiser le protocole d'appel, c'est-à-dire la fonction `invoke` qui n'aura plus qu'à vérifier que son premier argument est bien une fonction, c'est-à-dire un objet applicable :

```
(define (invoke fn args)
  (if (procedure? fn)
      (fn args)
      (wrong "Not a function" fn) ) )
```

On ne peut vraiment faire plus simpliste[15].

Un nouveau cas d'erreur est apparu, lorsque l'on cherche à appliquer un objet qui n'est pas une fonction. La détection se fait ici au moment de l'application, c'est-à-dire après évaluation de tous les arguments. D'autres stratégies sont possibles comme de chercher à prévenir au plus tôt l'utilisateur. On peut, dans ce dernier cas, imposer un ordre dans l'évaluation des applications fonctionnelles :

1. évaluer le terme en position fonctionnelle ;
2. si sa valeur n'est pas applicable, déclencher une erreur ;
3. évaluer les arguments de la gauche vers la droite ;
4. comparer le nombre d'arguments et l'arité de la fonction à appliquer et déclencher une erreur s'il n'y a pas accord.

Ordonner l'évaluation des arguments de la gauche vers la droite a l'intérêt de la simplicité pour ceux qui lisent de gauche à droite et permet de mettre au point plus aisément puisque l'ordre d'évaluation est apparent mais complique toutefois la tâche du compilateur s'il cherche à réordonner l'évaluation des arguments pour, par exemple, améliorer l'allocation des registres car il lui faut prouver que le réordonnancement des expressions ne perturbe pas la sémantique du programme.

[15] Le lecteur pourrait se demander pourquoi avoir individualisé la fonction `invoke`, de définition aussi simple et seulement appelée une fois depuis `evaluate`. La raison est que nous fixons ici la structure générale des interprètes dont nous allons gaver le lecteur et que nous verrons d'autres codages moins simples mais plus performants requérant de compliquer `invoke`. On pourra aussi, avec fruit, s'atteler aux exercices 1.7 et 1.8.

On peut bien sûr chercher à faire encore mieux, en hâtant la vérification d'arité :

1. évaluer le terme en position fonctionnelle ;
2. si sa valeur n'est pas applicable, déclencher une erreur, sinon déterminer son arité ;
3. évaluer les arguments de la gauche vers la droite tant que leur nombre est toujours en accord avec l'arité précédente, déclencher une erreur sinon ;
4. appliquer la fonction à ses arguments[16].

COMMON LISP impose que les arguments soient évalués de la gauche vers la droite mais permet, pour des raisons d'efficacité, au terme en position fonctionnelle d'être évalué avant ou après. Scheme, quant à lui, n'impose aucun ordre d'évaluation pour les termes d'une application fonctionnelle. La position fonctionnelle n'y est pas distinguée. Qu'il n'y ait pas d'ordre imposé signifie que l'évaluateur est libre de choisir celui qui lui plaît ; le compilateur peut donc réordonner sans souci les termes. L'utilisateur, ne sachant plus quel est l'ordre choisi, doit donc utiliser `begin` lorsqu'il souhaite séquentialiser certains effets. Stylistiquement parlant, il est peu élégant d'user de l'application fonctionnelle pour séquentialiser des effets de bord : évitez donc, par exemple, d'écrire `(f (set! f` π`) (set! f` π'`))` où rien ne transparaît de la fonction qui sera réellement appliquée. Les erreurs de ce genre, dues à la méconnaissance de l'ordre précis d'évaluation sont très dures à détecter.

Environnement d'exécution du corps d'une fonction

L'application d'une fonction revient à évaluer son corps dans un environnement où ses variables sont liées aux valeurs qu'elles ont prises lors de l'application. Rappelons qu'à l'appel de `make-function`, nous lui avions fourni tous les paramètres dont nous disposions depuis `evaluate`. Dans tout le reste de cette section, les différents environnements en jeu seront notés en italiques afin d'attirer agréablement le regard.

Nudisme

Tentons une première définition :

```
(define (make-function variables body env)
  (lambda (values)
     (eprogn body (extend env.init variables values)) ) )
```

Conformément au contrat précédemment exposé, le corps de la fonction est bien évalué dans un environnement où les variables sont liées à leur valeur. Par exemple, le combinateur K ainsi défini `(lambda (a b) a)` peut être appliqué comme dans :

```
(K 1 2)   →  1
```

L'ennui est que les moyens mis à disposition d'une fonction sont dès lors assez réduits. Le corps d'une fonction ne peut rien utiliser d'autre que ses variables puisque l'environnement initial *env.init* est vide. Il n'a même pas accès à l'environnement global dans lequel sont prédéfinis les fonctions usuelles comme `car`, `cons`, etc.

[16] La fonction pourra alors examiner le type de ses arguments mais cette tâche ne relève pas du protocole d'appel fonctionnel.

