ELEMENTS DE

PROGRAMMATION

Cours et exercices corrigés en Scheme

Avril 2000, octobre 2005

Pascal Gribomont

L'auteur remercie L. Arditi, P. Fontaine, J.-M. Hufflen, L. Moreau et D. Ribbens qui ont relu différentes versions du manuscrit et contribué à améliorer le texte.

Contents

1 Introduction

Ce livre se veut un support à l'apprentissage de la programmation. Il existe plusieurs grands styles de programmation, parfois appelés "paradigmes" de programmation. On distingue notamment les styles impératif, fonctionnel, logique, orienté objet. Ce texte développe les bases du style fonctionnel de programmation et utilise le langage SCHEME; il s'adresse au lecteur n'ayant aucune connaissance de programmation fonctionnelle. Il (ou elle) peut avoir ou ne pas avoir d'expérience antérieure de la programmation impérative (le style le plus répandu), dans un langage tel que Fortran, Pascal ou C.

Comme son nom l'indique, la programmation fonctionnelle a pour concept de base la fonction, notion mathématique fondamentale bien connue du lecteur, mais sur laquelle il n'est pas inutile de revenir. En particulier, on verra jusqu'à quel point on peut assimiler l'activité mathématique consistant à *définir* une fonction et l'activité informatique consistant à *programmer* une fonction. On présentera à ce propos la notion de récursivité, centrale en programmation fonctionnelle.

Dans ce livre, on insiste sur les principes de la programmation fonctionnelle mais ils n'occupent guère d'espace, car ils sont peu nombreux, simples et clairs. Il est nécessaire de les assimiler parfaitement pour comprendre les programmes en profondeur, et surtout pour pouvoir programmer soi-même.[1] Ce livre se veut à orientation pratique; son but est d'aider le lecteur à apprendre à programmer et à concrétiser cet apprentissage par une résolution simple et fiable de problèmes variés, impliquant l'emploi correct d'un langage de programmation, SCHEME (en fait, un sous-ensemble réduit de SCHEME).

Ce texte ne constitue pas un recueil de problèmes et de programmes.[2] Les exercices sont donc relativement peu nombreux, mais ils sont très importants. Il servent d'abord à présenter une technique particulière ou à démontrer concrètement son utilité, mais ils sont aussi et surtout destinés à convaincre le lecteur d'un fait largement méconnu : la programmation est essentiellement une technique, ou un ensemble de techniques. Bien programmer requiert toujours compétence, rigueur et soin; ce n'est qu'occasionnellement que le programmeur doit en outre faire preuve de créativité, d'astuce, voire même de sens esthétique. Dans nos exercices corrigés, la démarche de résolution a donc autant d'importance que le programme produit. Nous nous sommes efforcés d'apporter des solutions complètes et détaillées aux problèmes traités; en particulier, chaque fonction auxiliaire est spécifiée; nous pensons en effet que la documentation doit toujours accompagner le programme. Notons enfin que la programmation s'apprend surtout ... en programmant. Le lecteur ne pourra tirer un bénéfice optimal de cet ouvrage que s'il

[1] C'est la raison pour laquelle certains points sont présentés deux fois. La première présentation donne l'idée, immédiatement mise en œuvre; la seconde situe le concept dans un contexte plus large, permettant de mieux apprécier son importance. Par contre, ce livre n'est pas un manuel de référence du langage SCHEME et de nombreux aspects importants du langage ne sont évoqués que de manière indirecte et occasionnelle. Par exemple, nous ne mentionnons pas explicitement que le nombre 3/4 en SCHEME peut s'écrire `3/4`, `0.75`, `75e-2`, `.75+0i`, etc., ni que la première écriture seule permet un calcul "exact", ni enfin que cette écriture fractionnaire n'est pas admise sur tous les systèmes SCHEME. Ces aspects ont leur importance mais leur traitement n'aurait pas contribué à l'objectif premier de cet ouvrage.

[2] Le lecteur consultera utilement [6], où il trouvera de nombreux problèmes supplémentaires.

écrit ses propres programmes et expérimente largement sur machine.[3]

1.1 De la fonction mathématique à la fonction programmée

Considérons la définition suivante :

> *La factorielle d'un entier naturel n*
> *est le produit des entiers de 1 à n.*

Le mathématicien admettra la définition en observant qu'elle repose sur des notions antérieurement admises (entier naturel, ensemble, produit d'un ensemble de nombres) et que ces notions sont correctement combinées.[4]

L'algorithmicien suivra une démarche analogue, mais ira plus loin. En effet, une définition mathématiquement acceptable d'une fonction f (ici, la fonction qui à tout entier naturel n associe sa factorielle) n'implique pas automatiquement l'existence d'un *processus d'évaluation* qui, au départ de l'expression syntaxique $f(x)$, calcule la *valeur* de cette expression, c'est-à-dire l'image de x par f. Si de tels algorithmes existent, ils peuvent n'être pas d'égal intérêt pour l'algorithmicien, qui s'interrogera notamment sur leur efficacité. Pour le mathématicien, les fonctions

$$g : n \mapsto \sum_{i=1}^{n} i^2$$

et

$$h : n \mapsto \frac{n(n+1)(2n+1)}{6}$$

sont égales, parce que, pour tout entier naturel n, les images de n par g et par h sont égales. L'algorithmicien notera une différence importante entre g et h : la définition de g suggère que le calcul de l'image de 100 par g, c'est-à-dire le processus d'évaluation de $g(100)$, requiert 100 multiplications et 99 additions tandis que la définition de h suggère que le calcul de l'image de 100 par h requiert seulement trois multiplications, deux additions et une division. Dès que l'algorithmicien connaît l'égalité mathématique de g et h, il peut cesser de s'intéresser à g.

