La programmation fonctionnelle et le langage Lisp:

Scheme lave plus blanc

Henri Garreta *Janvier 99*

1. Introduction

1.1. Lisp et Scheme

Le langage Lisp dérive des travaux de Alonzo Church sur le λ-calcul et de ceux de John McCarthy sur les fonctions récursives. A l'origine de Lisp il y a l'idée de définir un langage de programmation qui soit en même temps un formalisme puissant pour la définition et la transformation des fonctions mathématiques.

En plus d'opérer avec des quantités numériques, Lisp a été conçu pour manipuler des atomes abstraits, les symboles, et des combinaisons arbitrairement complexes d'atomes, les listes. Les listes ont donné son nom au langage (LISt Processing language), lequel est rapidement devenu un outil privilégié pour l'écriture d'applications en Intelligence Artificielle, sans doute à cause de son caractère foncièrement symbolique ; à l'heure actuelle, Lisp reste le plus utilisé des langages de l'I.A.

La première version de Lisp a été réalisée en 1958 par John McCarthy, au Massachusetts Institute of Technology, avec un groupe d'étudiants et de collègues. Lisp partage donc avec Fortran l'honneur d'être les plus anciens langages de programmation encore en usage. Mais Lisp a présenté depuis ses débuts un ensemble de traits résolument modernes. Citons :

- la généralité et la souplesse des données manipulées, qui sont des expressions formelles construites à l'aide de symboles, organisés en structures arborescentes de complexité quelconque,
- le caractère fonctionnel du langage, articulé autour de la notion de fonction, implantée sans restriction inutile. L'appel de fonction est l'opération élémentaire la plus naturelle dans laquelle se réduit la résolution d'un problème. L'emploi de la récursivité est banalisé, y compris pour implanter les processus itératifs,
- la simplicité et l'uniformité de la syntaxe, et l'absence du fossé qui, dans d'autres langages, sépare les éléments passifs (les données) des éléments actifs (les programmes). En Lisp, les programmes obéissent à la même syntaxe que les données qu'ils manipulent, une donnée pouvant devenir à tout moment le programme actif.

Le nombre et la disparité des versions de Lisp en circulation interdisent d'envisager un exposé de longueur raisonnable qui rendrait compte des concepts communs et des caractéristiques de chaque dialecte. Il faut donc choisir un Lisp particulier, s'y tenir, et s'en remettre au lecteur pour ce qui est de dégager les principes universels.

Nous avons choisi Scheme, car c'est un dialecte simple et moderne, résultat de quinze années d'études et recherches *avec* et *sur* les principales versions de Lisp. Propre, débarrassé des reliques qui encombrent des dialectes plus anciens, il a considérablement influencé la définition de Common Lisp, le standard de fait de Lisp. Principalement destiné à la recherche et à l'enseignement, Scheme a la simplicité, la régularité et la rigueur qui font bien défaut à d'autres dialectes de Lisp.

Etant entendu que l'objet de ce cours est le dialecte Scheme du langage Lisp, nous dirons indistinctement Lisp ou Scheme pour le nommer, en essayant toutefois d'utiliser plutôt Lisp dans l'exposé d'éléments communs à beaucoup de dialectes, et plutôt Scheme lors de la discussion d'aspects plus caractéristiques de ce dialecte.

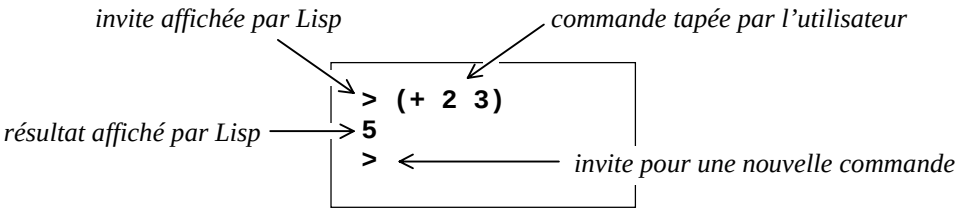
1.2. L'interprète de Lisp

Lisp est un langage fonctionnel : le langage tout entier est construit autour de la notion de fonction. L'opération de base est l'évaluation d'une expression, qui est presque toujours l'application d'une fonction à une liste d'arguments. Ce qui se fait dans d'autres langages par des séquences d'instructions, se fait en Lisp par des compositions de fonctions. Programmer en Lisp, c'est ajouter ses propres fonctions à celles que fournit le système.

Une machine Lisp en état de marche se présente comme un automate qui répète inlassablement les trois opérations qui forment ce qu'on appelle la boucle principale ou niveau supérieur (*Top level* ou *read-eval-print loop*) :

- *read* : lecture d'une commande tapée par l'utilisateur, généralement précédée de l'affichage d'une invite. Cette commande doit obéir à la syntaxe des S-expressions, expliquée au chapitre suivant,
- *eval* : évaluation de l'expression lue, selon les règles expliquées plus loin,
- *print* : écriture du résultat, qui est encore une S-expression[1].

Exemple :



Notez bien que la machine Lisp n'entend et ne parle qu'une langue, celle des S-expressions ; toute construction qui n'est pas une S-expression correcte sera rejetée.

En résumé, apprendre à programmer en Lisp, c'est :

- apprendre la syntaxe des S-expressions,
- apprendre les règles de l'évaluation,
- apprendre l'existence et le fonctionnement d'un petit nombre de fonctions prédéfinies (livrées avec le système Lisp)

et, bien entendu, acquérir la méthodologie, l'esprit « programmation fonctionnelle ». Pour cela, un seul moyen : pratiquer, écrire des programmes en Lisp, faire le plus grand nombre d'exercices parmi ceux proposés dans les planches de TD ou dans les livres cités en référence.

☺

[1] Parfois, une expression n'est évaluée que pour les effets secondaires qu'elle produit, sa valeur est sans intérêt. Certains systèmes Lisp, dont *gsi Scheme* (celui que nous utilisons) n'affichent pas de tels résultats sans intérêt.

2. Écrire

La syntaxe des S-expressions est incroyablement simple. On a :

- des atomes, c'est-à-dire :
  - des constantes (généralement des nombres),
  - des identificateurs,
- des listes, faites d'atomes et d'autres listes.

2.1. Atomes

2.1.1. Nombres

En Lisp les constantes numériques (les nombres entiers et les nombres flottants) s'écrivent comme dans les autres langages de programmation. Mais ils y sont réalisés avec beaucoup plus de soin. Cela se voit principalement à trois endroits :

1. Les entiers sont en précision infinie : ils ont autant de chiffres qu'il leur en faut.

Exemple : un exercice auquel on n'échappe pas quand on apprend un langage de programmation est l'écriture du programme qui calcule la factorielle *n*! d'un entier *n*. On est alors déçu devant l'incapacité de la machine à représenter le résultat, dès que *n* n'est pas ridiculement petit (7 ou 12, selon la « taille des entiers » du langage utilisé). Alors qu'en Lisp, les entiers n'étant pas limités en taille, on fait de belles expériences :

```
> (factorielle 100)
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
>
```

2. Lisp connaît les nombres rationnels (c'est-à-dire, les fractions). Cela lui permet de faire des calculs *exacts* avec des nombres non entiers :

```
> (/ 1 3)
1/3
>
```

Dans d'autres langages, dès que les nombres à manipuler ne sont pas entiers, on doit utiliser les nombres flottants, qui sont une représentation approchée, très imprécise, des nombres décimaux (songez qu'un nombre aussi simple que 0.1 ne peut pas être représenté exactement !). Cette imprécision est propagée et augmentée par les calculs. Ainsi, par exemple, des algorithmes courants pour la résolution de systèmes d'équations linéaires sont inapplicables, uniquement parce que les suites d'opérations qu'ils mettent en œuvre amplifient les erreurs au point d'enlever toute signification aux résultats trouvés.

3. En Lisp les types numériques sont réellement vus comme des ensembles inclus les uns dans les autres[1], comme en mathématiques : $\mathbf{Z} \subset \mathbf{Q} \subset \mathbf{R} \subset \mathbf{C}$.

Dans d'autres langages on a les entiers d'un côté, les flottants de l'autre. Ces ensembles sont définis par des représentations internes différentes, ils sont donc disjoints. Bien sûr, des conversions plus ou moins automatiques associent à un entier un flottant qui lui ressemble, ou à un flottant un entier qui lui ressemble plus ou moins. Mais les calculs se font toujours dans l'un ou l'autre ensemble, le programmer ne peut ignorer cette question.

---

1 Cela est une conséquence du typage dynamique de Lisp, que nous commenterons plus loin.

En Lisp, une valeur entière est réellement une valeur rationnelle, qui est elle-même une valeur réelle, etc. Si une fonction a été écrite pour opérer sur des réels, on pourra lui passer aussi bien des entiers que des rationnels ou des réels. De même, si une opération sur les rationnels donne un résultat entier, nous obtiendrons un entier, non un rationnel de dénominateur 1.

```
> (+ 1/3 2/3)
1
>
```

2.1.2. Caractères et chaînes de caractères

Ecriture des caractères (ici : « A », « i », « 3 » et « \ »)

```
#\A      #\i      #\3      #\\
```

Ecriture des chaînes (ici : « Bonjour », « J'ai dit "bonjour" », « A » et la chaîne vide :

```
"Bonjour"    "J'ai dit \"bonjour\""    "A"    ""
```

2.1.3. Booléens

La valeur « vrai » se représente par `#t`, la valeur « faux » par `#f`.

Certaines formes et fonctions prédéfinies requièrent une expression qui est interprétée comme une condition, une « question » dont la réponse est oui ou non. L'idéal est de mettre une expression booléenne (dont la valeur est `#t` ou `#f`), mais le langage admet n'importe quelle sorte d'expression : toute valeur autre que `#f` sera tenue pour vraie.

2.1.4. Identificateurs

En Lisp, on a une grande liberté pour la formation des identificateurs. Toute suite de caractères qui ne contient pas de séparateur (c'est-à-dire un des caractères[1] ' ', '`'`', '`;`', '`(`' et '`)`') et qui n'est pas une constante (un nombre, un booléen, une chaîne, etc.) est un identificateur. Exemple :

```
x
vitesse_du_vent
*stop*
240Volts
+
<o>_/\_decoratif_hein?_/\_<o>
```

En revanche, ne sont pas des identificateurs :

`123`	*c'est une constante*
`"Bonjour"`	*idem*
`Ceci est un symbole`	*contient des blancs*
`(x)`	*contient des parenthèses*

En Lisp il n'y a pas besoin de déclarer les identificateurs préalablement à leur utilisation.

Ce « détail » révèle l'importante différence qui existe entre les identificateurs de Lisp et ceux d'autres langages. En Fortran, Pascal, C, etc., un identificateur est toujours un renvoi à une autre entité (constante, type, variable…). En Lisp, un identificateur ne renvoie pas forcément à une autre entité. Bien sûr, il peut être le nom d'une variable mais, tout d'abord, un identificateur ne représente que lui-même, en tant que donnée primitive, susceptible d'être lue, écrite, comparée, etc. Cela fonde le caractère symbolique du langage, les identificateurs étant les plus éminents des symboles.

---

1 Notez que, malgré son rôle important, le point '`.`' n'est pas considéré comme un séparateur. Ainsi, `ab.cd` est un identificateur valide. Conseil : évitez d'utiliser de tels identificateurs.

Dans les identificateurs les majuscules et les minuscules sont confondues ; par exemple, les identificateurs `abc`, `Abc` et `ABC` sont équivalents. Lisp mémorise tous les identificateurs rencontrés dans une table, dans laquelle chaque identificateur figure une et une seule fois. Ainsi, la propriété suivante est garantie :

> PROPRIETE : unicité de la représentation d'un identificateur. Les diverses occurrences d'identificateurs ayant la même expression écrite – aux équivalences majuscules/minuscules près – ont des représentations internes indiscernables à tout point de vue.

2.2. Structures

2.2.1. Paires pointées

Maintenant que nous savons ce qu'est un atome, nous pouvons donner la définition complète des S-expressions :

> DEFINITION : S-expression, paire pointée. Les expressions symboliques ou S-expressions sont les expressions définies par les deux règles suivantes :
>
> - les atomes sont des S-expressions,
> - si α et β sont des S-expressions, alors **(** α **.** β **)** est une S-expression.
>
> Une S-expression de la forme (α . β) s'appelle une *paire pointée*.