Rustine

Réitérons donc notre précédent et infructueux essai :

```
(define (make-function variables body env)
  (lambda (values)
     (eprogn body (extend env.global variables values)) ) )
```

Cette nouvelle version permet donc au corps des fonctions d'utiliser l'environnement global et toutes ses fonctions usuelles. Mais comment définir globalement deux fonctions mutuellement récursives et que vaut l'expression de gauche qui figure macroexpansée à droite ?

```
(let ((a 1))                          ((lambda (a)
  (let ((b (+ 2 a)))    ≡               ((lambda (b)
     (list a b) ) )                        (list a b) )
                                         (+ 2 a) ) )
                                       1 )
```

Détaillons comment est évaluée cette expression :

```
((lambda (a) ((lambda (b) (list a b)) (+ 2 a))) 1)|
                                                  | env.global
= ((lambda (b) (list a b)) (+ 2 a))|
                                   | a→ 1
                                   | env.global
= (list a b)|
            | b→ 3
            | env.global
```

Le corps de la fonction interne `(lambda (b) (list a b))` s'évalue dans un environnement obtenu en étendant l'environnement global par la variable `b`. Cet environnement exclut la variable `a` qui n'est pas visible ! Notre essai échoue donc encore.

Rustine révisée

Puisque l'on désire voir la variable `a` au sein de la fonction interne, il suffit de fournir l'environnement courant à `invoke` qui le transmettra aux fonctions appelées. Il y a donc une notion d'environnement courant, tenu à jour par `evaluate` et `invoke`. Modifions donc ces fonctions mais, toutefois, afin de ne pas confondre ces nouvelles (et temporaires) définitions avec les anciennes, nous les préfixerons par `d.`.

```
(define (d.evaluate e env)
  (if (atom? e) ...
      (case (car e)
        ...
        ((lambda) (d.make-function (cadr e) (cddr e) env))
        (else (d.invoke (d.evaluate (car e) env)
                        (evlis (cdr e) env)
                        env )) ) ) )
(define (d.invoke fn args env)
  (if (procedure? fn)
      (fn args env)
      (wrong "Not a function" fn) ) )

(define (d.make-function variables body def.env)
  (lambda (values current.env)
     (eprogn body (extend current.env variables values)) ) )
```

On pourra observer dans la définition de `d.make-function` que fournir l'environnement de définition `env` n'est plus utile puisque seul est utilisé l'environnement courant *current.env*.

Et maintenant, notre exemple marche comme le montre la même expression redécorée des environnements utilisés :

```
((lambda (a) ((lambda (b) (list a b)) (+ 2 a))) 1)|
                                                  | env.global
= ((lambda (b) (list a b)) (+ 2 a))|
                                   | a→ 1
                                   | env.global
= (list a b)|
            | b→ 3
            | a→ 1
            | env.global
```

On voit bien sur cet exemple la discipline de pile qu'adoptent les liaisons. Toute forme liante empile ses nouvelles liaisons sur l'environnement courant et les enlève après exécution.

Rustine révisée révisée

Mais il y a encore un problème. Considérez cette légère variation :

```
(((lambda (a)
    (lambda (b) (list a b)) )
  1 )
 2 )
```

La fonction `(lambda (b) (list a b))` est créée dans un environnement où `a` est lié à `1` mais au moment où elle est appliquée, l'environnement courant est étendu par la seule liaison concernant `b` d'où, encore une fois, `a` est absente. La fonction `(lambda (b) (list a b))` a donc oublié (ou perdu) sa variable `a`.

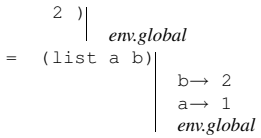
Le lecteur avait pu le remarquer dans la précédente définition de `d.make-function`, deux environnements étaient présents : l'environnement de définition de la fonction *def.env* et l'environnement d'appel de la fonction, *current.env*. Deux types d'instants marquent la vie d'une fonction : sa naissance et ses applications. S'il n'y a qu'une seule et unique naissance, il peut en revanche y avoir de nombreuses applications voire même aucune. Le *seul* environnement que l'on puisse donc naturellement associer à une fonction est celui de sa création[17]. Revenons donc aux précédentes définitions des fonctions `evaluate` et `invoke` puis posons que `make-function` est maintenant :

```
(define (make-function variables body env)
  (lambda (values)
    (eprogn body (extend env variables values)) ) )
```

Tous les exemples se comportent alors favorablement et en particulier le précédent qui a pour trace d'évaluation :

```
(((lambda (a) (lambda (b) (list a b))) 1) 2)|
                                            | env.global
= ( (lambda (b) (list a b))|
                           | a→ 1
                           | env.global
```

[17] On peut aussi laisser au programme le soin de choisir explicitement l'environnement dans lequel clore la fonction, voir la forme `closure` page 107.



La forme `(lambda (b) (list a b))` est créée dans l'environnement global étendu par la variable `a`. Lorsqu'elle est appliquée (dans l'environnement global), elle étend son environnement de définition par `b` et permet donc à son corps de s'évaluer dans un environnement où `a` et `b` sont connues. Lorsqu'elle retourne son résultat, l'évaluation se poursuit dans l'environnement global. On nomme souvent *fermeture* la valeur d'une abstraction car elle clôt (ou enferme) son environnement de définition.