Le programmeur ira encore un peu plus loin, puisqu'il doit coder l'algorithme dans un certain langage de programmation. Les primitives (opérations fournies par le système) varient d'un langage à l'autre; en outre, dans un langage fixé, il arrive que plusieurs

[3] Des systèmes SCHEME performants et gratuits existent en abondance; on peut les télécharger via Internet. Un point d'entrée intéressant vers des références multiples est `http://www.cs.indiana.edu/scheme-repository/home.html` .

[4] En particulier, les cas $n = 0$ et $n = 1$ ne sont pas exclus; on convient que "l'ensemble des entiers de 1 à 0" est l'ensemble vide et que "l'ensemble des entiers de 1 à 1" est le singleton $\{1\}$; le produit de chacun de ces ensembles vaut 1. (Toute autre convention conduirait à des difficultés.) Notons en outre que la notion de produit d'un ensemble (fini) de nombres n'est pas primitive, mais dérivée de la notion de produit de deux nombres, opération associative et commutative.

solutions soient envisageables pour un même algorithme; le programmeur choisit en fonction de plusieurs facteurs (efficacité, clarté, et même goûts personnels).

Une définition donnée en langage naturel, même claire et précise, devra être formalisée. Le mathématicien écrira simplement

$$n! =_{def} 1 * 2 * \cdots * n\,,$$

les points de suspension ayant ici une signification parfaitement claire.[5]

L'algorithmicien et le programmeur n'admettront pas les points de suspension si leur signification n'est pas formellement spécifiée.[6] En programmation fonctionnelle, la technique privilégiée pour l'élimination des points de suspension est la *récursivité*. Pour représenter de manière finie le tableau infini

$$\begin{aligned} 0! &= 1\,,\\ 1! &= 1 * 1\,,\\ 2! &= 2 * (1 * 1)\,,\\ 3! &= 3 * (2 * (1 * 1))\,,\\ &\vdots \quad\quad \vdots \end{aligned}$$

on observe qu'il peut se récrire en

$$\begin{aligned} 0! &= 1\,,\\ 1! &= 1 * 0!\,,\\ 2! &= 2 * 1!\,,\\ 3! &= 3 * 2!\,,\\ &\vdots \quad\quad \vdots \end{aligned}$$

ce qui suggère l'écriture plus concise suivante :

$$n! =_{def} [\,\text{if } n = 0 \text{ then } 1 \text{ else } n * (n-1)!\,]\,.$$

On voit en particulier que, d'après cette définition,

$$4! = 4 * 3! = 4 * (3 * 2!) = 4 * (3 * (2 * 1!)) = 4 * (3 * (2 * (1 * 0!))) = 4 * (3 * (2 * (1 * 1)))\,.$$

La maîtrise de la récursivité est la première et la principale difficulté rencontrée lors de l'apprentissage de la programmation fonctionnelle.[7] Par ailleurs, l'approche fonctionnelle préserve entièrement le principal atout de la notion mathématique de fonction, à savoir la facilité avec laquelle des fonctions se composent et se combinent pour donner lieu à de nouvelles fonctions. Cela permet l'élaboration de fonctions très compliquées au moyen d'un petit nombre de mécanismes élémentaires produisant des fonctions à partir de fonctions primitives, données au départ, et/ou de fonctions antérieurement construites.

[5] Cette écriture est acceptable même pour $n = 0$ et $n = 1$; on a $0! = 1$ et $1! = 1$.

[6] De plus, ils souhaiteront peut-être lever le non-déterminisme et préciser l'ordre dans lequel les multiplications seront effectuées. C'est obligatoire pour le programmeur, sauf s'il utilise un langage non déterministe.

[7] C'est quasi la seule si, comme dans ce livre, on se limite à la programmation fonctionnelle élémentaire; cette simplicité est un avantage décisif par rapport à d'autres styles de programmation.

1.2 La récursivité

Le concept de récursivité étant essentiel, il est intéressant de l'aborder dans un cadre général, avant même de considérer son rôle particulier en programmation fonctionnelle. Le point important de cette courte étude est de permettre au lecteur de déterminer si une définition récursive est bien une définition, au sens habituel du terme. Nous verrons en effet qu'un texte ayant la syntaxe d'une définition récursive peut très bien ne rien définir du tout !

1.2.1 Principe et exemples

Une définition de fonction telle que

$$h(x, y) \ =_{def} \ \sqrt{x^2 + y^2}$$

présuppose l'existence des fonctions d'addition, d'élévation au carré et d'extraction de racine carrée. Plus précisément, on admettra la validité d'une égalité telle que

$$h(3, 4) = 5$$

en se basant sur des égalités concernant les trois fonctions auxiliaires déjà citées, en l'occurrence

$$3^2 = 9\,,\ 4^2 = 16\,,\ 9 + 16 = 25\,,\ \sqrt{25} = 5\,.$$

On le souligne, les égalités justificatives concernent seulement les fonctions auxiliaires, prédéfinies, et non pas la fonction h elle-même, nouvellement définie. On note également que le nom h intervient seulement à gauche du symbole de définition "$=_{def}$".