Par exemple, chacune des quatre lignes ci-dessous définit *une* paire pointée (dont, éventuellement, les composantes sont à leur tour des paires pointées) :

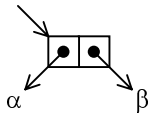
```
( A . B )
(Mieux . (vaut . (tenir . (que . courir))))
(((((1 . 2) . 3) . 4) . 5) . oh!)
(1 . (2 . (3 . (4 . (5 . oh!)))))
```

Notez bien qu'aucune des S-expressions précédentes n'équivaut à une expression comme

```
1 . 2 . 3 . 4 . etc.
```

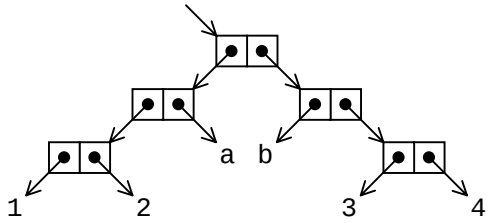
Cela n'est pas une S-expression, mais tout simplement une expression erronée, car les parenthèses ne sont pas facultatives dans les S-expressions : il n'y a ici aucune sorte de « d'associativité » qui dispenserait d'écrire certaines parenthèses.

Si l'utilisateur a le droit d'écrire des paires pointées, c'est que la machine est capable de les lire et d'en construire une représentation interne. On peut donner une idée de cette représentation en dessinant les diagrammes suivants : si α et β sont des S-expressions, alors la représentation interne de l'expression **(** α **.** β **)** ressemble à ceci :



Ce diagramme veut montrer que, pour autant que le problème de la représentation interne des deux S-expressions α et β ait été résolu, alors la représentation interne de la paire pointée **(** α **.** β **)** consiste simplement en une double case, appelée *cellule-cons*, formée de deux pointeurs : un vers la représentation de α, l'autre vers celle de β.

En réitérant ce procédé, on obtient la représentation interne de S-expressions plus complexes. Par exemple, celle de `(((1 . 2) . a) . (b . (3 . 4)))` :



2.2.2. Listes

Ayant défini les atomes et les paires pointées, la définition des S-expressions est, strictement parlant, terminée. Cependant, il se trouve que certaines S-expressions d'un type particulier sont très fréquemment utilisées, à tel point qu'elles ont reçu une notation spéciale, simplifiée :

DEFINITION : liste.

- Il existe un (et un seul) atome qui est une liste, on l'appelle la liste vide,
- si α_0 est une S-expression et Λ une liste, alors la paire pointée $(\alpha_0 \ . \ \Lambda)$ est une liste.

DEFINITION : notation des listes.

- La liste vide se note $()$
- si la liste Λ peut se noter aussi $(\alpha_1 \ \alpha_2 \ \dots \ \alpha_n)$ alors la liste $(\alpha_0 \ . \ \Lambda)$ peut se noter aussi $(\alpha_0 \ \alpha_1 \ \alpha_2 \ \dots \ \alpha_n)$

Plus simplement, ces deux définitions affirment que toute liste est de la forme :

$$(\alpha_1 \ . \ (\alpha_2 \ . \ \dots \ . \ (\alpha_n \ . \ ()) \ \dots))$$

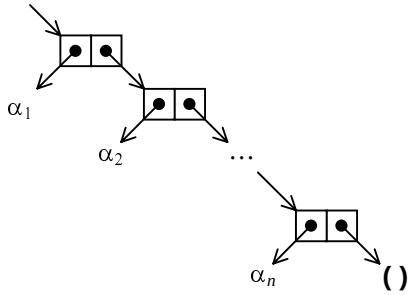
et qu'elle peut alors se noter :

$$(\alpha_1 \ \alpha_2 \ \dots \ \alpha_n)$$

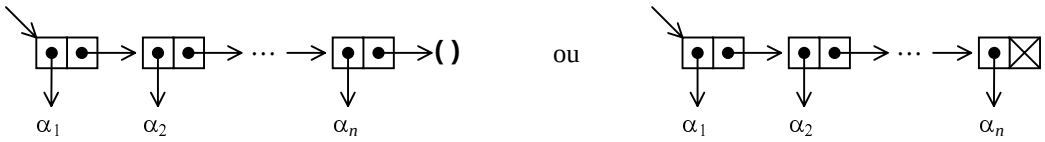
Exemples :

```
()
(1 . ())                    ou          (1)
(2 . (1 . ()))              ou          (2 1)
(3 . (2 . (1 . ())))        ou          (3 2 1)      etc.
(Mieux vaut tenir que courir)
(3 fois 5 égale 15)
(((Ceci est) (une liste)) (de 2 éléments))
```

De sa définition, nous pouvons déduire une représentation interne de la liste $(\alpha_1 \ \alpha_2 \ \dots \ \alpha_n)$:



En fait, pour représenter les listes, on utilise plutôt des diagrammes tels que :



dont la dissymétrie veut rappeler que, dans une liste, les deux pointeurs de chaque cellule n'ont pas tout à fait le même statut : le premier est l'adresse d'une S-expression *quelconque*, tandis que le deuxième est obligatoirement l'adresse *d'une liste*.

2.2.3. Listes pointées

Considérons les deux S-expressions suivantes :

```
(1 . (2 . (3 . (4 . (5 . ()))))))
(1 . (2 . (3 . (4 . 5))))
```

Elles ne sont pas très différentes ; cependant, seule la première est une liste et a droit à une notation spéciale. En fait, Lisp tolère, et utilise, une notation « hybride » :

DEFINITION : liste pointée. La S-expression

$$(\alpha_1 \ . \ (\alpha_2 \ . \ \ldots \ . \ (\alpha_{n-1} \ . \ \alpha_n) \ \ldots))$$

peut se noter aussi

$$(\alpha_1 \ \alpha_2 \ \ldots \ \alpha_{n-1} \ . \ \alpha_n)$$

Ainsi, les deux expressions précédentes seront écrites par Lisp sous la forme :

```
(1 2 3 4 5)
(1 2 3 4 . 5)
```

Nous venons de terminer la description syntaxique des données manipulés par Lisp. Or, nous avons dit qu'il n'y avait pas de distinction entre les données et les programmes. Par conséquent, nous venons de décrire la totalité de la syntaxe. Comparée à celle d'autres langages, la syntaxe de Lisp s'avère d'une simplicité stupéfiante. Hélas, la pratique montre que cette simplicité se paye quelquefois par une certaine illisibilité des programmes (explosion du nombre de parenthèses, etc.).

Quoi qu'il en soit, il semble que l'absence dans Lisp du « sucre syntaxique » dont d'autres langages regorgent soit la fierté des authentiques lispiens car, comme l'affirme un de leurs vieux proverbes[1] :

Syntactic sugar causes cancer of the semicolon

☺ ☹

[1] « Le sucre syntaxique donne le cancer du point-virgule »

3. Évaluer

Nous savons donc écrire des S-expressions. Il nous faut maintenant apprendre comment Lisp les évalue.

L'évaluation d'une S-expression est définie dans les cas suivants :

- évaluation d'un atome { une constante,, , un identificateur,
- évaluation d'une liste,
- évaluation d'une expression quotée,
- évaluation d'une forme spéciale.

D'une certaine manière, les premières situations font figure de cas général, tandis que les deux dernières apparaissent plutôt comme des dérogations à la règle commune.

3.1. Evaluation d'une constante ou d'un identificateur

DEFINITION : évaluation d'un atome. L'évaluation d'un atome α est définie de la manière suivante :

- si α est une constante, son évaluation fournit la valeur dont α est l'expression écrite,
- sinon (α est donc un identificateur) il est nécessaire que α soit le nom d'une variable couramment définie ; l'évaluation de α fournit alors la valeur courante de cette variable.

Une tentative d'évaluation d'un identificateur qui n'est pas le nom d'une variable définie au moment de l'évaluation provoque une erreur.

Exemples :

```
> 123
123
> "Bonjour"
"Bonjour"
> (define année 1998)
année
> année
1998
> mois
ERREUR : la variable mois est indéfinie
>
```

3.2. Évaluation d'une liste

DEFINITION : évaluation d'une liste. Si $expr_0$ n'est pas le mot-clé d'une forme spéciale, alors l'évaluation de la liste

$$(expr_0 \ \ expr_1 \ \ expr_2 \ \ \ldots \ \ expr_n)$$

commence par l'évaluation de ses éléments $expr_0$, $expr_1$, $expr_2$, … $expr_n$.

Notons ν_i le résultat de l'évaluation de $expr_i$. Il est nécessaire que ν_0 soit une fonction, et le résultat de l'évaluation de la liste est le résultat rendu par la fonction ν_0 quand on l'appelle avec les arguments ν_1 ν_2 … ν_n.

Une erreur survient si ν_0 n'est pas une fonction ou si n n'est pas le nombre d'arguments requis par cette fonction.

Exemples :

```
> (2 . 3)
ERREUR : appel mal formé
> (1 2 3)
ERREUR : 1 n'est pas une fonction
> (+ 2 3)
5
> (* (+ 2 3) (- 7 3))
20
> (define x 2)
x
> (define y 3)
y
> (define ope +)
ope
> (ope x y)
5
>
```

En Lisp, les opérateurs et les fonctions utilisent donc la *notation préfixée parenthésée*, y compris pour les opérateurs les plus courants. Pour le prix (modique ?) d'une augmentation certaine du nombre de parenthèses, cette notation offre une syntaxe extrêmement simple et régulière. Du point de vue du programmeur ou de l'utilisateur, tout se passe bien dès qu'on a pris l'habitude d'écrire

(sin x) à la place de *sin(x)*

et

(+ a b) à la place de *a + b*

3.3. Quote

DEFINITION : quote. L'expression

`(quote` *expr*`)`

peut s'écrire aussi

`'`*expr*

On dit que `'`*expr* est une expression quotée[1].

DEFINITION : évaluation d'une expression quotée. Le résultat de l'évaluation de

`(quote` *expr*`)` *ou* `'`*expr*

est

expr

Exemples :

```
> salut
ERREUR : la variable salut est indéfinie
> 'salut
salut
> (1 2 3 4 5)
ERREUR : 1 n'est pas une fonction
> '(1 2 3 4 5)
(1 2 3 4 5)
>
```

La forme *quote* apparaît comme le moyen de *suspendre* l'évaluation d'une expression.

Elle est indispensable : d'après la définition de l'évaluation, sans quote il ne serait pas possible de manipuler ni les symboles (tout symbole serait pris pour un nom de variable) ni les listes (toute liste serait prise pour un appel de fonction). Quote indique que l'expression qu'elle précède est à prendre

[1] « Quote » et « quoté » ne sont pas des mots de la langue française ; ils appartiennent au *lispien populaire*.

littéralement, comme une donnée et non pas comme une commande. Sa nécessité découle du fait qu'en Lisp syntaxe des données est la même que celle des instructions :

```
> (+ 2 3)              ; une instruction : « additionnez 2 et 3 »
5
> '(+ 2 3)             ; une donnée : la liste des trois atomes +, 2 et 3
(+ 2 3)
> ''(+ 2 3)            ; pour voir…
'(+ 2 3))              ; ou, sur certains systèmes :   (quote (+ 2 3))
>
```

3.4. Formes spéciales

DEFINITION : forme spéciale. Une forme spéciale est une construction qui suit la syntaxe des appels de fonctions (c'est-à-dire : c'est une liste) mais dont l'évaluation n'obéit pas à la règle générale de l'évaluation des listes.

Le premier élément d'une forme spéciale est un mot-clé réservé, son nom ; cela est le seul moyen de reconnaissance de la forme spéciale. Les règles d'évaluation de chaque forme spéciale dépendent de la forme, et en constituent la définition. En tout cas, contrairement aux appels de fonctions, l'évaluation d'une forme spéciale *ne commence pas nécessairement* par l'évaluation de chaque arguments de l'appel.

Les formes spéciales de Scheme sont expliquées dans la suite de ce document. Nous avons déjà utilisé *define* pour définir et initialiser des variables :

```
> nombre
ERREUR : la variable nombre est indéfinie
> (define nombre 123)
nombre
> nombre
123
>
```

Il est clair que *define* doit être une forme spéciale : si c'était une fonction, l'évaluation de *(define nombre 123)* provoquerait l'évaluation préalable de la variable *nombre*, ce qui entraînerait une erreur.