On remarquera que la présente définition de `make-function` use elle-même (au sein du langage de définition) de fermetures. Ce n'est pas obligatoire comme on le verra au chapitre 3. **[☞ p. 87]** La fonction `make-function` a pour valeur une fermeture, ceci est un trait distinctif des langages fonctionnels d'ordre supérieur.

1.6.1 Liaisons dynamique et lexicale

La précédente discussion a au moins deux mérites. En premier lieu, elle exhibe clairement la complexité liée aux problèmes d'environnement. Une évaluation s'effectue toujours au sein d'un certain environnement et la gestion de celui-ci est un point majeur que doivent résoudre efficacement les évaluateurs. Nous verrons, au chapitre 3, des constructions plus complexes, comme les échappements et la forme `unwind-protect` qui nécessitent de définir très précisément quels sont les environnements en lice.

Les deux dernières variantes présentées plus haut caractérisent respectivement (comme on les nomme usuellement) les liaisons *dynamique*[18] ou *lexicale*. Dans un *Lisp lexical*, une fonction évalue son corps dans son environnement de définition étendu par ses variables tandis que, dans un *Lisp dynamique*, une fonction étend l'environnement courant, c'est-à-dire celui d'application. Si la mode actuellement préfère les Lisps lexicaux, il ne faut pas en conclure que les langages dynamiques n'ont aucun avenir. D'une part, il existe des langages fort utiles, récents et dynamiques comme TEX [Knu84], Gnu Emacs Lisp [LLSt93] ou Perl [WS90][19] ; d'autre part, le concept de liaison dynamique est un important concept de programmation : il correspond à l'établissement d'une liaison valide seulement le temps d'un calcul et défaite automatiquement (de façon garantie) à l'achèvement de celui-ci[20]. Ce tour programmatique peut être employé avec fruit dans des calculs prospectifs comme par exemple ceux que l'on mène en intelligence artificielle. On pose une hypothèse à partir de laquelle on développe des conséquences. Lorsque l'on découvre une incohérence, on se doit

[18] Dans le contexte des langages à objets, le terme de *liaison dynamique* est plutôt associé au fait que la méthode associée à un message est déterminée par le type dynamique de l'objet à qui il est envoyé plutôt que par son type statique.

[19] Perl depuis procure à la fois des variables dynamiques (avec le mot-clé `local`) et des variables lexicales (avec le mot-clé `my`).

[20] C'est cette même fin de calcul que capture la clause `finally` d'une forme `try` dans de nombreux langages tels que Caml, Java, Perl, etc.

d'abandonner cette hypothèse afin d'en explorer une autre : c'est la technique bien connue de retour arrière. Si les conséquences ont été enregistrées sans aucun effet de bord, par exemple dans des structures telles que des listes d'association, l'abandon de l'hypothèse recyclera automatiquement les conséquences. Si, par contre, on avait usé de modifications physiques telles que des affectations de variables globales, modifications de tableaux, etc., alors l'abandon de l'hypothèse doit conduire à la restauration de l'environnement dans lequel cette dernière fut formulée. Un seul oubli serait fatal ! La liaison dynamique permet d'assurer que, pendant tout un calcul et quelle que soit l'issue de celui-ci, la variable dynamique sera toujours correctement positionnée.

Les variables sont des entités programmatiques dotées d'une *portée*. La portée d'une variable est une notion essentiellement géographique correspondant à la zone du texte du programme où elle est visible et donc accessible. En Scheme pur (c'est-à-dire débarrassé des syntaxes superflues, mais utiles, telle `let`), une seule forme liante existe : `lambda`, c'est la seule forme introduisant des variables et leur conférant une portée réduite au corps seul de la fonction. La portée d'une variable dans un Lisp dynamique n'est par contre et a priori aucunement restreinte. Considérez l'exemple suivant :

```
(define (foo x) (list x y))
(define (bar y) (foo 1991))
```

En Lisp lexical, la variable `y` dans `foo` est une référence à la variable globale `y` qui ne peut aucunement être confondue avec la variable locale `y` de `bar`. En Lisp dynamique, la variable `y` de `bar` est visible, et vue, du corps de la fonction `foo`[21] puisque lorsque `foo` est invoquée, l'environnement courant contient cette variable `y`. Si l'on donne donc la valeur `0` à la variable globale `y`, on obtient alors les résultats suivants :

```
(define y 0)
(list (bar 100) (foo 3))    → ((1991 0) (3 0))      en Lisp lexical
(list (bar 100) (foo 3))    → ((1991 100) (3 0))    en Lisp dynamique
```

On remarquera, en Lisp dynamique, que `bar` n'a aucun moyen de savoir que la fonction `foo` qu'elle appelle référence sa propre variable `y`. Inversement, la fonction `foo` ne sait pas où peut se trouver la variable `y` qu'elle référence. C'est pour cela que `bar` doit installer `y` dans l'environnement courant afin que `foo` puisse l'y retrouver. Enfin et juste avant que `bar` ne retourne, `y` doit être retirée de l'environnement courant. On remarquera cependant qu'en l'absence de variables libres, les deux sortes de Lisp sont indiscernables.