Considérons alors, sur le domaine des entiers naturels, la définition classique de la suite de Fibonacci :

$$\mathit{fib}(n) \ =_{def} \ [\,\text{if } n < 2 \text{ then } n \text{ else } \mathit{fib}(n-1) + \mathit{fib}(n-2)\,]\,.$$

Cette définition est *récursive* parce que le nom de l'entité que l'on définit, à savoir la fonction *fib*, intervient non seulement à gauche du symbole "$=_{def}$", mais aussi dans l'expression définissante, à droite de ce même symbole. On admet immédiatement les égalités

$$\mathit{fib}(0) = 0\,,\ \mathit{fib}(1) = 1\,,$$

parce que $\mathit{fib}(n)$ est explicitement spécifié égal à n si $n < 2$. Sur base de ces deux égalités évidentes, et des égalités supplémentaires

$$2 - 1 = 1\,,\ 2 - 2 = 0\,,\ 1 + 0 = 1\,,$$

on admet ensuite l'égalité $\mathit{fib}(2) = 1$ ou, plus explicitement, la chaîne d'égalités

$$\mathit{fib}(2) = \mathit{fib}(2-1) + \mathit{fib}(2-2) = \mathit{fib}(1) + \mathit{fib}(0) = 1 + 0 = 1\,;$$

de proche en proche, on constate que l'expression $fib(n)$ a une valeur bien précise pour tout entier naturel n. La différence essentielle entre la définition (non récursive) de la fonction h et celle (récursive) de la fonction *fib* est l'intervention, dans les égalités justificatives, de la fonction définie *fib* elle-même. On conçoit immédiatement le danger de la récursivité, assimilable à un "cercle vicieux". Ce danger existe : des "définitions" telles que

$$\begin{aligned} \alpha(x) &=_{def} \alpha(x)\,, \\ \beta(x) &=_{def} \beta(x+1)\,, \\ \gamma(x) &=_{def} \gamma(x)+1\,, \end{aligned}$$

sont clairement à déconseiller ! Nous verrons comment éviter le risque de cercle vicieux.

Beaucoup de relations de récurrence classiques de l'arithmétique se transforment aisément en définitions récursives de fonctions. Par exemple, la fonction *cb*, qui à tout couple d'entiers naturels (n, p) tels que $p \leq n$ associe le coefficient binomial correspondant,[8] donne lieu à la relation de récurrence bien connue

$$cb(n,p) = cb(n-1,p-1) + cb(n-1,p)\,,$$

valable à condition que p soit strictement compris entre 0 et n. Cette relation peut servir à définir la fonction *cb*; il suffit de préciser, en outre, la valeur de $cb(n,p)$ quand p vaut 0 ou n. On obtient ainsi une (première) définition récursive de la fonction *cb* :

$$cb1(n,p) =_{def} [\,\text{if } n{=}p \text{ or } p{=}0 \text{ then } 1 \text{ else } cb1(n{-}1,p{-}1) + cb1(n{-}1,p)\,]\,.$$

On voit notamment que

$$cb1(3,1) = cb1(2,0) + cb1(2,1) = 1 + cb1(1,0) + cb1(1,1) = 1+1+1 = 3\,.$$

Une autre relation de récurrence bien connue est

$$cb(n,p) = [n * cb(n-1,p-1)]/p\,,$$

valable si p n'est pas nul; il suffit de préciser la valeur de $cb(n,0)$ pour obtenir une seconde définition récursive :

$$cb2(n,p) =_{def} [\,\text{if } p = 0 \text{ then } 1 \text{ else } [n * cb2(n-1,p-1)]/p\,]\,.$$

On voit notamment que

$$cb2(4,2) = [4 * cb2(3,1)]/2 = [4 * 3 * cb2(2,0)/1]/2 = (4*3*1)/2 = 6\,.$$

Remarque. Les fonctions *cb*1 et *cb*2 sont mathématiquement égales, puisqu'elles contiennent exactement les mêmes couples. Pour l'informaticien, elles sont cependant très différentes. En particulier, comme nous le verrons plus loin, *cb*1 est inefficace mais *cb*2 est efficace. La différence est considérable.[9]

[8] c'est-à-dire le coefficient de x^p dans le développement du binôme $(1+x)^n$.

[9] Par exemple, l'évaluation de $cb1(20,10) = 184\,756$ prend beaucoup plus de temps que celle de $cb2(200,100) = 90548514656103281165404177077484163874504589675413336841320$; le lecteur verra comment le vérifier expérimentalement au chapitre suivant.

1.2.2 Domaine de validité

Les définitions récursives des fonctions f, $cb1$ et $cb2$ sont acceptables parce que la détermination de la valeur associée à un point quelconque de leur domaine exige une suite *finie* de calculs, puisque les développements sont finis. Une définition telle que

$$g(n) \;=_{def}\; [\,\text{if } n<2 \text{ then } n \text{ else } g(n-1)+g(n+2)\,]\,,$$

qui ressemble à la définition de la fonction de Fibonacci *fib* donnée plus haut, n'a pas cette propriété :

$$g(2)=g(1)+g(4)=1+g(3)+g(6)=1+g(2)+g(5)+g(5)+g(8)=\cdots$$

Même si, par extraordinaire, il existait une fonction g vérifiant l'*égalité*

$$g(n) \;=\; [\,\text{if } n<2 \text{ then } n \text{ else } g(n-1)+g(n+2)\,]$$

pour tout entier naturel n, on ne considérerait pas la définition précédente comme une définition (récursive) acceptable, notamment parce que l'expression $g(2)$ donne lieu à un développement infini.