☺☺

4. Programmer

On l'a dit : Lisp est un *évaluateur*. L'utilisateur compose une commande, presque toujours l'appel d'une fonction avec une liste d'arguments, la machine l'évalue et affiche le résultat. Dans ces conditions, qu'est-ce que « écrire des programmes ? »

Programmer en Lisp c'est définir de nouvelles variables et fonctions, surtout des fonctions. Cela se fait avec la forme spéciale *define*, dont l'évaluation produit un résultat sans importance, mais un « effet secondaire » extrêmement intéressant : l'ajout d'une nouvelle variable ou fonction à l'ensemble des variables et fonctions déjà connues.

4.1. Définition de variables et de fonctions

DEFINITION : définition des variables, version naïve. La forme *define*, dans le cas de la définition d'une variable, s'utilise de la manière suivante :

(define *nom expression*)

nom doit être un identificateur. L'évaluation de cette forme crée une variable, associée au *nom* donné, et lui affecte la valeur de l'*expression* donnée.

Define rend comme résultat le nom de la variable définie ; c'est sans utilité.

Exemple:

```
> (define x (+ 2 3))
x
> x
5
>
```

DEFINITION : définition des fonctions, version naïve. La forme *define*, dans le cas de la définition d'une fonction, s'utilise de la manière suivante :

(define (*nom arg-for*$_1$ … *arg-for*$_k$) *expr*$_1$ … *expr*$_n$)

nom, *arg-for*$_1$, … *arg-for*$_k$ sont des identificateurs.

On dit que *nom* est le nom de la fonction et que *arg-for*$_1$, … *arg-for*$_k$ sont ses arguments formels ; *expr*$_1$, … *expr*$_n$ constituent le corps de la fonction.

Dans le corps de la fonction, outre les variables couramment définies, peuvent apparaître les arguments formels *arg-for*$_1$, … *arg-for*$_k$.

Exemple :

```
> (define (carre x) (* x x))
carre
>
```

DEFINITION : appel d'une fonction, version naïve. Une fonction définie comme ci-dessus peut alors être appelée, par exemple à travers l'expression :

$$(nom\ \ arg\text{-}eff_1\ \ \ldots\ \ arg\text{-}eff_k)$$

où $arg\text{-}eff_1$ … $arg\text{-}eff_k$ (les *arguments effectifs* de l'appel) sont des S-expressions. L'évaluation d'un tel appel se déroule de la manière suivante :

- les arguments effectifs $arg\text{-}eff_1$ … $arg\text{-}eff_k$ sont évalués et leurs valeurs affectées aux identificateurs $arg\text{-}for_1$ … $arg\text{-}for_k$, comme s'il s'agissait de variables nouvellement définies,
- $expr_1$ … $expr_n$ sont alors successivement évaluées, et la valeur de la dernière de ces expressions, $expr_n$, est rendue à titre de résultat de l'application.

L'association des valeurs des arguments effectifs $arg\text{-}eff_i$ aux arguments formels $arg\text{-}for_i$ n'est connue qu'à l'intérieur du corps de la fonction.

Exemple :

```
> (carre (+ 4 5))
81
>
```

La correspondance entre les arguments formels et les arguments effectifs est *positionnelle* : le premier argument formel reçoit la valeur du premier argument effectif, le second argument formel celle du second argument effectif, etc.

Les parenthèses autour du nom et des arguments de la fonction doivent apparaître, dans la définition et dans l'appel de la fonction, même lorsque celle-ci est sans arguments. Sinon on tombe dans l'autre cas d'emploi de *define* : la définition d'une variable. Voici, par exemple, la définition d'une variable *deux* et d'une fonction sans argument, *trois* :

```
> (define deux 2)
deux
> (define (trois) 3)
trois
> deux                              ; que vaut deux ?
2
> (deux)                            ; et un appel de la fonction deux ?
ERREUR : deux n'est pas une fonction
> trois                             ; que vaut trois ?
#[procedure #x263090]               ; façon de dire « c'est une fonction »
> (trois)                           ; et un appel de la fonction trois ?
3
>
```

4.2. Structures de contrôle

4.2.1. Prédicats

DEFINITION : prédicat. Au sens propre, un prédicat est une fonction à valeurs dans l'ensemble {**#t**, **#f**}.

Par extension, toute expression peut occuper la place d'un prédicat. Elle sera tenue pour fausse si sa valeur est **#f** ; elle sera tenue pour vraie dans *tous les autres cas*.

En Scheme, la plupart des noms des prédicats prédéfinis se terminent par le caractère '?', pour rappeler qu'un prédicat correspond à une question, à laquelle l'évaluateur répond par « oui » ou par « non ». Parmi les prédicats les plus usuels :

Test de type

Tous ces prédicats prennent un seul argument, qui est une S-expression tout à fait générale, et sont vrais si et seulement si…

(null? *expr***)** la valeur de *expr* est la liste vide

(pair? *expr***)** la valeur de *expr* est une paire pointée
/ la valeur de *expr* n'est pas un atome

(symbol? *expr***)** la valeur de *expr* est un identificateur

(number? *expr***)** la valeur de *expr* est un nombre

(real? *expr***)** la valeur de *expr* est un nombre réel

(rational? *expr***)** la valeur de *expr* est un nombre rationnel

(integer? *expr***)** la valeur de *expr* est un nombre entier

Comparaison numérique

(= $expr_1$ $expr_2$**)** **(<** $expr_1$ $expr_2$**)** **(<=** $expr_1$ $expr_2$**)** **(>** $expr_1$ $expr_2$**)** **(>=** $expr_1$ $expr_2$**)**
la valeur de $expr_1$ est égale [resp. inférieure, inférieure ou égale, etc.] à celle de $expr_2$. Ces valeurs doivent être des nombres

(zero? *expr***)** la valeur de *expr* est nulle

(even? *expr***)** [resp. **(odd?** *expr***)**]
la valeur de *expr* est paire [resp. impaire]

4.2.2. « If » et « cond »

DEFINITION : forme spéciale *if*. Les syntaxes de la forme spéciale *if* sont les suivantes :

(if *prédicat* $expression_1$ $expression_2$**)**

(if *prédicat* $expression_1$**)**

Évaluation : *prédicat* est évalué :

- si sa valeur est vraie, alors $expression_1$ est évaluée, et sa valeur est celle que rend la forme *if* ; $expression_2$, si elle est présente, n'est pas évaluée.
- sinon :
  - si $expression_2$ est présente, alors elle est évaluée et sa valeur est celle que rend la forme *if*
  - sinon, la valeur rendue par la forme *if* est indéterminée.

Exemple :

```
> (define x -5)
x
> (if (>= x 0) x (- x))
5
>
```

DEFINITION : forme spéciale *cond*. La syntaxe de la forme spéciale *cond* est la suivante :

(cond **(***prédicat*$_1$ *expr*$_{1,1}$ *expr*$_{1,2}$ … **)**
(*prédicat*$_2$ *expr*$_{2,1}$ *expr*$_{2,2}$ … **)**
…
(*prédicat*$_k$ *expr*$_{k,1}$ *expr*$_{k,2}$ … **))**

Évaluation : *prédicat*$_1$ est évalué, puis *prédicat*$_2$ etc., tant que les valeurs de ces prédicats sont fausses. Soit *j* un entier tel que :

- les valeurs de *prédicat*$_1$, *prédicat*$_2$, … *prédicat*$_{j-1}$ sont fausses,
- la valeur de *prédicat*$_j$ est vraie

alors

- *prédicat*$_{j+1}$, … *prédicat*$_k$ ne sont pas évalués,
- *expr*$_{j,1}$, *expr*$_{j,2}$, etc., sont évaluées ; la valeur de la dernière de ces expressions est celle que rend la forme *cond*.

En définitive, les seules expressions qui auront été évaluées sont celles qui apparaissent soulignées ci-après (toutes les autres expressions auront été ignorées) :

(cond **(***prédicat*$_1$ *expr*$_{1,1}$ *expr*$_{1,2}$ … **)**
(*prédicat2* *expr*$_{2,1}$ *expr*$_{2,2}$ … **)**
…
(*prédicatj* *expr*$_{j,1}$ *expr*$_{j,2}$ … *expr*$_{j,m}$**)**
…
(*prédicatk* *expr*$_{k,1}$ *expr*$_{k,2}$ … **))**

Exemple :

```
> (define (examiner a)
       (cond   ((< a -1000) 'trop-petit)
               ((> a 1000) 'trop-grand)
               ((= a 0) 'nul)))
```

Si aucun des prédicats *prédicat*$_j$ n'a une valeur vraie, alors aucune expression n'est évaluée et la valeur de *cond* est indéfinie. En général, on évite cette situation en ajoutant une dernière condition toujours vraie : par exemple, `#t` :

```
> (define (examiner a)
       (cond   ((< a -1000) 'trop-petit)
               ((> a 1000) 'trop-grand)
               ((= a 0) 'nul)
               (#t 'rien-a-dire)))
```

Scheme fournit le mot-clé *else*, prédéfini, valant `#t`, qui permet d'écrire le programme précédent sous la forme (plus élégante ?) :

```
> (define (examiner a)
       (cond   ((< a -1000) 'trop-petit)
               ((> a 1000) 'trop-grand)
               ((= a 0) 'nul)
               (else 'rien-a-dire)))
```

4.2.3. Deux autres formes conditionnelles

Les connecteurs logiques *and* et *or* ressemblent à de simples opérateurs, mais en Scheme ce sont des formes spéciales aussi complexes que *if* ou *cond* :

DEFINITION : forme spéciale *and*. La syntaxe de la forme spéciale *and* est la suivante :

(and $expr_1$ … $expr_n$**)**

Évaluation : $expr_1$, $expr_2$ … sont évaluées, successivement et dans cet ordre, tant que les valeurs de ces expressions sont vraies (c'est à dire, différentes de `#f`).

Soit *j* un entier tel que les valeurs de $expr_1$, … $expr_{j-1}$ sont vraies et celle de $expr_j$ est fausse. Alors :

- $expr_{j+1}$, … $expr_n$ ne sont pas évaluées,
- la valeur rendue par la forme *and* est `#f`.

Si toutes les $expr_i$ sont vraies, alors la valeur de la forme *and* est celle de $expr_n$.

DEFINITION : forme spéciale *or*. La syntaxe de la forme spéciale *or* est la suivante :

(or $expr_1$ … $expr_n$**)**

Évaluation : $expr_1$, $expr_2$ … sont évaluées successivement et dans cet ordre, tant que les valeurs de ces expressions sont fausses.

Soit *j* un entier tel que les valeurs de $expr_1$, … $expr_{j-1}$ sont fausses et celle de $expr_j$ est vraie. Alors :

- $expr_{j+1}$, … $expr_n$ ne sont pas évaluées,
- la valeur rendue par la forme *or* est celle de $expr_j$.

Si toutes les $expr_i$ sont fausses, alors la valeur de la forme *or* est `#f`.

4.2.4. Les effets de bord et la forme begin

Nous disons qu'une expression est *pure* lorsque son évaluation ne modifie en rien l'état du système : chaque variable qui existait avant l'évaluation existe après et a la même valeur ; de plus, aucune donnée n'a été transmise, enregistrée, affichée ou imprimée, aucun appareil n'a été mis en marche ni éteint, etc. Bref, d'une expression pure on peut dire qu'elle a une valeur, mais *aucun effet*.

Lorsqu'une expression n'est pas pure on dit qu'elle a un *effet de bord*. L'évaluation d'une expression à effet de bord rend une valeur, parfois sans intérêt, et de plus produit un changement dans l'état du système. Exemple :

```
> (define x 50)
x
>
```

La valeur rendue par l'évaluation, l'identificateur *x*, est sas intérêt. C'est l'effet de bord qui nous intéresse : la création d'une nouvelle variable. Autre exemple, un affichage :

```
> (display "Bonjour")
Bonjour7
>
```

Effet de bord : affichage d'une chaîne de caractères. La valeur rendue, 7 (le nombre de caractères affichés) est sans intérêt, et même gênant.