La liaison lexicale a été ainsi nommée car l'on peut toujours, de toute variable et à partir du simple texte de la fonction, retrouver la forme l'ayant liée ou savoir si c'est une variable globale. La méthode est simple, on pointe son crayon (ou sa souris) sur la variable et on remonte de la droite vers la gauche, de bas en haut jusqu'à trouver la première forme englobante liant cette même variable.

Le nom de la liaison dynamique joue sur un autre concept qui est la *durée de vie dynamique* que l'on étudiera plus tard. **[☞ p. 73]** Scheme ne connaît de variables que lexicales ; COMMON LISP procure les deux sortes de liaisons sous une même syntaxe ; EULISP et IS-Lisp enfin séparent clairement (et syntaxiquement) les deux en deux espaces de noms séparés. **[☞ p. 40]**

[21] Pour l'étymologie de `foo`, voir [Ray91].

La portée peut être occultée localement par masquage : une variable peut en cacher une autre si elles portent le même nom. Les formes liantes étant lexicalement emboîtées, les portées lexicales sont donc disjointes ou emboîtées : c'est la fameuse discipline de bloc héritée d'Algol 60.

Légèrement inspiré du λ-calcul à qui il a emprunté le nom de la forme spéciale `lambda` [Per79], Lisp 1.0 fut défini comme un Lisp dynamique. Toutefois John McCarthy a reconnu assez tôt qu'il attendait que l'expression suivante retournât `(2 3)` plutôt que `(1 3)` :

```
(let ((a 1))
  ((let ((a 2)) (lambda (b) (list a b)))
   3 ) )
```

Cette anomalie (bogue ?) fût corrigée par introduction d'une nouvelle forme spéciale nommée `function` dont l'argument était une forme `lambda` et qui créait une fermeture, c'est-à-dire une fonction associée à son environnement de définition. Lorsque cette fermeture était appliquée, au lieu d'étendre l'environnement courant, elle étendait son environnement de définition qu'elle avait clos (c'est-à-dire préservé) en elle. Plus programmatiquement, la forme spéciale `function`[22] se définirait ainsi avec `d.evaluate` et `d.invoke` :

```
(define (d.evaluate e env)
  (if (atom? e) ...
      (case (car e)
        ...
        ((function)   ; Syntax: (function (lambda variables body))
         (let* ((f (cadr e))
                (fun (d.make-function (cadr f) (cddr f) env)) )
           (d.make-closure fun env) ) )
        ((lambda) (d.make-function (cadr e) (cddr e) env))
        (else (d.invoke (d.evaluate (car e) env)
                        (evlis (cdr e) env)
                        env )) ) ) )

(define (d.invoke fn args env)
  (if (procedure? fn)
      (fn args env)
      (wrong "Not a function" fn) ) )
(define (d.make-function variables body env)
  (lambda (values current.env)
     (eprogn body (extend current.env variables values)) ) )
(define (d.make-closure fun env)
  (lambda (values current.env)
     (fun values env) ) )
```

Mais l'histoire ne se finit pas avec `function` qui fut considérée comme une complaisance que l'utilisateur devait accorder à une implantation insuffisante. Les premiers compilateurs se rendirent très vite compte que, pour des raisons de performance,

[22] Cette simulation n'est point tout à fait exacte en ce sens que dans de nombreux dialectes (comme dans [Ste84]), `lambda` n'est pas une forme spéciale mais un mot-clé (une sorte de marqueur syntaxique similaire au `else` qui peut figurer, en Scheme, dans un `cond` ou un `case`). Il est en effet trivial de remarquer que `lambda` ne nécessite pas `d.evaluate` pour être traitée et de restreindre les emplacements où peuvent apparaître les formes `lambda` : en premier terme des applications fonctionnelles, accompagnée de `function` ou dans les définitions fonctionnelles.

les environnements lexicaux avaient un très grand avantage qui était que l'on savait dès la compilation, et dans tous les cas, où se trouveraient les variables à l'exécution : un accès quasi direct pouvait être engendré plutôt qu'une recherche dynamique de la valeur. Par défaut donc, toutes les variables des fonctions compilées étaient considérées comme lexicales, sauf celles qui étaient explicitement déclarées comme dynamiques ou, dans le jargon, *spéciales*. La déclaration `(declare (special x))` pour le seul usage du compilateur (Lisp 1.5, MacLisp, COMMON LISP etc.) désigne la variable *x* comme ayant un comportement spécial.