Pour préciser notre notion d'acceptabilité, considérons un cas célèbre (l'expression $even?(n)$ est vraie si n est pair et fausse sinon) :

$$\begin{array}{rl} t(n) \;=_{def} & \text{if } n=1 \text{ then } 1 \\ & \text{else if } even?(n) \text{ then } t(n/2) \\ & \text{else } t(3n+1)\,. \end{array}$$

On pense que ce texte définit la fonction t qui à tout entier strictement positif associe le nombre 1 ... mais cela n'a pas été démontré.[10] On a par exemple

$$t(52)=t(26)=t(13)=t(40)=t(20)=t(10)=t(5)=t(16)=t(8)=t(4)=t(2)=t(1)=1\,.$$

1.2.3 Terminaison, totalité

En pratique, il semble peu réaliste de développer une étude mathématique parfois difficile chaque fois qu'une définition récursive d'une fonction f sur le domaine des entiers naturels est envisagée, dans le seul but de prouver que le calcul de $f(n)$ se termine pour tout n, c'est-à-dire que la fonction f est *partout définie*, ou encore *totale*, sur le domaine $\mathbb{N}$.

[10] *A priori*, la fonction t divise l'ensemble des naturels en deux parties. La première contient tout nombre pour lequel le calcul se termine et la seconde tout nombre pour lequel le calcul ne se termine pas. Si n appartient à la première partie, on a $t(n)=1$; sinon, $t(n)$ n'est pas défini. La conjecture qui, à ce jour, n'a pas été démontrée, est que la seconde partie est vide.

Heureusement, dans beaucoup de cas, la conclusion est évidente. Considérons la définition suivante, d'apparence compliquée :

$$\begin{aligned} h(n) \;=_{def}\quad & \text{if } n = 0 \text{ then } 1 \\ & \text{else if } n = 1 \text{ then } 5 \\ & \text{else if } \mathit{even}?(n) \text{ then } h(n/2) + h(n-2) \\ & \text{else } h((n-3)/2) + h(n-1) * h((n+1)/2)\,. \end{aligned}$$

La valeur de $h(n)$ est susceptible de dépendre des valeurs de $h(n/2)$, $h(n-1)$, $h((n-3)/2)$, $h(n-2)$ et $h((n+1)/2)$, mais on observe que, chaque fois que $h(n)$ dépend de $h(m)$, l'argument m est un entier naturel, strictement inférieur à m.

En conclusion, la fonction h est totale sur le domaine $\mathbb{N}$ et il est possible de calculer $h(n)$, quel que soit l'entier naturel n; un moyen simple, et raisonnablement efficace, consiste à calculer successivement $h(0)$, $h(1)$, ..., $h(n)$. En particulier, on a :

n	0	1	2	3	4	5	6	7	8	9
$h(n)$	1	5	6	37	12	449	49	594	61	27426

Notons que, en pratique, la généralisation consistant à permettre que $f(n)$ dépende non seulement de $f(n-1)$ mais aussi de n'importe quel $f(i)$ où $i = 0, \ldots, n-1$, est rarement nécessaire; au contraire, on observe que le cas particulier où $f(n)$ est défini en terme de $f(n-1)$ seulement (si $n > 0$) suffit le plus souvent. C'est vrai notamment dans le cas des nombres de Fibonacci, moyennant une petite astuce. En effet, plutôt que de définir immédiatement la fonction *fib*, on peut définir la fonction *fib2* comme suit :

$$\begin{aligned} \mathit{fib2}(0) \;&=_{def}\; (0,1)\,, \\ \mathit{fib2}(n+1) \;&=_{def}\; (b, a+b)\,, \text{ où } (a,b) = \mathit{fib2}(n)\,. \end{aligned}$$

On observe que $\mathit{fib2}(n)$ dépend de $\mathit{fib2}(n-1)$ (si $n > 0$) mais pas de $\mathit{fib2}(n-2)$. On peut alors démontrer que, pour tout naturel n, on a

$$\mathit{fib2}(n) \;=\; (\mathit{fib}(n), \mathit{fib}(n+1))\,.$$

On procède par récurrence. Pour $n = 0$, c'est vrai par construction. Si $n > 0$, on peut supposer que la propriété est vraie pour $n-1$ et donc que $\mathit{fib2}(n-1) = (\mathit{fib}(n-1), \mathit{fib}(n))$. Par construction on a $\mathit{fib2}(n) = (\mathit{fib}(n), \mathit{fib}(n-1) + \mathit{fib}(n))$, c'est-à-dire $\mathit{fib2}(n) = (\mathit{fib}(n), \mathit{fib}(n+1))$.