DEFINITION : forme spéciale *begin*. La syntaxe de la forme spéciale *begin* est :

(begin $expr_1$ … $expr_n$**)**

Évaluation : $expr_1$ … $expr_n$ sont évaluées, successivement et dans cet ordre. La valeur de la dernière est la valeur rendue par la forme *begin*.

Il est clair que $expr_1$ … $expr_{n-1}$ ne sont évaluées qu'en vue de leur effet de bord. Si elles n'en ont pas, on perd notre temps, car leur valeur n'est pas récupérée.

4.3. La récursivité

Une fonction peut s'appeler elle-même. Ce « détail » est un élément fondamental de la notion de fonction en Lisp. Exemple classique, le calcul de la factorielle d'un nombre, à partir de la définition mathématique récursive (on suppose $n \geq 0$) :

$$n! = \begin{cases} 1, \text{si } n \leq 1 \\ \\ n \times (n-1)!, \text{sinon} \end{cases}$$

Cela donne :

```
> (define (fact n)
     (if (<= n 1)
          1
          (* n (fact (- n 1)))))
fact
> (fact 36)
371993326789901217467999448150835200000000
>
```

Il est en général difficile de « dérouler » l'appel d'une fonction récursive. On peut toutefois le faire dans un cas aussi simple que celui de la factorielle : un appel comme *(fact 5)* se résout en l'évaluation de *(* 5 (fact 4))*, une multiplication qui ne peut être faite avant de connaître la valeur de *(fact 4)*. Or *(fact 4)* à son tour se réduit à l'évaluation de *(* 4 (fact 3))*, multiplication qu'il faut mettre en attente jusqu'à l'obtention du résultat de *(fact 3)*, etc.

En définitive, l'appel de *(fact 5)* crée un certain nombre d'instances de la fonction *fact*, chacune restant bloquée sur un produit dont elle attend de connaître le second facteur :

```
(fact 5)
     (* 5 (fact 4))                ; 1ère instance de la fonction fact
          (* 4 (fact 3))           ; 2ème instance
               (* 3 (fact 2))      ; 3ème
                    (* 2 (fact 1)) ; 4ème
```

Cette partie du processus s'appelle la « descente » récursive : la fonction s'appelle elle-même, sans pouvoir conclure. Le nombre d'instances de fonctions commencées et non terminées ne fait qu'augmenter.

Or le dernier appel est terminal : *(fact 1)* se calcule sans rappel de la fonction. Se produit alors la « remontée », au cours de laquelle les calculs se font, le second facteur des multiplications mises en attente étant enfin connu :

```
(fact 5)
     (* 5 (fact 4))
          (* 4 (fact 3))
               (* 3 (fact 2))
                    (* 2 (fact 1))
                    (* 2 1)
               (* 3 2)
          (* 4 6)
     (* 5 24)
120
```

Cet exemple permet de comprendre quel est le coût, en termes de mémoire occupée, de la récursivité : il faut mémoriser chaque valeur de *n*, car cette valeur intervient comme facteur dans un produit qui ne pourra être effectué qu'au retour de l'appel récursif. L'encombrement est donc d'un argument formel *n* pour chaque instance de la fonction. Cet encombrement est notamment proportionnel au nombre de fois que la fonction s'appelle elle même (le nombre de niveaux dans la descente récursive).

Retenons ceci : parce que la fonction fait encore des opérations au retour de l'appel d'elle-même (ici, une multiplication), Lisp doit mémoriser la valeur de ses objets locaux (ici, l'argument formel, *n*).

4.3.1. Les boucles

Il existe des formes spéciales pour écrire des boucles (« tant que », « pour », etc.) dans les programmes Lisp. Elles sont parfois bien utiles, mais compliquées et inesthétiques. Nous les éviterons, dans le cadre de cette initiation à la programmation fonctionnelle, nous en tenant à une technique beaucoup plus basique pour l'écriture de boucles : l'utilisation de fonctions récursives.

Par exemple, donnons-nous le problème suivant, qui est évidemment soluble à l'aide d'une « boucle pour » : écrire les nombres entiers de 1 à *n* :

```
> (define (boucle i n)
     (if (<= i n)
          (begin
               (display i)
               (display " ")
               (boucle (+ i 1) n))))
boucle
> (boucle 1 10)
1 2 3 4 5 6 7 8 9 10 #[undefined]
>
```

4.3.2. La récursivité terminale

Lorsqu'une fonction s'appelle elle-même de telle manière qu'il n'y ait rien à faire au retour de l'appel récursif, on dit que l'on a un appel récursif terminal : l'appel récursif est l'opération qui *termine* la fonction en question. Par exemple, l'appel de *boucle* qui figure dans la fonction *boucle* précédente est terminal.

En Scheme, les appels récursifs terminaux sont optimisés. Puisqu'il n'y a aucune opération à faire au retour de l'appel, il n'est pas nécessaire de conserver la valeur des arguments formels et des variables locales de la fonction, avant d'activer une nouvelle instance de la fonction (d'où : gain d'espace mémoire) ; de plus, le retour de la fonction appelée devra se confondre avec le retour de la fonction appelante. Tout cela permet à Scheme de remplacer le mécanisme habituel de l'appel d'une fonction par des opérations beaucoup plus légères :

- *remplacement* des valeurs des arguments formels par les valeurs qu'auraient eu les arguments d'une nouvelle instance de la fonction, s'il y avait eu un appel normal,
- *branchement au début* de l'instance en cours de la fonction, plutôt que création d'une nouvelle instance de la fonction.

La récursivité terminale apparaît tout naturellement dans les fonctions qui sont l'écriture en Lisp d'un algorithme qui, dans la tête du programmeur, se présentait comme une boucle. Par exemple, dans la fonction *boucle* donnée plus haut, l'appel de *boucle* est un appel récursif terminal.

La récursivité terminale peut aussi être introduite, au prix d'un certain effort de réflexion, dans des fonctions qui au départ n'étaient pas récursives terminales. A titre d'exemple[1], cherchons à écrire la version récursive terminale de la factorielle.

---

[1] Précisons bien que la traduction de fonctions récursives (claires) en fonctions récursives terminales (généralement plus obscures) est une question qui ne nous intéressera pas beaucoup dans le cadre de ce cours. Pour nous, ici, la clarté sera toujours une qualité préférable à la rapidité ou au faible encombrement.

Pour cela, considérons la fonction à deux variables *fact2* : (*n*,*p*) ∅ *p* ∞ *n*! ; elle est plus générale que *fact*, car *fact*(*n*) = *fact2*(*n*,1). Remarquons que :

$$p \times n! = \begin{cases} p, \text{ si } n \geq 1 \\ (p \times n) \times (n-1)!, \text{ sinon} \end{cases}$$

d'où la fonction :

```
> (define (fact2 n p)
    (if (<= n 1)
        p
        (fact2 (- n 1) (* n p))))
fact2
> (fact2 36 1)
371993326789901217467999448150835200000000
>
```

L'ajout d'un deuxième argument nous a permis de faire les multiplications pendant la descente récursive. En réalité, il n'y a plus de descente récursive : la récursivité est devenue terminale, donc Scheme a optimisé notre fonction, la transformant en une boucle : la boucle qui, à chaque tour, remplace *n* par *n* – 1 et *p* par *n* × *p* jusqu'à ce que *n* ≤ 1 ; *p* contient alors le résultat cherché.

☺ ☹ ☹

5. Quelques fonctions prédéfinies

5.1. Traitement de listes

5.1.1. Consultation

Les trois fonctions suivantes forment le cœur du traitement des paires pointées et des listes. Toutes les autres opérations sur les listes s'y ramènent :

DEFINITION : fonctions *car* et *cdr*. Syntaxe :

(car *expr***)**
(cdr *expr***)**

Évaluation : si la valeur de *expr* est la paire pointée **(**α **.** β**)**, alors

- la valeur de **(car** *expr***)** est α,
- la valeur de **(cdr** *expr***)** est β.

Exemples :

```
> (car '(Pierre . Paul))
Pierre
> (cdr '(Pierre . Paul))
Paul
> (car '(1 2 3 4 5))
1
> (cdr '(1 2 3 4 5))
(2 3 4 5)
>
```

Restreintes aux listes, *car* et *cdr* sont donc définies par la propriété suivante : si la valeur de *expr* est la liste non vide $(\alpha_0\ \alpha_1\ \ldots\ \alpha_n)$, alors

- la valeur de **(car** *expr***)** est la tête de la liste : α_0
- la valeur de **(cdr** *expr***)** est la queue de la liste : $(\alpha_1\ \ldots\ \alpha_n)$

Car et de *cdr* ne sont pas définies pour la liste vide, puisque celle-ci n'a pas une structure de paire pointée.

Exemple : la fonction *element-de?* répond à la question « une valeur donnée figure-t-elle dans une liste donnée ? ». Elle met en scène les trois principales fonctions de consultation des listes, *null?*, *car* et *cdr*, selon un scénario très fréquent :

```
> (define (element-de? valeur liste)
     (cond ((null? liste) #f)
           ((equal? valeur (car liste)) #t)
           (else (element-de? valeur (cdr liste)))))
element-de?
> (element-de? 6 '(2 4 6 8 10 12))
#t
> (element-de? 7 '(2 4 6 8 10 12))
#f
>
```

5.1.2. Construction

DEFINITION : fonction *cons*. Syntaxe :

(cons *expr*$_1$ *expr*$_2$**)**

Évaluation : si la valeur de *expr*$_1$ est α et la valeur de *expr*$_2$ est β, alors la valeur de **(cons** *expr*$_1$ *expr*$_2$**)** est la paire pointée **(**α **.** β**)**

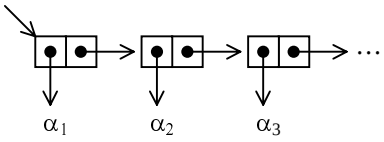
Exemples :

```
> (cons 'Pierre 'Paul)
(Pierre . Paul)
> (cons 1 '(2 3 4 5))
(1 2 3 4 5)
> (cons 1 '())
(1)
>
```

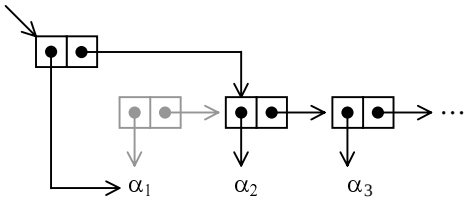
Parmi les fonctions primitives, *cons* est la seule qui « consomme » de la mémoire. En réalité, l'évaluation de (cons *expr*$_1$ *expr*$_2$) se déroule de la manière suivante :

- obtention d'une *cellule-cons* (de la place pour deux pointeurs) entièrement nouvelle,
- affectation aux composantes de cette cellule des valeurs de *expr*$_1$ et *expr*$_2$,
- fourniture, à titre de résultat, de l'adresse de cette nouvelle cellule.

Voici les diagrammes représentatifs des listes Λ et **(cons (car** Λ**) (cdr** Λ**))** :



la liste Λ



la liste (cons (car Λ) (cdr Λ))

On constate que ces deux listes :

- ont la même expression écrite : (α_0 α_1 α_2 …),
- en tant que structures pointées, n'ont pas la même adresse,
- ne sont pas disjointes : elles partagent des sous-structures.

Ces propriétés sont très fréquentes; nous en rencontrerons de nombreux autres exemples.

Exemple. Proposons-nous d'écrire la fonction *append*[1] qui prend deux listes et en construit une troisième, constituée par la mise bout à bout des deux listes données :

```
> (define (append a b)
    (if  (null? a)
         b
         (cons (car a) (append (cdr a) b))))
append
> (append '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
>
```

[1] La fonction *append* est une fonction prédéfinie.