Mais il n'y avait pas seulement ces raisons d'efficacité, il y a aussi la perte de la *transparence référentielle*. La transparence référentielle est la propriété que possède un langage quand la substitution d'une expression dans un programme par une autre expression équivalente (c'est-à-dire calculant la même chose) ne modifie pas le comportement du programme : ils retournent la même valeur ou ne terminent pas tous les deux. Considérez par exemple :

```
(let ((x (lambda () 1))) (x)) ≡ ((let ((x (lambda () 1))) x)) ≡ 1
```

La transparence référentielle est perdue dès que le langage dispose d'effets de bord ; dans ce cas, de nouvelles définitions plus précises quant à la relation d'équivalence à employer pour parler de transparence référentielle sont nécessaires. Scheme, sans affectation, ni effets de bord, ni manipulation de continuation est référentiellement transparent. **[☞ Ex. 3.10]** C'est aussi un but vers lequel on doit tendre lorsque l'on cherche à écrire des programmes réutilisables, c'est-à-dire dépendant le moins possible du contexte d'emploi.

Les variables d'une fonction telle que `(lambda (u) (+ u u))` sont ce que l'on nomme traditionnellement, *muettes*. Leur nom n'a aucune importance et peut être remplacé par tout autre nom. La fonction `(lambda (n347) (+ n347 n347))` n'est ni plus ni moins[23] que la fonction `(lambda (u) (+ u u))`. On peut donc attendre du langage que soit respecté cet invariant. Il n'en est rien en Lisp dynamique. Considérez seulement[24] :

```
(define (map fn l)          ; mapcar in Lisp
  (if (pair? l)
      (cons (fn (car l)) (map fn (cdr l)))
      '() ) )
(let ((l '(a b c)))
  (map (lambda (x) (list-ref l x))
       '(2 1 0) ) )
```

En Scheme, le résultat serait `(c b a)` mais deviendrait `(0 0 0)` en Lisp dynamique ! La raison est que la variable `l` qui est libre dans le corps de `(lambda (x) (list-ref l x))` est interceptée, capturée, accrochée par la variable `l` de `map`.

On peut simplement résoudre cet ennui en changeant les noms en conflit, il suffit de renommer l'une des deux variables `l`. On choisira de renommer celle de `map` qui est probablement plus utile. Mais quel nom choisir qui puisse à peu près garantir que le même problème ne resurgira pas ? Nous suggérons de préfixer les noms des variables par le numéro de sécurité sociale du programmeur et de les suffixer par l'heure universelle exprimée en centièmes de seconde sexagésimale de temps moyen compté

[23] En termes techniques du λ-calcul, ce changement de noms de variables s'appelle une α-conversion.

[24] La fonction `list-ref` extirpe d'une liste son élément de rang n.

à partir du 1er janvier 1901 à midi. Les risques de collision seront ainsi nettement amoindris mais seulement amoindris, au péril de la lisibilité !

La situation au début des années quatre-vingt était donc particulièrement délicate. On enseignait Lisp par l'observation de son interprète qui différait du compilateur sur le point (fondamental) de la valeur d'une variable. Scheme ayant montré dès 1975 [SS75] que l'on pouvait réconcilier interprète et compilateur et donc vivre dans un monde complètement lexical, la définition de COMMON LISP entérina ce fait et postula que la *bonne* sémantique était celle du compilateur, que donc tout était lexical par défaut. L'interprète n'avait plus qu'à se conformer à ces nouveaux canons. Le succès grandissant de Scheme, des langages fonctionnels tel ML et de leurs divers épigones, répandit puis imposa cette nouvelle vue.

1.6.2 Implantation superficielle ou profonde

Mais les choses ne sont pas si simples et comme dans la fable de la lance et de la cuirasse, les implanteurs ont su trouver des moyens d'accélérer la vitesse de recherche de la valeur des variables dynamiques. Quand l'environnement est représenté comme une liste d'association, le coût de la recherche de la valeur d'une variable (la fonction `lookup`) est linéaire par rapport à la longueur de la liste[25]. Ce mécanisme est nommé *liaison profonde* (*deep binding*). Une autre technique existe, dite de *liaison superficielle* (*shallow binding*), où à chaque variable est associé un endroit où est toujours stockée sa valeur indépendamment de l'environnement courant. La plus simple réalisation est que cet endroit soit un champ particulier du symbole associé à la variable : la `Cval` ou *value-cell*. Le coût de `lookup` devient constant (une indirection suivie d'un éventuel déplacement). Comme il y a rarement gain sans perte, il faut avouer que l'appel fonctionnel est alors plus coûteux puisqu'il faut sauvegarder les valeurs des variables qui vont être liées et modifier les symboles associés aux variables pour qu'ils contiennent leurs nouvelles valeurs. Au retour de la fonction, il faut en plus restaurer les anciennes valeurs dans les symboles d'où elles provenaient, ce qui peut compromettre la récursion terminale (mais voir [SJ93]).