1.2.4 Récurrence simple, récurrence complète

Le principe de récurrence vient d'être utilisé pour démontrer une certaine relation existant entre deux fonctions définies récursivement. Plus généralement, il sera largement utilisé par la suite pour démontrer les propriétés des programmes SCHEME. Il est donc utile de revenir sur ce principe ici. L'idée de base est que tout naturel est, soit 0, soit le successeur d'un autre naturel, que l'on nomme son prédécesseur. Le principe de récurrence est un

mécanisme de raisonnement qui, en logique formelle, se représente par la règle d'inférence[11] suivante :

$$\frac{P(0)\,,\ \forall n > 0\,[P(n-1) \Rightarrow P(n)]}{\forall n\, P(n)}$$

Cette règle est la traduction formelle d'un énoncé bien connu :

> *Si la propriété P est vraie du nombre naturel 0,*
> *si dès qu'elle est vraie d'un naturel $n-1$, elle l'est aussi de n,*
> *alors elle est vraie de tout naturel n.*

Intuitivement, le principe de récurrence est évident car les prémisses peuvent se récrire en

$$P(0)\,,\ P(0) \Rightarrow P(1)\,,\ P(1) \Rightarrow P(2)\,, \ldots$$

dont on déduit de proche en proche

$$P(0)\,,\ P(1)\,,\ P(2)\,, \ldots$$

c'est-à-dire la conclusion. On justifie plus formellement le principe de récurrence sans aucune difficulté. Supposons, par l'absurde, qu'une certaine propriété P vérifie les deux prémisses mais pas la conclusion. L'ensemble des naturels pour lesquels la propriété est fausse ne serait pas vide et admettrait donc un minimum m. Cependant, m ne peut pas être 0 à cause de la première prémisse; il ne peut non plus être le successeur de $m-1$, car on aurait $P(m-1)$ sans avoir $P(m)$, ce qui contredirait la seconde prémisse. On a obtenu une contradiction : le nombre m ne peut donc pas exister.

Le principe de récurrence présenté ci-dessus est en fait le principe de récurrence simple; le principe de récurrence complète prévoit que, pour établir $P(n)$, on peut disposer de $P(n-1)$ mais aussi de $P(n-2), \ldots, P(0)$. La récurrence complète semble donc plus puissante que la récurrence simple, mais elle lui est en fait équivalente. Cette variante s'énonce classiquement comme suit :

> *Si dès que la propriété P est vraie des nombres naturels $0, \ldots, n-1$*
> *elle l'est aussi de n,*
> *alors elle est vraie de tout naturel n.*

La règle d'inférence formelle correspondante est

$$\frac{\forall n\,[(\forall m < n\ P(m)) \Rightarrow P(n)]}{\forall n\, P(n)}$$

Intuitivement, le principe de récurrence complète est évident car la prémisse peut se récrire en

$$P(0)\,,\ P(0) \Rightarrow P(1)\,,\ (P(0) \wedge P(1)) \Rightarrow P(2)\,, \ldots$$

[11] Une *règle d'inférence* est une notation du type $\frac{P_1, \ldots, P_m}{C}$; les P_i sont des énoncés nommés *prémisses*; C est un énoncé nommé *conclusion*. La règle est *correcte* si la conclusion est *conséquence logique* des prémisses, c'est-à-dire si, dans toute situation où chaque prémisse est vraie, la conclusion est également vraie. Rappelons aussi que "$a \Rightarrow b$" est "a implique b" ou "si a alors b"; "$a \wedge b$" est "a et b"; "$\forall n\ \Phi$" est "pour tout n, Φ".

dont on déduit de proche en proche

$$P(0)\,,\ P(1)\,,\ P(2)\,,\ldots$$

c'est-à-dire la conclusion. La preuve formelle est quasi la même que pour la récurrence simple.

1.2.5 Notation lambda

Dans une écriture du type "$X\ =_{def}\ \mathcal{A}$", le terme défini est normalement X, tandis que l'expression définissante est $\mathcal{A}$. Cette constatation semble évidente, mais n'est pas parfaitement respectée dans les définitions fonctionnelles données plus haut (qu'elles soient récursives ou non). En effet, nous avons écrit "$f(n)\ =_{def}\ \mathcal{A}$", alors que le terme défini était f et non $f(n)$. Une meilleure écriture, parfois utilisée par les mathématiciens, est

$$f\ =_{def}\ [n \mapsto \mathcal{A}]\,;$$

dans le présent contexte, on utilisera plutôt la "notation lambda"[12] et on écrira

$$f\ =_{def}\ \lambda n.\mathcal{A}\,.$$

L'expression définissante est une "forme lambda",[13] dont $\mathcal{A}$ est le "corps". On écrira, par exemple,

$$h\ =_{def}\ \lambda xy.\sqrt{x^2+y^2}\,,$$

ce qui se lit "h est la fonction qui, à tous x, y, associe $\sqrt{x^2+y^2}$". De même, la définition classique de la fonction de Fibonacci se récrit

$$\mathit{fib}\ =_{def}\ \lambda n.\,[\,\text{if } n<2 \text{ then } n \text{ else } \mathit{fib}(n-1)+\mathit{fib}(n-2)\,]\,.$$

1.2.6 Aspects opérationnels de la récursivité

Nous verrons que toutes les définitions fonctionnelles données plus haut se transposent immédiatement en SCHEME.

Une définition acceptable donne lieu à une procédure qui se termine toujours.[14] Cependant, l'efficacité de la procédure peut grandement varier et, dans certains cas, se révéler inacceptable. Considérons d'abord le programme

[12] Ce nom fait référence au "lambda-calcul", un formalisme abstrait proposé par A. Church en 1936 pour étudier les notions d'algorithme et de calculabilité.

[13] On dit aussi, d'après l'anglais, "lambda-forme", ce que l'on écrira dans la suite, "λ-forme".