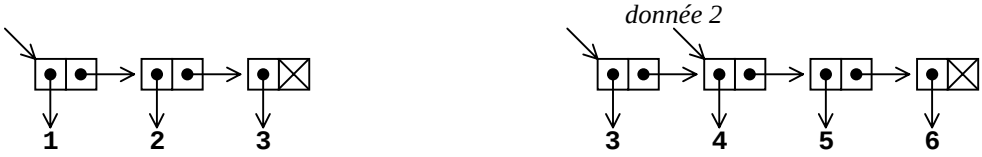
Si nous déroulons « à la main » l'appel de *append*, nous obtiendrons les états suivants :

```
1   (append (1 2 3) (4 5 6))
2   (cons 1 (append (2 3) (4 5 6)))
3   (cons 1 (cons 2 (append (3) (4 5 6))))
4   (cons 1 (cons 2 (cons 3 (append () (4 5 6)))))
5   (cons 1 (cons 2 (cons 3 (4 5 6))))
6   (cons 1 (cons 2 (3 4 5 6)))
7   (cons 1 (2 3 4 5 6))
8   (1 2 3 4 5 6)
```

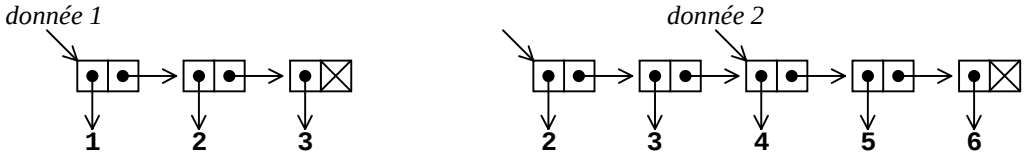
Mettons notre évaluateur sous les rayons X pendant qu'il travaille. Nous assisterons à la reproduction des cellules :

États *1* à *5* :

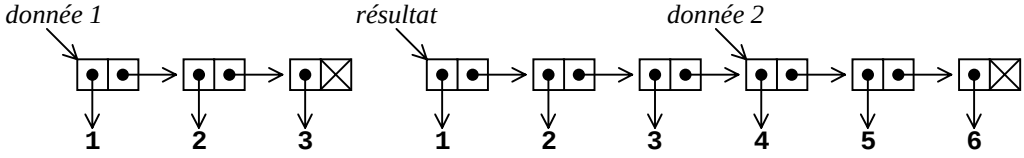
État 6 :



État 7 :



État *8* :



Nous constatons que :

- aucune des deux listes originales n'a été modifiée,
- la première liste a été réellement dupliquée,
- la structure résultante n'est pas tout à fait disjointe d'avec les données.

5.1.3. Autres fonctions

Compositions de *car* et *cdr* :

DEFINITION : fonctions *caar* … *cddddr*. Syntaxe : soient ε_1, $\varepsilon_2 \ldots \varepsilon_k$ deux, trois ou quatre lettres parmi { *a* , *d* }. L'expression

(c$\varepsilon_1\varepsilon_2 \ldots \varepsilon_k$**r** *expr***)**

équivaut à :

(cε_1**r (c**ε_2**r** … **(c**ε_k**r** *expr***)** … **))**

Exemple :

```
> (caar '((Pierre Paul) . (Marie Anne)))
Pierre
> (cdar '((Pierre Paul) . (Marie Anne)))
(Paul)
> (cadddr '(1 2 3 4 5 6))
4
>
```

DEFINITION : fonction *list*. Syntaxe :

(list $expr_1$ $expr_2$ … $expr_n$**)**

Évaluation : rend la liste dont les éléments sont les valeurs de $expr_1$ $expr_2$ … $expr_n$.

Exemple :

```
> (list (+ 2 3) 'Pierre (symbol? 'Paul))
(5 Pierre #t)
>
```

5.2. Comparaison et recherche de structures

5.2.1. Équivalence de structures

Il s'agit de répondre à la question « deux S-expressions données sont-elles égales ? », le problème étant surtout de savoir ce qu'on entend par « égales ».

Tout d'abord, il faut remarquer que les nombres, les caractères et les chaînes de caractères ont leurs propres prédicats d'équivalence, qui tiennent compte des particularités de la représentation interne de ces types, ils sont donc rigoureux et efficaces :

`=`	pour les nombres
`char=?`	pour les caractères
`string=?`	pour les chaînes de caractères

Pour une S-expression quelconque, on a les trois prédicats *eq?*, *eqv?* et *equal?* :

DEFINITION : prédicat *eq?*. Syntaxe :

(eq? $expr_1$ $expr_2$**)**

Évaluation : ce prédicat est vrai si et seulement si les représentations internes des valeurs de $expr_1$ et $expr_2$ sont indiscernables à tout point de vue.

On veut dire par là que :

1. Deux identificateurs ayant la même forme écrite, aux équivalences majuscule/minuscule près, seront reconnus comme égaux par *eq?*. Cela est garanti par la propriété d'unicité de la représentation des identificateurs (cf. page 6).

2. Si les valeurs de $expr_1$ et $expr_2$ renvoient à une même et unique structure, alors elles seront trouvées égales par *eq?*

3. Deux structures qui ont la même expression écrite, mais qui ont été construites indépendamment l'une de l'autre seront trouvées différentes par *eq?*, car elles auront des adresses différentes. Exemple :

```
> (define a '(1 2 3))
a
> (define b a)
b
> (define c '(1 2 3))
c
> (eq? a b)
#t
> (eq? a c)
#f
>
```

4. Le résultat de la comparaison par *eq?* de deux nombres[1], deux caractères ou deux chaînes des caractères est indéfini, car dépendant des propriétés spécifiques à ces objets et de leur codage dans la machine. L'utilisation de *eq?* pour de tels objets est déconseillée.

DEFINITION : prédicat *eqv?*. Syntaxe :

(eqv? $expr_1$ $expr_2$**)**

Évaluation : ce prédicat rend vrai si et seulement si les valeurs de $expr_1$ et $expr_2$ sont :

- équivalents au sens de *eq?*, ou bien
- des nombres équivalents au sens de =, ou bien
- des caractères équivalents au sens de *char=?*, ou bien
- des chaînes équivalentes au sens de *string=?*

Autrement dit, *eqv?* ne parcourt pas plus les structures que *eq?* mais, lorsque ses arguments sont des atomes, il fait la bonne comparaison.

DEFINITION : prédicat *equal?*. Syntaxe :

(equal? $expr_1$ $expr_2$**)**

Évaluation : ce prédicat est défini de la manière suivante :

- si les valeurs de $expr_1$ et $expr_2$ sont des atomes, alors **(equal?** $expr_1$ $expr_2$**)** est vrai si et seulement si ces valeurs sont équivalentes au sens de *eqv?*,
- si les valeurs de $expr_1$ et $expr_2$ sont des paires, alors **(equal?** $expr_1$ $expr_2$**)** est vrai si et seulement si **(car** $expr_1$**)** et **(car** $expr_2$**)** d'une part, et **(cdr** $expr_1$**)** et **(cdr** $expr_2$**)** d'autre part, sont équivalentes au sens de *equal?*,
- dans les autres cas, ce prédicat est faux.

Dit rapidement, **(equal?** $expr_1$ $expr_2$**)** est vrai si et seulement si les valeurs de $expr_1$ et $expr_2$ ont la même expression écrite.

```
> (equal? '(1 2 3) '(1 2 3))
#t
>
```

En résumé : *eq?* est très rapide mais trouve différentes des expressions qu'on souhaiterait qualifier d'égales ; *equal?* n'a pas cet inconvénient, mais est plus lent, car il parcourt les structures.

1 Sauf dans le cas de nombres entiers « raisonnablement » petits.

5.2.2. Recherches dans les listes et dans les listes associatives

DEFINITION : fonctions *memq*, *memv*, *member*. Syntaxe :

(memq *expr liste***)**
(memv *expr liste***)**
(member *expr liste***)**

Évaluation. Ces fonctions rendent la plus grande sous liste de *liste* dont le premier élément est équivalent à la valeur de *expr*, ou **#f** si une telle sous liste n'existe pas.

Avec :			
	memq	équivalence au sens de	**eq?**
	memv	" "	**eqv?**
	member	" "	**equal?**

Exemple :

```
> (memq 'b '(a b a c))
(b a c)
> (memq 'c '(a b a c))
(c)
> (memq 'd '(a b a c))
#f
>
```

DEFINITION : fonctions *assq*, *assv*, *assoc*. Syntaxe :

(assq *expr liste-associative***)**
(assv *expr liste-associative***)**
(assoc *expr liste-associative***)**

liste-associative doit être une liste de paires, c'est à dire de la forme :

$$((\alpha_1 \ . \ \beta_1) \ (\alpha_2 \ . \ \beta_2) \ \ldots \ (\alpha_n \ . \ \beta_n))$$

Cette fonction renvoie la paire $(\alpha_i \ . \ \beta_i)$, où i est le plus petit indice tel que α_i soit équivalent à la valeur de *expr*, ou **#f** si une telle paire n'existe pas.

Avec :			
	assq	équivalence au sens de	**eq?**
	assv	" "	**eqv?**
	assoc	" "	**equal?**

Exemple :

```
> (define répertoire '( (Pierre . 10-101010)
                        (Paul   . 20-202020)
                        (Marie  . #f)
                        (Jeanne . 30-303030)  )
répertoire
> (assq 'Paul répertoire)
(Paul . 20-202020)
> (assq 'Marie répertoire)
(Marie  . #f)
> (assq 'Jacques répertoire)
#f
>
```

5.3. Application des fonctions aux listes

DEFINITION : fonction *map*. Syntaxe :

(map $expr_0$ $expr_1$ … $expr_n$**)**

La valeur de $expr_0$ doit être une fonction φ à n arguments, celles de $expr_1$ … $expr_n$ des listes $(\nu_{1,1}\ \nu_{1,2} \ldots \nu_{1,k})$, $(\nu_{2,1}\ \nu_{2,2} \ldots \nu_{2,k})$, … $(\nu_{n,1}\ \nu_{n,2} \ldots \nu_{n,k})$ ayant le même nombre k d'éléments.

Évaluation : cette fonction renvoie la liste de k éléments obtenue en appliquant la fonction φ aux n-uplets obtenus en prenant les éléments correspondants des listes données, soit :

$$(\ \varphi(\nu_{1,1}\ \nu_{2,1} \ldots \nu_{n,1})\quad \varphi(\nu_{1,2}\ \nu_{2,2} \ldots \nu_{n,2})\ \ldots\ \varphi(\nu_{1,k}\ \nu_{2,k} \ldots \nu_{n,k})\ )$$

L'ordre chronologique dans lequel la fonction est appliquée aux éléments des listes, c'est-à-dire « l'heure » à laquelle chaque $\varphi(\nu_{1,i}\ \nu_{2,i} \ldots \nu_{n,i})$ est évalué, n'est pas spécifié.

Exemple :

```
> (define (carre x) (* x x))
carre
> (map carre '(1 2 3 4 5 6 7 8 9 10))
(1 4 9 16 25 36 49 64 81 100)
> (map cdr '((1 2 3) (4 5 6) (7 8 9)))
((2 3) (5 6) (8 9))
> (map + '(4 3 2 1) '(100 200 300 400))
(104 203 302 401)
>
```

DEFINITION : fonction *apply*. Syntaxe :

(apply $expr_1$ $expr_2$**)**

La valeur de $expr_1$ doit être une fonction φ à k arguments, celle de $expr_2$ une liste ayant k éléments $(\nu_1 \ldots \nu_k)$.

Évaluation : cette fonction renvoie la valeur fournie par la fonction φ quand on l'appelle avec les arguments $\nu_1 \ldots \nu_k$.

```
> (define opérateur +)
opérateur
> (define opérandes '(2 3))
opérandes
> (apply opérateur opérandes)
5
>
```

DEFINITION : fonction *eval*. Syntaxe :

(eval *expr***)**

Évaluation : cette fonction évalue[1] la valeur de *expr*.

Cela fait *deux* évaluations de *expr*, ce en quoi réside la principale utilité de *eval* :

```
> (define formule '(+ 2 3))
formule
> formule
(+ 2 3)                          ; une évaluation de formule
> (eval formule)
5                                ; deux évaluations de formule
>
```

Autre exemple : proposons-nous de vérifier si les « symboles vus comme des données » et les « symboles vus comme des noms de variables » sont bien les mêmes :

---

1 En toute rigueur il faut dire : « cette fonction évalue, *dans l'environnement global*, la valeur de *expr* ».

```
> (define b 123)          ; on définit la variable b, valant 123
b
> (define a 'b)           ; on définit la variable a, valant le symbole b
a
> a                       ; quelle est la valeur de a ?
b
> (eval a)                ; quelle est la valeur de la valeur de a ?
123
>
```

L'expérience est concluante : le symbole *b* et la variable *b* sont tous les deux construits au-dessus de la représentation unique de l'identificateur *b*.