On peut simuler partiellement la liaison superficielle[26] en changeant la représentation de l'environnement. On supposera dans ce qui suit que les listes de variables ne sont pas pointées (ce qui allège le décodage des listes de variables) et l'on ne vérifiera pas l'arité des fonctions. Ces nouvelles fonctions sont préfixées par `s.` afin d'être reconnues.

```
(define (s.make-function variables body env)
  (lambda (values current.env)
    (let ((old-bindings
           (map (lambda (var val)
                  (let ((old-value (getprop var 'apval)))
                    (putprop var 'apval val)
                    (cons var old-value) ) )
```

[25] Fort heureusement, les statistiques montrent que l'on cherche plus souvent les premières variables que celles qui sont plus profondément enfouies. D'autre part on remarquera que les environnements lexicaux sont plus courts que les environnements dynamiques qui comportent toutes les liaisons en cours de calcul.

[26] L'affectation d'une variable close [BCSJ86] n'est toutefois pas traitée ici.

```
                variables
                values ) ))
        (let ((result (eprogn body current.env)))
          (for-each (lambda (b) (putprop (car b) 'apval (cdr b)))
                    old-bindings )
          result ) ) ) )
(define (s.lookup id env)
  (getprop id 'apval) )
(define (s.update! id env value)
  (putprop id 'apval value) )
```

Les fonctions `putprop` et `getprop` (non standard, en Scheme, mais semblables aux fonctions `put` et `get` qui apparaissent dans [AG85] **[☞ Ex. 2.6]** car elles mettent en œuvre de fort inefficaces effets de bord globaux) simulent le champ[27] dans lequel les symboles mémorisent la valeur des variables éponymes. Indépendamment de leur implantation réelle[28], ces fonctions doivent être considérées comme de coût constant.

On remarquera dans la simulation précédente l'évanouissement complet de l'environnement `env` qui ne sert plus à rien. Cette disparition impose que soit modifiée l'implantation des fermetures car elles ne peuvent plus clore l'environnement puisque celui-ci n'existe plus ! La technique, que l'on reverra plus tard, consiste en l'analyse du corps de la fonction pour en extirper les variables libres et les traiter de la belle manière.

La liaison profonde favorise le changement d'environnement et la programmation multi-tâches au détriment de la recherche des valeurs des variables ; la liaison superficielle favorise la recherche des valeurs des variables au détriment de l'appel fonctionnel. Les deux techniques ont été mariées par Henry Baker [Bak78] : c'est la technique de marcottage ou *rerooting*. Rappelons pour finir que liaisons superficielle ou profonde ne sont que des techniques d'implantation qui n'ont rien à voir avec la sémantique des liaisons.

1.7 Environnement global

Un environnement global vide fait pauvre. La plupart des systèmes Lisp procurent des bibliothèques (pour *libraries*) plus qu'étoffées : plus de 700 fonctions pour COMMON LISP(CLtL1), plus de 1 500 pour Le-Lisp, plus de 10 000 pour ZetaLisp etc. Sans bibliothèque, Lisp ne serait qu'une sorte de λ-calcul dans lequel on ne pourrait même pas imprimer les résultats de ses computs. La notion de bibliothèque est importante car, si l'essentiel du langage, pour le réalisateur de l'évaluateur, tient dans les formes spéciales, ce sont les bibliothèques qui souvent font la différence pour l'utilisateur. Le folklore Lisp retient que c'est l'absence des fonctions trigonométriques banales qui ont fait s'écarter de Lisp les numériciens des premiers âges. Savoir différencier

[27] Le nom de la propriété utilisée, `apval`, est un hommage au nom utilisé dans [MAE+62] lorsque les valeurs étaient *réellement* stockées dans les P-listes.

[28] Ces fonctions balayent linéairement la liste des propriétés (ou P-liste) des symboles jusqu'à trouver la propriété concernée. Le coût de cette recherche est linéaire en la longueur de la P-liste sauf si l'implantation associe symboles et propriétés par une table de hachage.

ou intégrer symboliquement [Sla61] des fonctions, c'est bien mais sans sécante, ni cosécante, que peut-on faire ?

On s'attend à trouver dans l'environnement global toutes les fonctions usuelles comme `car`, `cons`, etc. On y trouve aussi de simples variables dont les valeurs sont des données bien connues comme les booléens ou la liste vide. Par commodité et bien que nous n'ayons encore rien dit des macros **[☞ p. 295]** qui sont des phénomènes très complexes que nous analyserons plus tard, nous allons en définir deux qui amélioreront l'élaboration de l'environnement global. On définira l'environnement global comme une extension de l'environnement initial (vide) `env.init`.

```
(define env.global env.init)
(define-syntax definitial
  (syntax-rules ()
    ((definitial name)
     (begin (set! env.global (cons (cons 'name 'void) env.global))
            'name ) )
    ((definitial name value)
     (begin (set! env.global (cons (cons 'name value) env.global))
            'name ) ) ) )
(define-syntax defprimitive
  (syntax-rules ()
    ((defprimitive name value arity)
     (definitial name
        (lambda (values)
          (if (= arity (length values))
              (apply value values)       ; The real apply of Scheme
              (wrong "Incorrect arity"
                     (list 'name values) ) ) ) ) ) ) )
```