[14] Nous emploierons souvent les mots "fonction" et "procédure" de manière interchangeable. Le mot "procédure" insiste sur le fait que la fonction (au sens mathématique du terme) se double d'un programme, c'est-à-dire d'un outil permettant le calcul des couples de cette fonction. A la notion mathématique de "fonction définie pour la valeur x de l'argument" correspond la notion informatique de "procédure se terminant pour la valeur x de l'argument". La notion d'efficacité (en programmation) concerne la procédure et non la fonction mathématique sous-jacente. Deux procédures très différentes peuvent avoir même fonction mathématique sous-jacente. Par exemple, il existe de nombreux algorithmes pour trier une liste d'entiers mais ils partagent tous la même fonction mathématique sous-jacente.

```
(define fib
  (lambda (n)
    (if (< n 2)
        n
        (+ (fib (- n 1)) (fib (- n 2))))))
```

Même sans connaître le langage SCHEME, on devine que ce programme est une transcription naturelle immédiate de la définition de la suite de Fibonacci. Il se termine toujours, mais on constate que le temps d'exécution est anormalement long. La raison est simple; pour évaluer `(fib 33)`, par exemple, le système évalue *séparément* `(fib 32)` et `(fib 31)`; il s'agit clairement d'un gaspillage, puisque tout calcul fait dans le cadre de l'évaluation de `(fib 31)` est aussi fait dans le cadre de l'évaluation de `(fib 32)`. On peut facilement vérifier que `(fib 31)` est calculé deux fois, `(fib 30)` trois fois, `(fib 29)` cinq fois, etc. On peut éviter ce gaspillage de ressources en utilisant plutôt la fonction *fib2* introduite au paragraphe précédent.[15] L'évaluation de *fib2*(33) en (3524578, 5702887) est maintenant immédiate; bien plus, l'évaluation de *fib2*(3300) reste quasi instantanée, quoiqu'elle produise deux nombres de 691 chiffres. Un programme d'efficacité comparable s'obtient en transposant en SCHEME la définition suivante :

$$\mathit{fib\_it}(n, a, b) \;=_{\mathit{def}}\; [\text{if } n = 0 \text{ then } a \text{ else } \mathit{fib\_it}(n-1, b, a+b)]\,.$$

En effet, on démontre facilement que, pour tous nombres de Fibonacci consécutifs $\mathit{fib}(i)$ et $\mathit{fib}(i{+}1)$, on a

$$\mathit{fib\_it}(n, \mathit{fib}(i), \mathit{fib}(i{+}1)) = \mathit{fib}(i{+}n)$$

et donc, en particulier

$$\mathit{fib\_it}(n, 0, 1) = \mathit{fib}(n)\,,$$

pour tout naturel n. Le temps de calcul sera quasi proportionnel à n, alors qu'avec le programme précédent il était proportionnel à $\mathit{fib}(n)$.

Quoique cela ne soit guère utile, on peut encore faire mieux, en notant que

$$\begin{pmatrix} \mathit{fib}(n) \\ \mathit{fib}(n+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

En effet, que M soit un nombre ou une matrice, on peut calculer M^n en un temps proportionnel à $\log n$, en adaptant l'algorithme classique dit "exponentielle rapide" :

$$\begin{aligned} \mathit{exp}(M, n) \;=_{\mathit{def}}\quad & \text{if } n = 0 \text{ then I (matrice unité)} \\ & \text{else if } \mathit{even?}(n) \text{ then } [\mathit{exp}(M, n/2)]^2 \\ & \text{else } M * \mathit{exp}(M, n-1)\,. \end{aligned}$$

On notera que cette solution très efficace est récursive, de même que la solution naïve, très inefficace, donnée plus haut. Contrairement à une opinion tenace, la récursivité n'est pas incompatible avec l'efficacité.

[15] Nous verrons la transposition en SCHEME plus loin.

Dans la mesure où un programme est, avant tout, une entité de commande pour une machine, la notion d'efficacité est très importante. Nous avons déjà mentionné au paragraphe 1.2.1 que les programmes *cb*1 et *cb*2, quoique réalisant tous deux la même fonction mathématique, différaient radicalement dans les processus de calcul mis en œuvre. Le lecteur peut immédiatement observer que la définition *cb*1 est inefficace parce qu'elle implique des recalculs (exactement comme la version naïve du programme de Fibonacci), alors que la version *cb*2 n'en implique pas et est efficace.

1.3 Structures de données

Les données constituent un aspect essentiel de la programmation et des langages de programmation. D'une part, les caractéristiques des données d'un problème influencent largement la manière dont le programmeur tentera de résoudre le problème. Nous venons de voir que, si l'une des données d'un problème est un entier naturel n, une stratégie souvent féconde consiste à considérer d'abord le cas $n = 0$, puis à essayer de réduire le cas n au cas $n–1$ si $n > 0$. D'autre part, les mécanismes de structuration, de représentation et de manipulation de données qu'offrent les différents langages de programmation déterminent, dans une large mesure, l'adéquation de ces langages aux problèmes mettant en jeu ces données.