5.4. Bang!, la modification des structures

5.4.1. Set-car! Set-cdr!

Une paire pointée est un enregistrement composé de deux champs, appelés familièrement le *car* et le *cdr*, qui sont généralement des pointeurs. Les fonctions *set-car!* et *set-cdr!* permettent de modifier la valeur de l'un ou l'autre de ces champs.

DEFINITION : fonctions *set-car!* et *set-cdr!*. Syntaxe :

(set-car! $expr_1$ $expr_2$**)**
(set-cdr! $expr_1$ $expr_2$**)**

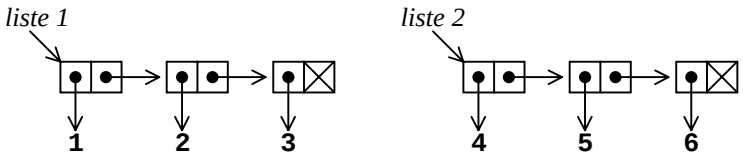
La valeur de $expr_1$ doit être une paire pointée Π.

Évaluation : *set-car!* [resp. *set-cdr!*] remplace la valeur de la première [resp. la deuxième] composante de Π par la valeur de $expr_2$.

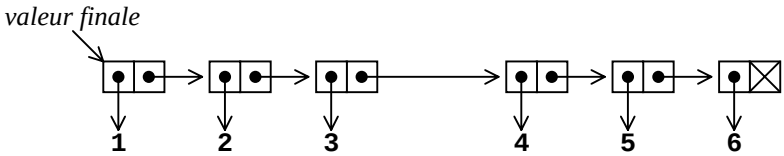
Il est clair que *set-car!* et *set-cdr!* ne sont utilisées que pour leur effet de bord ; la valeur rendue par ces fonctions est indéterminée.

Exemple : proposons-nous d'écrire une nouvelle version de la fonction *append!* qui concatène deux listes sans allouer de maillon nouveau, selon le schéma suivant :

Avant la concaténation :



Après la concaténation :



Voici le programme :

```
> (define (append! liste1 liste2)
    (if (null? liste1)
        liste2
        (begin
            (if (null? (cdr liste1))
                (set-cdr! liste1 liste2)
                (append! (cdr liste1) liste2))
            liste1)))
append!
> (append! '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
>
```

Les fonctions *set-car!* et *set-cdr!* modifient une structure existante ; en Lisp, cela est peu fréquent et toujours risqué (voyez la section suivante). Pour rappeler ce risque, les noms de *set-car!* et *set-cdr!* et ceux des fonctions qui les utilisent directement ou indirectement, se terminent par le caractère '!' qui, en lispien, se prononce « bang ! ».

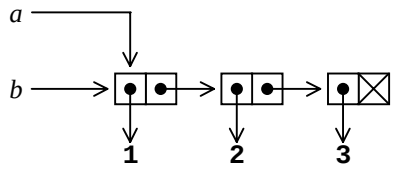
5.4.2. Ces opérations sont dangereuses

Cela provient du fait que Lisp suppose que, une fois construite, une structure n'est jamais modifiée. Or *set-car!* et *set-cdr!* violent cette convention ; d'où les incongruités auxquelles il faudra s'attendre si on les utilise autrement qu'avec une extrême prudence. Illustrons cela :

1. Parce que « les structures ne sont jamais modifiées » la duplication formelle des structures n'entraîne pas forcément leur duplication effective :

```
> (define a '(1 2 3))
a
> (define b a)
b
> a
(1 2 3)
> b
(1 2 3)
>
```

Ici, *a* et *b* contiennent formellement deux exemplaires de la liste (1 2 3) ; mais en réalité il n'y a jamais eu qu'un seul exemplaire de la liste, pointé simultanément par *a* et *b* :



Ce partage des structures ne nuit pas à la correction des programmes… à la condition qu'on n'utilise pas d'opération du type de *set-car!* ou *set-cdr!* :

```
> (set-car! a 0)
(0 2 3)                          ; la valeur de a
> a
(0 2 3)
>
```

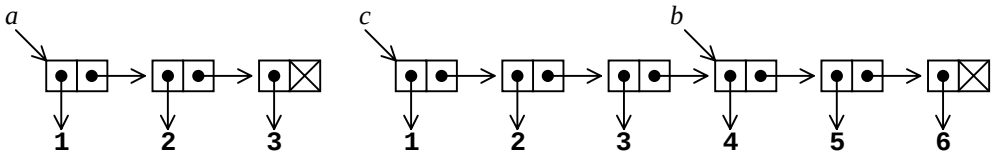
jusqu'ici tout parait bien. Mais :

```
> b
(0 2 3)
>
```

On avait oublié que *a* et *b* sont la même structure ! A retenir : l'emploi de *set-car!* ou de *set-cdr!* peut modifier des objets auxquels on ne pense pas !

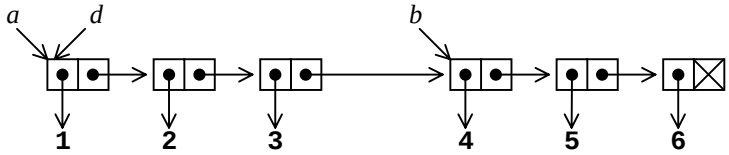
2. Afin que « les structures ne soient pas modifiées », on est amené à effectuer parfois de réelles duplications, onéreuses et apparemment inutiles. Exemple déjà vu :

```
> (define a '(1 2 3))
a
> (define b '(4 5 6))
b
> (define c (append a b))
c
>
```



La première liste a été purement et simplement dupliquée. Voyons ce qu'aurait donné une solution sans duplication, avec les fonctions :

```
> (define d (append! a b))
d
> d
(1 2 3 4 5 6)
>
```



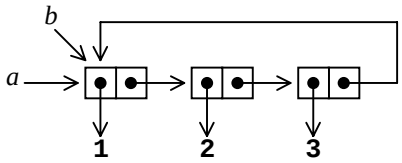
Cela semble marcher aussi bien, et avec une gestion de l'espace plus économe. Mais les inconvénients l'emportent sur les avantages car, d'en faire la concaténation avec *b*, cela a modifié *a*, ce qui n'est pas tolérable : une donnée ne devrait pas être modifiée par le simple fait qu'elle est intervenue dans un calcul :

```
> a
(1 2 3 4 5 6)
>
```

3. Les opérations destructives peuvent provoquer la création de structures circulaires, que Lisp ne sait pas manipuler correctement. Exemple :

```
> (define a '(1 2 3))
a
> (define b (append! a a))
b
> b
(1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 ...
```

Ici, l'utilisation abusive de *append!* (donc de *set-cdr!*) a entraîné la création d'une structure bouclée :



ce qui n'est pas prévu par la fonction d'affichage de l'évaluateur.

☺☹☺

6. La vérité sur les variables et les fonctions

Cette section reprend et complète les notions fondamentales de variable et de fonction, dont nous n'avons eu jusqu'ici que des versions naïves.

6.1. Environnements. La (vraie) notion de variable

6.1.1. L'évaluation revue et corrigée

L'*environnement* d'un point d'un programme est l'ensemble des variables connues en ce point, avec leurs valeurs. Nous pouvons le décrire en utilisant les structures de Lisp :

- le fait qu'un identificateur *ident* soit le nom d'une variable ayant une certaine *valeur* est représenté par une *liaison*, qui est une paire pointée $(ident\ .\ valeur)$;
- un *bloc de liaisons* est une liste de liaisons $((ident_1\ .\ valeur_1)\ \ldots\ (ident_n\ .\ valeur_n))$, où les $ident_i$ sont deux à deux distincts ;
- un *environnement* est une liste de blocs de liaisons.

La notion d'évaluation que nous avons donné au § 3 était naïve. En toute rigueur, l'évaluation d'une expression n'est définie que relativement à un environnement, celui du point où l'expression est écrite. Par conséquent, pour avoir la version complète des règles de l'évaluation, il faut remplacer partout « … l'évaluation d'une expression *expr* … » par « … l'évaluation d'une expression *expr* dans un environnement *E* … ».

Avec cette modification, les règles de l'évaluation déjà données restent valables, sauf celle qui concerne l'évaluation d'un identificateur (cf. page 9), qu'il faut réécrire :

DEFINITION : évaluation d'un atome. L'évaluation d'un atome α dans un environnement *E* est définie de la manière suivante :

- si α est une constante, son évaluation fournit la valeur dont α est l'expression écrite,
- sinon (α est donc un identificateur), soit $(\alpha\ .\ \nu)$ la liaison, concernant α, appartenant au premier[1] bloc de liaisons de *E* qui contient une telle liaison.

  Dans ces conditions, le résultat de l'évaluation est ν.

Une erreur se produit si aucun des blocs de *E* ne contient une liaison concernant α.

Un certain nombre de formes spéciales créent ou modifient les environnements. On les appelle les *formes liantes* :

- *let*, qui crée un bloc de liaison et l'ajoute devant l'environnement courant,
- *define*, utilisée pour modifier l'environnement global,
- *lambda*, pour définir des fonctions, car l'appel d'une fonction est aussi un processus liant.

[1] *E* est une liste de blocs de liaison. Par « premier » nous entendons « le premier trouvé en cherchant à partir du début de *E* » ; compte tenu de la manière dont les blocs de liaison s'empilent dans les environnement (voyez la définition de la forme *let*) cela voudra dire en pratique « le bloc le plus récent » ou encore « le bloc créé par la forme liante la plus imbriquée (parmi celles qui ont contribué à créer l'environnement *E*) ».

6.1.2. La forme let et les variables locales

La notion d'environnement permet de donner un sens précis à l'expression « définir un paquet de variables locales » :

DEFINITION : forme spéciale *let*. Syntaxe :

(let ((*ident*$_1$ *expr*$_1$**)** … **(***ident*$_n$ *expr*$_n$**))**
expr'$_1$ … *expr'*$_k$**)**

où *ident*$_1$ … *ident*$_n$ sont des identificateurs deux à deux distincts.

Évaluation dans un environnement *E* :

- les expressions *expr*$_1$, … *expr*$_n$ sont évaluées dans l'environnement *E* ; soient ν_1, … ν_n les valeurs obtenues,
- les expressions *expr'*$_1$ … *expr'*$_k$ sont alors évaluées, dans cet ordre, dans l'environnement

  E' = **(((***ident*$_1$ **.** ν_1**)** … **(***ident*$_n$ **.** ν_n**)) .** *E* **)**

  la valeur de *expr'*$_k$ est la valeur rendue par la forme *let*.

```
> (define x 1000)
x
> (let ((x 2) (y 3))
    (+ x y))
5
>
```

Explication opérationnelle :

- quand l'exécution « rentre » dans la forme *let*, Lisp crée le paquet de liaisons indiquées dans le *let* et le place au sommet de l'environnement courant, au-dessus de toutes les autres liaisons ;
- durant l'évaluation des expressions *expr'*$_1$ … *expr'*$_k$ (le « corps » du *let*), ces liaisons masquent d'éventuelles autres liaisons concernant *ident*$_1$ … *ident*$_n$ qui pourraient être connues en dehors du *let* ;
- enfin, quand l'exécution « sort » du *let*, ces liaisons sont détruites et l'environnement redevient ce qu'il était avant d'entrer dans le *let*.

DEFINITION : forme spéciale *let**. Syntaxe :

(let* ((*ident*$_1$ *expr*$_1$**)** … **(***ident*$_n$ *expr*$_n$**))**
expr'$_1$ … *expr'*$_k$**)**

L'évaluation est définie comme celle de la forme *let*, sauf que

- les expressions *expr*$_1$ … *expr*$_n$ sont évaluées dans l'ordre où elles apparaissent,
- chacune est évaluée dans un environnement déjà étendu par les liaisons précédemment calculées.