On définira tout d'abord quelques constantes bien utiles qui, toutes les trois, n'appartiennent pas au standard Scheme. Notez que `t` est une variable du Lisp en cours de définition tandis que `#t` est une valeur dans le Lisp de définition. Toute valeur autre que celle de `the-false-value` ferait tout aussi bien l'affaire.

```
(definitial t #t)
(definitial f the-false-value)
(definitial nil '())
```

S'il est utile de disposer de variables globales permettant d'obtenir les véritables objets qui représentent les booléens ou la liste vide, une autre solution est de procurer une syntaxe appropriée pour ce faire. Scheme use des syntaxes `#t` et de `#f` dont les valeurs sont les booléens *vrai* et *faux*. L'intérêt de ces syntaxes est qu'elles sont impervertissables et toujours visibles : (*i*) visible globalement, car on peut écrire dans tout contexte `#t` pour dire *vrai*, même si une variable locale se nomme `t` ! ; (*ii*) impervertissable, car de nombreux évaluateurs autorisent d'altérer la valeur de la variable `t`, ce qui conduit à de délicieux casse-têtes où l'on peut voir `(if t 1 2)` valoir 2 comme dans `(let ((t #f)) (if t 1 2))`. Plusieurs solutions sont possibles. La plus simple est de visser l'immuabilité de ces constantes dans l'évaluateur :

```
(define (evaluate e env)
  (if (atom? e)
      (cond ((eq? e 't) #t)
            ((eq? e 'f) #f)
            ...
            ((symbol? e) (lookup e env))
```

```
            ...
            (else (wrong "Cannot evaluate" e)) )
        ... ) )
```

On peut aussi introduire le concept de liaison *muable* ou *immuable*. Une liaison immuable ne peut faire l'objet d'une affectation. Nul ne peut donc altérer la valeur d'une variable ayant une liaison immuable. Ce concept existe de fait, quoique occultement, dans de nombreux systèmes. Le concept de *fonctions intégrables* désigne les fonctions (*inlined* ou *open coded*) **[☞ p. 190]** dont le corps convenablement instancié peut remplacer les appels. Remplacer `(car x)` par le code qui extrait, de la valeur de la paire pointée `x`, le contenu de son champ `car` sous-entend l'hypothèse importante que nul n'altère (n'a altéré ou n'altèrera) la valeur de la variable globale `car`. Songez au malheur qu'illustre le fragment de session suivant :

```
(set! my-global (cons 'c 'd))
   → (c . d)
(set! my-test (lambda () (car my-global)))
   → #<MY-TEST procedure>
(begin (set! car cdr)
       (set! my-global (cons 'a 'b))
       (my-test) )
   → ?????
```

Heureusement encore, la réponse ne peut être que `a` ou `b` ! Si `my-test` utilise la valeur de `car`, qui était courante au moment où `my-test` fut définie, la réponse sera `a`. Si `my-test` utilise la valeur courante de `car`, la réponse sera `b`. On comparera avec fruit le comportement de `my-test` et de `my-global`, sachant que l'on observe plutôt le premier comportement pour `my-test` (quand l'évaluateur est un compilateur) et plutôt le second pour `my-global`. **[☞ p. 51]**

On ajoutera quelques variables de travail usuelles à l'environnement global car il n'y a pas de création dynamique de variables globales dans l'évaluateur présenté. Les noms proposés ci-dessous recouvrent plus de 96% des noms de fonctions auxquels songent spontanément les lispiens soucieux d'éprouver un nouvel évaluateur[29] :

```
(definitial foo)
(definitial bar)
(definitial fib)
(definitial fact)
```

Enfin l'on définira quelques fonctions mais pas toutes car leur énoncé serait soporifique. La principale difficulté est d'adapter les primitives du Lisp sous-jacent au protocole d'appel du Lisp en cours de définition. Sachant que les arguments sont réunis par l'interprète en une liste, il suffit d'utiliser `apply`[30]. On remarquera que le respect de l'arité de la primitive est assuré grâce à l'expansion que procure `defprimitive`.

```
(defprimitive cons cons 2)
(defprimitive car car 1)
(defprimitive set-cdr! set-cdr! 2)
(defprimitive + + 2)
(defprimitive eq? eq? 2)
(defpredicate < < 2)
```

[29] Ces variables sont malheureusement initialisées. C'est un défaut qui sera corrigé plus tard.

[30] Félicitons-nous encore de ce qu'`invoke` ne s'appelle pas `apply` !

1.8 Lancement de l'interprète

Il ne reste plus qu'à indiquer au lecteur, pressé de mettre en œuvre le langage que nous venons de définir, comment « zapper » dans ce nouveau monde.

```
(define (chapter1-scheme)
  (define (toplevel)
     (display (evaluate (read) env.global))
     (toplevel) )
  (toplevel) )
```

On pourra, en guise d'exercice, affubler de quelques invites spirituelles (voire même zozotantes, ce que signifie *to lisp*), cet interprète quelque peu lapidaire ainsi qu'implanter une fonction permettant d'en sortir.