Les exemples considérés jusqu'ici étaient numériques parce que le type de donnée "entier naturel" est à la fois le plus familier et le plus important. Néanmoins, d'autres types de données importants existent. D'une part, certaines données élémentaires ne sont pas numériques, mais plutôt symboliques. Il est naturel, par exemple, de représenter les données "couleurs de l'arc-en-ciel" par les symboles "violet", "indigo", "bleu", "vert", "jaune", "orange" et "rouge", plutôt que par les nombres 0, 1, 2, 3, 4, 5 et 6. D'autre part, et ceci est plus important en pratique, les données élémentaires, numériques ou symboliques, apparaissent souvent sous forme d'un groupe de données bien structuré plutôt que sous forme de nombres ou de symboles isolés. Par exemple, une donnée de type "matrice 2×2" comporte quatre composants numériques; on peut imaginer qu'une donnée de type "personne" comporterait deux composants symboliques (nom et prénom) et un composant numérique (année de naissance). Une donnée de type "famille" sera un arbre (généalogique) dont les nœuds sont étiquetés par des données de type "personne". Toute donnée formée de composants est dite *structurée*.

Avant de programmer, c'est-à-dire de manipuler des données, il est utile de s'interroger sur la notion même de donnée. Dans ce paragraphe, nous décrivons d'abord trois types de données, à la fois importants et représentatifs; nous esquissons une technique abstraite et axiomatique de description des types de données qui donnera lieu dans la suite à un schéma général pour les programmes manipulant ces données. Nous montrons ensuite comment, à partir de types ainsi définis, on peut dériver des variantes et cas particuliers utiles.

1.3.1 Les entiers naturels

Les entiers naturels constituent un type de donnée bien connu et il peut paraître artificiel et vain de vouloir les définir. Toutefois, il est utile de les présenter ici d'une manière qui met en évidence l'intérêt du principe de récurrence dans le raisonnement à propos des entiers naturels et aussi dans la construction des programmes qui les manipulent. On se donne d'abord deux règles de construction :

- 0 est un naturel.
- Si n est un naturel, alors $s(n)$ est un naturel.

La fonction à un argument s (pour "successeur") est le *constructeur* du type "entier naturel". Des deux règles de construction, on déduit que 0, $s(0)$, $s(s(0))$, etc., sont des naturels (que l'on peut noter plus économiquement 0, 1, 2, etc.). On admet aussi que tout naturel peut s'obtenir de la sorte, et ceci d'une seule manière. On se donne enfin une fonction p (pour "prédécesseur"), inverse de s, qui à tout naturel non nul $s(n)$ associe le naturel n. L'ensemble des entiers naturels, présenté de la sorte, est le *domaine* des naturels, que nous noterons classiquement $\mathbb{N}$. On dit aussi que 0 est un *élément de base* de ce domaine (en fait, le seul); en outre, tout naturel n est un *composant direct* du naturel $s(n)$ (en fait, le seul) d'où, si m est un naturel non nul, $p(m)$ est son composant direct. Un *composant* d'un naturel non nul $s(n)$ est soit le composant direct n de ce naturel, soit un composant (direct ou non) de n.[16]

Cette présentation permet de reformuler les principes de récurrence comme suit :

> *(Récurrence simple.)*
> *Si la propriété P est vraie du naturel de base 0*
> *et si dès qu'elle est vraie du composant direct d'un naturel,*
> *elle l'est aussi du naturel lui-même,*
> *alors elle est vraie de tout naturel.*
>
> *(Récurrence complète.)*
> *Si dès que la propriété P est vraie des composants d'un naturel,*
> *elle l'est aussi du naturel lui-même,*
> *alors elle est vraie de tout naturel.*

Cette nouvelle formulation suggère que la notion de composant est la clef du principe de récurrence, et que ce dernier pourrait se généraliser à tout domaine structuré, c'est-à-dire à tout ensemble dont les éléments se partitionnent en objets de base et en objets composés au moyen des objets de base.[17] C'est bien le cas et cela permet d'étendre à des données non numériques la technique de définition récursive de fonctions. Nous ne traitons pas ici

[16] La notion de composant est ici définie récursivement. On pourrait aussi dire que m est un composant de n si n s'obtient en appliquant un certain nombre de fois le constructeur s au naturel m; plus simplement, m intervient dans la construction de n.

[17] En tant que type de donnée structuré, le domaine $\mathbb{N}$ est structuré. Le zéro est l'unique élément de base, les fonctions successeur et prédécesseur sont respectivement le *constructeur* et l'*accesseur* du domaine.

cette question dans toute sa généralité, mais la suite de ce paragraphe introduit quelques domaines structurés importants.

1.3.2 Les arbres binaires

Considérons un ensemble non vide E. On associe à E un domaine structuré dont les éléments sont appelés *E-arbres binaires*, ou simplement *arbres binaires*. On se donne deux règles de construction :

- Tout élément de E est un arbre binaire.
- Si A et B sont des arbres binaires, alors $c(A, B)$ est un arbre binaire.

Les arbres binaires les plus simples sont les *feuilles*, éléments de E. Les autres arbres sont dits *arbres composés*. La fonction à deux arguments c est le constructeur du type "arbre binaire". Un objet est un arbre binaire si et seulement si cet objet peut être construit au moyen des deux règles de construction, appliquées un nombre fini de fois. Si c'est le cas, le mode de construction est unique; des arbres construits de manières distinctes sont toujours distincts. Par exemple, pour tous $x, y, z \in E$, les deux arbres $c(c(x, y), z)$, $c(z, c(x, y))$ sont distincts (même si x, y et z sont égaux); d'autre part, les deux arbres $c(c(x, y), z)$ et $c(c(x', y'), z')$ sont égaux si et seulement si $x = x'$, $y = y'$ et $z = z'$.