Ainsi,

(let* ((*i*$_1$ *e*$_1$**) (***i*$_2$ *e*$_2$**)** … **(***i*$_n$ *e*$_n$**))** *e'*$_1$ … *e'*$_k$**)**

est un raccourci pour :

(let ((*i*$_1$ *e*$_1$**))** **(let ((***i*$_2$ *e*$_2$**))** … **(let ((***i*$_n$ *e*$_n$**))** *e'*$_1$ … *e'*$_k$**)** … **))**

Exemple :

```
> x
0
> (let  ((x 1) (y (+ x 1))) y)
1
> (let* ((x 1) (y (+ x 1))) y)
2
>
```

DEFINITION : forme spéciale *letrec*. Syntaxe

(letrec ((*ident*$_1$ *expr*$_1$**)** … **(***ident*$_n$ *expr*$_n$**))**
expr'$_1$ … *expr'*$_k$**)**

L'évaluation est définie comme celle de la forme *let*, sauf que les expressions *expr*$_1$ … *expr*$_n$ sont évaluées dans un environnement dans lequel les liaisons **(***ident*$_1$ **.** *expr*$_1$**)** … **(***ident*$_n$ **.** *expr*$_n$**)** sont déjà connues.

Contrainte : chaque expression *expr*$_i$ doit pouvoir être évaluée sans utiliser les *valeurs* des autres expressions *expr*$_j$ ($j \neq i$).

En pratique, cela veut dire le plus souvent que les expressions *expr*$_i$ sont des définitions de fonctions dont les corps mentionnent les identificateurs *ident*$_j$ ($j \neq i$).

Exemple (prématuré et passablement idiot) :

```
> (define (nombre-pair x)
      (letrec (    (nbpr (lambda (n)
                           (if (zero? n) #t
                               (not (nbim (– n 1))))))
                   (nbim (lambda (n)
                           (if (zero? n) #f
                               (not (nbpr (– n 1)))))))
      (nbpr x)))
pair
> (nombre-pair 8)
#t
>
```

6.1.3. L'affectation

DEFINITION : forme spéciale *set!* Syntaxe :

(set! *ident expr***)**

où *ident* est un identificateur.

Évaluation dans un environnement *E* : soit γ la liaison **(***ident* **.** *valeur***)** appartenant au premier[1] bloc de *E* qui contient une liaison concernant *ident*.

L'évaluation de **(set!** *ident expr***)** équivaut alors à celle de **(set-cdr!** γ *expr***)**.

Une erreur se produit si aucun bloc de *E* ne contient de liaison concernant *ident*.

On l'a reconnue : c'est l'affectation pure et simple des autres langages[2] ! La variable *ident* doit exister dans l'environnement courant, l'évaluation de **(set!** *ident expr***)** lui affecte la valeur de *expr* en écrasant sa valeur précédente. Exemple :

```
> (define n 0)
n
> n
0
```

---

1 Encore une fois, il s'agit du premier bloc trouvé en cherchant à partir du début de *E*, c'est-à-dire le bloc le plus récent, celui qui a été créé par la forme liante la plus imbriquée.

2 Ce qui est surprenant, c'est qu'on ait pu se passer d'en parler jusqu'à présent !

```
> (set! n (+ n 1))
#[undefined]
> n
1
>
```

Remarque. Le nom de la forme *set!* comporte un '!' parce qu'un appel de cette fonction modifie une structure de l'interprète de Lisp (la liaison concernant la variable en question). Comme nous l'avons indiqué dans sa définition, *set!* utilise de manière interne *set-cdr!*

On notera cependant que *set!* ne permet pas de modifier une structure du programmeur[1]. Contrairement à *set-car!* et *set-cdr!*, elle ne peut donc pas être tenue pour « dangereuse ». On peut se passer longtemps de *set!*, d'autant plus que la bonne programmation fonctionnelle consiste à composer des appels de fonctions, non à enchaîner des affectations de variables. Mais l'affectation est la seule manière de résoudre certains problèmes, elle est bien utile et ne présente pas plus de « dangers » que l'affectation de Pascal ou C.

6.1.4. L'environnement global

Les évaluations écrites au niveau supérieur (la boucle principale) se font dans l'*environnement global*, qui ne comporte qu'un seul bloc de liaisons, composé de l'ensemble des variables globales. La forme *define*, évaluée au niveau supérieur, ajoute une liaison à ce bloc, sans en créer de nouveau.

L'environnement global a une propriété que n'ont pas les autres environnements : le fait qu'une liaison concernant une variable existe un non dépend du moment auquel la question est posée. Le nombre de liaisons qui composent l'environnement global change au cours du temps :

```
> x
ERREUR : la variable x n'est pas définie
…
> (define x 0)              ; plus tard…
x
…
> x                         ; encore plus tard…
120
>
```

6.2. Lambda. La (vraie) notion de fonction

6.2.1. Définition et appel des fonctions

On est enfin en mesure de définir précisément deux notions parmi les plus importantes d'un langage fonctionnel : la création des fonctions et l'appel des fonctions.

DEFINITION : forme spéciale *lambda*. Syntaxe :

(lambda (*arg-for*$_1$ … *arg-for*$_n$**)** *expr'*$_1$ … *expr'*$_k$**)**

arg-for$_1$ … *arg-for*$_n$ doivent être des identificateurs ; on les appelle les *arguments formels* de la fonction. Les expressions *expr'*$_1$ … *expr'*$_k$ constituent le *corps* de la fonction.

Évaluation. Le résultat de l'évaluation de cette expression, dans un environnement E_0, est un objet Φ, appelé *objet fonctionnel*, *fonction* ou encore *fermeture*, qui est un triplet formé de trois éléments :

- la liste des arguments formels (*arg-for*$_1$ … *arg-for*$_n$),
- une certaine représentation, familièrement appelée le *code de la fonction*, des expressions *expr'*$_1$ … *expr'*$_k$,
- l'environnement E_0 (c'est-à-dire l'environnement de l'endroit du programme où la forme lambda est évaluée).

1 Se convaincre de cela est un bon exercice.

La définition suivante complète la définition de l'évaluation d'une liste, donnée au § 3.2, et définit précisément ce qu'on entend par appel (ou application) d'une fonction.

DEFINITION : évaluation d'une liste et appel, ou application, d'une fonction. Syntaxe :

$$(expr_0 \;\; expr_1 \;\; \ldots \;\; expr_n)$$

Évaluation. Si $expr_0$ n'est pas le mot-clé d'une forme spéciale, l'évaluation dans un environnement E de la liste ci-dessus commence par l'évaluation dans E de chacun de ses éléments. Notons ν_i le résultat de l'évaluation dans E de $expr_i$.

Une erreur survient si ν_0 n'est pas un objet fonctionnel, ou si ν_0 est un objet fonctionnel ayant un nombre d'arguments formels différent de n.

Supposons que ν_0 soit l'objet fonctionnel Φ construit par un appel de la forme *lambda* comme indiqué dans la définition précédente. L'évaluation de l'appel de fonction consiste alors en l'évaluation des expressions $expr'_1 \ldots expr'_k$, successivement et dans cet ordre, dans l'environnement

$$E_1 = \mathbf{(}\,\mathbf{(}\,\mathbf{(}arg\text{-}for_1 \;\mathbf{.}\; \nu_1\mathbf{)} \;\ldots\; \mathbf{(}arg\text{-}for_k \;\mathbf{.}\; \nu_k\mathbf{)}\,\mathbf{)} \;\mathbf{.}\; E_0\,\mathbf{)}$$

La valeur rendue par l'application est la valeur de $expr_k$.

Plus simplement : pour évaluer un appel de fonction, Scheme :

- évalue les paramètres effectifs dans l'environnement dans lequel la fonction est appelée,
- construit un environnement, en ajoutant à l'environnement dans lequel la fonction a été *définie* les liaisons des arguments formels aux valeurs des arguments effectifs,
- évalue, dans cet environnement nouveau, les expressions qui forment le corps de la fonction.

Exemples :

```
> (lambda (x y) (+ (* x x) (* y y)))
#[procedure #x9D7F0]
> ((lambda (x y) (+ (* x x) (* y y))) 3 4)
25
> (define somcar (lambda (x y) (+ (* x x) (* y y))))
somcar
> (somcar 3 4)
25
>
```

Remarques. 1. Il est dit, dans la définition de la forme *lambda*, qu'un objet fonctionnel comporte « une certaine représentation » des expressions $expr'_1 \ldots expr'_k$ qui constituent le corps de la fonction. Selon l'interprète Lisp que vous utilisez, cela peut aller de la simple recopie des expressions $expr'_1 \ldots expr'_k$, jusqu'à leur totale traduction en langage machine (compilation). Ce point n'affecte pas la sémantique des programmes Lisp mais uniquement leur efficacité.

2. En principe, les objets fonctionnels n'ont pas d'expression écrite. Cela découle de la note précédente, car trouver l'expression écrite d'une fonction reviendrait sur certains systèmes à « décompiler » cette dernière. Tout ce qu'on obtient, à titre d'expression écrite d'une fonction, c'est une indication conventionnelle réputée suggestive, comme **`#[procedure #x9D7F0]`** (suggestif, hein ?).

Notez cependant que certains systèmes mémorisent le texte source des fonctions à côté de leur code. On peut donc retrouver ce dernier à l'aide d'une fonction d'écriture « soignée » (du genre de *pretty-print*, *pp*, etc.) :

```
> (define (somcar x y)
    (+ (* x x) (* y y)))
somcar
> somcar
#[procedure somcar]
> (pp somcar)
(lambda (x y) (+ (* x x) (* y y)))
>
```

3. Mis à part pour ce qui concerne le point précédent, les objets fonctionnels sont des objets ordinaires, avec le même statut que les autres objets manipulés par Lisp. En particulier, toutes les opérations qui peuvent être appliquées à un objet atomique peuvent l'être aussi à une fonction : affectation à une variable, rangement dans une structure, passage à une fonction à titre d'argument, etc.

6.2.2. La vérité sur define

Le moment est venu d'apprendre que, sauf dans un cas, la forme *define* n'est que du sucre syntaxique, c'est-à-dire une manière commode d'écrire des expressions possédant par ailleurs une formulation équivalente, moins lisible mais plus proche des mécanismes primitifs que sont *let* et *lambda*.

1. *Define-pour-définir-une-fonction* se ramène dans tous les cas à *define-pour-définir-une-variable*, selon le schéma suivant :

S'il s'agit de définir une fonction qui ne s'appelle pas elle-même, l'expression

(define (*id arg-for*$_1$ … *arg-for*$_n$**)**
expr$_1$ … *expr*$_k$**)**

peut être vue comme un raccourci de :

(define *id* **(lambda (***arg-for*$_1$ … *arg-for*$_n$**)**
expr$_1$ … *expr*$_k$**))**

En réalité, Lisp fait une transformation plus compliquée qui couvre le cas des fonctions directement récursives (les fonctions qui s'appellent elles-mêmes) : l'expression

(define (*id arg-for*$_1$ … *arg-for*$_n$**)**
expr$_1$ … *expr*$_k$**)**

est traduite par :

(define *id*
(letrec ((*id* **(lambda (***arg-for*$_1$ … *arg-for*$_n$**)**
expr$_1$ … *expr*$_k$**)))**
*id***))**

2. *Define-pour-definir-une-variable-locale* se ramène à *letrec*, selon le schéma suivant. L'expression :

(lambda (*argf*$_1$ … *argf*$_n$**)**
(define *id*$_1$ *expr*$_1$**)**
…
(define *id*$_m$ *expr*$_m$**)**
expr$_{m+1}$
…
expr$_k$**)**

équivaut à :

(lambda (*argf*$_1$ … *argf*$_n$**)**
(letrec ((*id*$_1$ *expr*$_1$**)**
…
(*id*$_m$ *expr*$_m$**))**
expr$_{m+1}$
…
expr$_k$**))**

3. *Define-pour-définir-une-variable-globale*, contrairement aux deux cas précédents, est une forme essentielle : elle est le seul moyen d'étendre « sur place » (c'est-à-dire, sans création de nouveaux blocs de liaisons) l'environnement global. Voir § 6.1.4.

☺☺☹

7. Applications

Cette section donne, sous forme d'exemples, quelques pistes pour des questions avancées qui seront développées durant les séances de TD.