1.9 Conclusion

A-t-on vraiment défini un langage ? Nul ne peut douter que la fonction `evaluate` peut être lancée, qu'on peut lui soumettre des expressions dont elle retourne patiemment la valeur, quand le calcul termine, bien évidemment. Mais la fonction `evaluate` elle-même n'a de sens que par rapport au langage qui a servi de définition et, en absence de définition du langage de définition, rien n'est sûr. Le réflexe d'auto-amorçage circulaire jouant naturellement chez tout lispien de pure souche, il suffit d'identifier le langage de définition à celui qu'on définit. On a donc un langage L défini par une fonction `evaluate` écrite dans le langage L. Le langage défini est donc solution de l'équation en L :

$$\forall \pi \in \textbf{Programme}, L_{\texttt{(evaluate (quote } \pi \texttt{) env.global)}} \equiv L\pi$$

Pour n'importe quel programme π, l'évaluation de π en L (notée $L\pi$) doit exhiber le même comportement (et donc la même valeur si π termine) que l'évaluation de l'expression `(evaluate (quote` π`) env.global)` toujours en L. Une conséquence amusante de cette équation est que la fonction `evaluate` est capable de s'auto-interpréter[31] : Les expressions suivantes sont donc équivalentes :

```
(evaluate (quote π) env.global) ≡
  (evaluate (quote (evaluate (quote π) env.global)) env.global)
```

Existe-t-il des solutions à cette équation ? La réponse est oui et elles sont légion ! Comme on l'a déjà vu plus haut, l'ordre d'évaluation n'est pas nécessairement apparent dans la définition d'`evaluate` et beaucoup d'autres propriétés du langage sont inconsciemment *héritées* par le langage défini. On ne peut en fait presque rien dire car il existe aussi des solutions triviales à cette équation comme le langage L_{2001}, qui a pour sémantique que tout programme écrit en L_{2001} a pour valeur le nombre 2001. Le langage L_{2001} satisfait trivialement l'équation précédente. Il faudra donc recourir à d'autres méthodes si l'on souhaite vraiment définir un langage et ce sera l'objet de chapitres ultérieurs.

[31] Il faut cependant débarrasser la définition d'`evaluate` des abréviations syntaxiques ou macros (`define`, `case`, etc.) et lui fournir un environnement global contenant les variables `evaluate`, `evlis`, etc.

1.10 Exercices

Exercice 1.1 : Modifiez la fonction `evaluate` en un traceur. Tous les appels fonctionnels verront leurs arguments et leur résultat imprimés. On songera avec fruit à l'extension de ce traceur rudimentaire en un metteur au point pas à pas susceptible de modifier le cheminement de l'exécution du programme ainsi contrôlé.

Exercice 1.2 : L'évaluation par `evlis` d'une liste contenant une seule expression nécessite une récursion un peu inutile. Trouvez-la puis éliminez-la.

Exercice 1.3 : La fonction `extend` est maintenant définie comme suit :

```
(define (extend env names values)
  (cons (cons names values) env) )
```

Définissez les fonctions associées `lookup` et `update!` ; comparez-les avec les précédentes définitions.

Exercice 1.4 : Une autre implantation de liaison superficielle est suggérée par le concept de *rack* [SS80]. À chaque symbole est associée non plus un champ pour contenir la valeur de la variable de même nom, mais une pile. À tout instant la valeur de la variable est la valeur qui se trouve au sommet de la pile des valeurs associée. Réécrivez les fonctions `s.make-function`, `s.lookup` et `s.update!` en profitant de cette nouvelle représentation.

Exercice 1.5 : La définition de la primitive `<` est fausse ! Elle retourne en effet un booléen de l'implantation au lieu d'un booléen du langage en cours de définition. Corrigez ce défaut.

Exercice 1.6 : Définissez la fonction `list`.

Exercice 1.7 : Pour les amoureux des continuations, définissez `call/cc`.

Exercice 1.8 : Définissez `apply`.

Exercice 1.9 : Définissez une fonction `end` permettant de sortir proprement de l'interprète présenté dans ce chapitre.

Exercice 1.10 : Mesurez le rapport de vitesse entre Scheme et `evaluate` puis entre `evaluate` et `evaluate` interprétée par `evaluate`.

Exercice 1.11 : La séquence `begin` avait été définie **[☞ p. 10]** par le biais de `lambda` mais usait de `gensym` afin d'éviter une éventuelle capture. Redéfinissez `begin` dans le même esprit sans user de `gensym`.

1.11 Lectures recommandées

Toutes les références aux interprètes mentionnés en début de chapitre sont intéressantes mais, s'il ne faut lire que quelques documents, les plus enrichissants sont probablement :

- parmi les « *λ-papers* » : [SS78a] ;
- le plus court article présentant un évaluateur qui ait jamais été écrit : [McC78b] ;
- pour déjà goûter un brin de formalisme non pédant : [Rey72] ;
- enfin, pour connaître la genèse, [MAE+62].