7.1. Lambda et le bon usage des variables libres

Les notions, opposées l'une de l'autre, d'*occurrence libre* et *occurrence liée* d'une variable dans une expression se définissent correctement dans le cadre du λ-calcul, une théorie très formelle qui sort de notre propos ici. Pour en avoir une idée intuitive, considérons l'expression

i `(* a x)`

elle comporte deux occurrences de variables, *a* et *x*, toutes les deux libres. Considérons maintenant les occurrences correspondantes de *a* et *x* dans l'expression

ii `(lambda (x) (* a x))`

l'occurrence de *a* est toujours libre, mais celle de *x* a été liée, par la forme *lambda*, qui fait apparaître cette variable dans sa liste d'arguments formels. L'expression (* *a x*) est maintenant le corps de la fonction $x \rightarrow$ (* *a x*). Notez que, comme on dit dans le cours de mathématiques, la variable *x* est devenue une « variable muette ». La preuve en est que la fonction définie par *ii* est la même que celle définie par :

ii' `(lambda (y) (* a y))`

Puisque l'expression *ii* a encore une variable libre, on peut réitérer l'opération :

iii `(lambda (a) (lambda (x) (* a x)))`

Cette expression représente la fonction $a \rightarrow (x \rightarrow$ (* *a x*)). Elle n'a aucune variable libre. Son évaluation dans un environnement *E* produit une fonction dont le corps ne mentionne aucune variable de *E* ; par conséquent, il n'est pas nécessaire de connaître le texte entourant la définition *iii* pour savoir exactement ce que la fonction ainsi construite calcule. Une expression sans variable libre est entièrement auto-explicative.

Être auto-explicative est une qualité. Cependant, dans certains cas, l'écriture d'expressions avec des variables libres au sein d'environnements savamment choisis permet d'obtenir des programmes élégants et efficaces.

Par exemple, cherchons à écrire la fonction qui rend la liste des éléments du produit cartésien de deux ensembles donnés par les listes de leurs éléments :

```
> (produit '(a b c) '(1 2 3 4))
((a . 1) (a . 2) (a . 3) (a . 4) (b . 1) (b . 2)
     (b . 3) (b . 4) (c . 1) (c . 2) (c . 3) (c . 4))
>
```

Il est certain qu'au cœur de notre programme on trouvera l'opération « construire une paire d'éléments », soit

```
(cons x y)
```

Cette expression a deux variables libres. Faisons-en une fonction :

```
(lambda (y) (cons x y))
```

et appliquons-la à chaque élément de la deuxième liste, notée *b* :

```
(map (lambda (y) (cons x y)) b)
```

La variable *x* est toujours libre. Transformons l'expression précédente en une fonction :

```
(lambda (x)
     (map (lambda (y) (cons x y)) b))
```

et appliquons-la à chaque élément de la première liste, notée *a* :

```
(map (lambda (x)
     (map (lambda (y) (cons x y)) b)) a)
```

Cette expression a deux variables libres, *a* et *b*, qui, dans notre esprit sont les deux listes en question. Nous tenons notre fonction :

```
(lambda (a b)
     (map (lambda (x)
          (map (lambda (y) (cons x y)) b)) a))
```

Baptisons-la *produit*, et testons-la :

```
> (define (produit a b)
     (map (lambda (x)
          (map (lambda (y) (cons x y)) b)) a))
produit
> (produit '(1 2 3) '(a b))
(((1 . a) (1 . b)) ((2 . a) (2 . b)) ((3 . a) (3 . b)))
```

Notre fonction a un petit défaut : elle ne rend pas une liste de couples, mais une liste de listes de couples. Pour en faire une seule liste, il faut les mettre bout à bout :

```
> (define (produit a b)
     (apply append
          (map (lambda (x)
               (map (lambda (y) (cons x y)) b)) a)))
produit
> (produit '(1 2 3) '(a b))
((1 . a) (1 . b) (2 . a) (2 . b) (3 . a) (3 . b))
```

7.2. Les fermetures

L'environnement du point où une fonction est créée fait partie de la fonction (§ 6.2.1). Cela signifie que dans le corps de la fonction peuvent apparaître des variables qui ne sont ni des arguments formels ni des variables locales, mais qui sont connues à l'endroit où la forme lambda est évaluée. Nous avons utilisé cette propriété dans la section précédente.

Lorsque la fonction est appelée dans l'environnement où elle a été créée cette possibilité semble tout à fait naturelle. Mais que se passe-t-il si la fonction est extraite de la région du programme dans laquelle elle a été construite, et est utilisée dans une région ayant un autre environnement ? Nous allons le voir, Scheme tient ses promesses : puisque la fonction doit pouvoir accéder aux variables de l'environnement qui l'a vue naître, si elle quitte sa région d'origine alors elle emporte avec elle son environnement de naissance[1].

C'est ce que nous avons voulu exprimer au § 6.2.1 en indiquant qu'un objet fonctionnel était un triplet (*arguments formels*, *corps*, *environnement de naissance*). Nous allons en tirer quelques conséquences :

Compteurs

Appelons *compteur* une fonction sans argument qui, à chaque appel, fournit un entier : 0 lors du premier appel, 1 à l'appel suivant, ensuite 2, etc. La fonction *nouveau-compteur* renvoie un compteur nouvellement construit :

```
> (define (nouveau-compteur)
     (let ((valeur 0))
          (lambda ()
               (set! valeur (+ valeur 1))
               valeur)))
nouveau-compteur
```

[1] Elle emporte au moins la *partie utile* de cet environnement, c'est-à-dire l'ensemble des variables qui sont mentionnées dans le corps de la fonction.

```
> (define c1 (nouveau-compteur))
c1
> (c1)
1
> (c1)
2
> (define c2 (nouveau-compteur))
c2
> (c2)
1
> (c1)
3
>
```

Examinons le corps de *nouveau-compteur* : un appel de la fonction *nouveau-compteur* rend un exemplaire de la fonction

```
(lambda ()
     (set! valeur (+ valeur 1))
     valeur)
```

Cette fonction a été créée dans un environnement comportant une variable nommée *valeur*. Il s'agit d'une variable locale du *let* qui forme le corps de *nouveau-compteur*. Dans un cas normal, cette variable aurait été détruite à la fin de l'appel de *nouveau-compteur*. Ici elle ne l'est pas car, étant référencée dans le corps de la fonction, elle est un des éléments de l'environnement que la fonction produite « emporte avec elle ».

On peut également créer des compteurs avec un incrément variable. Il suffit de produire une fonction avec un argument, l'incrément souhaité :

```
> (define (nouveau-compteur)
     (let ((valeur 0))
          (lambda (incr)
               (set! valeur (+ valeur incr))
               valeur)))
nouveau-compteur
> (define c (nouveau-compteur))
c
> (c 10)
10
> (c 5)
15
>
```

On peut de même créer des compteurs dont la valeur initiale n'est pas nulle :

```
> (define (nouveau-compteur valeur)
     (lambda (incr)
          (set! valeur (+ valeur incr))
          valeur))
nouveau-compteur
> (define k (nouveau-compteur 1000))
k
> (k 10)
1010
>
```

Messages

Les compteurs précédents sont très rudimentaires : tout ce qu'ils savent faire, c'est ajouter un nombre à leur valeur courante. Nous allons créer des compteurs avec un comportement plus riche : chacun saura réagir à quatre « messages » différents :

- *incrément k* le compteur incrémente sa valeur courante de *k*, et rend sa nouvelle valeur,
- *décrément k* le compteur décrémente sa valeur courante de *k*, et rend sa nouvelle valeur,
- *valeur* le compteur rend sa valeur, sans en changer,
- *remise-à-zéro* la valeur du compteur redevient zéro.

Programme :

```
> (define (nouveau-compteur)
     (let ((valeur 0))
          (lambda (message)
               (cond
                    ((eq? message 'incrément)
                         (lambda (k)
                              (set! valeur (+ valeur k))
                              valeur))
                    ((eq? message 'décrément)
                         (lambda (k)
                              (set! valeur (- valeur k))
                              valeur))
                    ((eq? message 'valeur) valeur)
                    ((eq? message 'remise-à-zéro)
                         (set! valeur 0)
                         valeur)
                    (else (error 'compteur message))))))
nouveau-compteur
> (define k (nouveau-compteur))
k
> (k 10)
*** ERROR -- nouveau-compteur 10
> ((k 'incrément) 1000)
1000
> ((k 'décrément) 10)
990
> (k 'valeur)
990
> (k 'remise-à-zéro)
0
> (k 'valeur)
0
>
```

Voici la même fonction, écrite avec *define* à la place de *lambda* :

```
> (define (nouveau-compteur)
     (define valeur 0)
     (define (incrementer k)
          (set! valeur (+ valeur k))
          valeur)
     (define (decrementer k)
          (set! valeur (- valeur k))
          valeur)
     (define (dispatcher message)
          (cond
               ((eq? message 'increment) incrementer)
               ((eq? message 'decrement) decrementer)
               ((eq? message 'valeur) valeur)
               ((eq? message 'remise-à-zéro)
                    (set! valeur 0)
                    valeur)
               (else (error 'nouveau-compteur message))))
     dispatcher)
```

7.3. Les continuations des fonctions

Une *continuation* d'un appel de fonction Φ est un traitement que l'on applique au résultat rendu par la fonction. Programmer dans le style « par passage de continuation » consiste à donner à Φ, comme argument supplémentaire, une fonction qui représente la continuation de l'appel de Φ.

Exemple : calcul de la « norme euclidienne » $\sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$ des éléments $x_1, x_2, \ldots x_n$ d'une liste de nombres donnée. Version ordinaire :

```
> (define (somcar l)
     (if (null? l)
          0
          (+ (carre (car l)) (somcar (cdr l)))))
somcar
> (define (norme l)
     (sqrt (somcar l))
norme
>
```

Dans l'expression *(sqrt (somcar l))* il est évident que l'appel de *sqrt* est la continuation de l'appel *(somcar l)*. Programmation avec continuation :

```
> (define (somcar l c)
     (if (null? l)
          (c 0)
          (somcar (cdr l)
               (lambda (u) (c (+ (carre (car l)) u))))))
somcar
> (define (norme l)
     (somcar l sqrt))
norme
>
```

La programmation par passage de la continuation parait obscure et prétentieuse. Pour sa défense, notons que

1. La PPC transforme la récursivité en récursivité terminale[1].

2. La PPC permet d'écrire de manière élégante et claire les fonctions qui rendent plusieurs résultats, ce qui n'est pas aisé en programmation fonctionnelle. Par exemple, écrivons la fonction qui prend une liste non vide de nombres et rend une paire formée de la valeur et l'indice du maximum de la liste. Version classique :

```
> (define (vimax l)
     (if (null? (cdr l))
          (cons (car l) 0)
          (let ((r (vimax (cdr l))))   ; r = (ValMax . IndMax)
               (if (<= (car r) (car l))
                    (cons (car l) 0)
                    (cons (car r) (+ (cdr r) 1))))))
vimax
>
```

Version avec continuation (la continuation est une fonction à deux arguments, observez l'appel initial de *vimax*) :

```
> (define (vimax l c)
     (if (null? (cdr l))
          (c (car l) 0)
          (vimax (cdr l)
               (lambda (v i)
                    (if (<= v (car l))
                         (c (car l) 0)
                         (c v (+ i 1))))))))
vimax
> (vimax '(10 40 20 60 30 50) cons)
(60 . 3)
>
```

3. La PPC permet de court-circuiter la suite d'un calcul dont on connaît le résultat. Exemple, la fonction qui calcule le produit des éléments d'une liste :

---

1 Mais au prix de constructions de fonctions qui finissent par être très lourdes.

```
> (define (produit l)
     (if (null? l)
          1
          (* (car l) (produit (cdr l)))))
produit
>
```

Cette fonction est juste mais, lorsqu'un élément de la liste est nul, elle continue à faire un tas de multiplications inutiles. Version avec continuation, optimisée :

```
> (define (produit l c)
     (cond ((null? l) (c 1))
          ((zero? (car l)) (c 0))
          (else (produit (cdr l)
                         (lambda (u) (c (* (car l) u)))))))
produit
> (produit '(1 2 3 4 5 0 7 8 9 10 11) (lambda (x) x))
0
>
```

☺☺☺

Références

★★★ J. Chazarain
Programmer avec Scheme. De la pratique à la théorie.
International Thomson Publishing France, 1996

★★★ H. Abelson, G.J. Sussman, J. Sussman
Structure et interprétation des programmes informatiques
InterEditions, 1989

★★ D.P. Friedman, M. Felleisen
Le petit Lispien
Masson, 1991

★ W. Clinger, J. Rees
Revised^4 Report on the Algorithmic Language Scheme
Gratuit. On peut le trouver sur le site *www.schemers.com* (entre autres)

Index des définitions