On se donne aussi deux fonctions a et d à un argument, appelées respectivement *fils gauche* et *fils droit* qui, à tout arbre composé $c(\alpha, \beta)$ associent respectivement ses *composants directs* α (aussi appelé fils gauche) et β (aussi appelé fils droit); ces deux fonctions sont les *accesseurs*.[18] Les *composants* d'un arbre binaire composé sont ses composants directs ou les composants de ceux-ci. Un arbre binaire α est un composant d'un arbre binaire β si α intervient dans la construction de β.
Remarque. Les éléments de E peuvent être variés (des nombres, des lettres, des symboles quelconques) mais ils ne peuvent être des arbres binaires; plus spécifiquement, ils ne peuvent être le résultat de l'application du constructeur c à des arguments. Dans la suite, nous appelons *atome* tout objet de ce type. On notera que les accesseurs a et d ne peuvent prendre pour argument un atome.

Les principes de récurrence se généralisent aux arbres binaires (ainsi qu'aux autres domaines structurés dont quelques exemples supplémentaires sont introduits ci-après); on les appelle alors *principes d'induction*. Dans ces énoncés, le symbole P désigne une propriété, susceptible d'être vraie ou fausse.

> *(Induction simple sur les arbres binaires.)*
> *Si P est vrai de toute feuille, élément de E*
> *et si dès que P est vrai des deux fils d'un E-arbre binaire composé,*
> *P l'est aussi du E-arbre binaire composé lui-même,*
> *alors P est vrai de tout E-arbre binaire.*

[18] Le constructeur d'arbre et les accesseurs ont été nommés respectivement c, a et d; notons déjà que leurs homologues SCHEME se nomment `cons`, `car` et `cdr`.

(Induction complète sur les arbres binaires.)
Si dès que P est vrai des composants d'un E-arbre binaire,
P l'est aussi du E-arbre binaire lui-même,
alors P est vrai de tout E-arbre binaire.

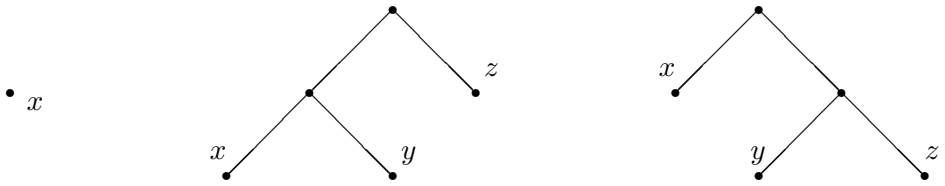


Figure 1: Représentations des arbres binaires x, $c(c(x,y),z)$ et $c(x,c(y,z))$

Remarque. Les arbres binaires doivent leur nom à leur représentation habituelle (cf. Figure 1); c'est de là aussi que provient la terminologie usuelle : les composants directs sont appelés "enfants" ou "fils"; les composants sont aussi appelés "descendants" ou "sous-arbres". Par rapport à un arbre donné, l'arbre lui-même est la *racine*; la racine et les sous-arbres composés forment les *nœuds internes*. On notera que les nœuds internes, au contraire des feuilles, ne sont pas étiquetés; il est parfois commode de considérer que chaque nœud interne est implicitement étiqueté par le sous-arbre dont il est la représentation.[19]
Remarque. La notion d'arbre binaire se généralise en celle d'arbre n-aire (tout nœud interne a exactement n fils; il y a un constructeur à n arguments et aussi n accesseurs. En ce sens, les entiers naturels ne sont rien d'autre que les $\{0\}$-arbres unaires; le constructeur (unaire) est la fonction successeur et les entiers naturels sont 0, $s(0)$, $s(s(0))$, etc. L'unique accesseur est la fonction prédécesseur. Les arbres n-aires sont appelés *arbres à degré fixe*; chaque nœud interne a exactement n fils.

1.3.3 Les listes

Intuitivement, une *liste* est une suite totalement ordonnée, comportant un nombre quelconque d'objets appelés *éléments* de la liste. Ce nombre d'objet est la *longueur* de la liste. On pourrait formaliser la notion de liste en créant un domaine dont le constructeur admettrait un nombre quelconque d'arguments mais il serait difficile de s'accommoder d'un nombre variable d'accesseurs. On utilisera donc une autre tactique pour formaliser le domaine des listes. Le constructeur prend deux arguments, un objet α et une liste β; il produit une liste dont le premier élément est α et dont les autres éléments forment la liste β; on dispose, pour commencer la construction des listes, de la liste vide notée [ ] et d'un ensemble non vide E d'éléments de base. Réciproquement, à toute liste non vide

[19] Dans la figure 1, la représentation du milieu comporte trois étiquettes explicites de feuilles; les étiquettes implicites des nœuds internes sont $c(c(x,y),z)$ pour la racine et $c(x,y)$ pour l'autre nœud, fils gauche de la racine.

on peut appliquer deux accesseurs, le premier fournissant la *tête*, c'est-à-dire le premier élément de la liste, et le second renvoyant le *reste*, c'est-à-dire la liste des éléments restants. Comme dans le cas des arbres binaires, le constructeur et les deux accesseurs seront notés respectivement c, a et d. Une liste dont les trois éléments sont, dans l'ordre, x, y et z, sera donc notée $c(x, c(y, c(z, [\,])))$. On devine que cette notation devient très incommode pour les longues listes, aussi on utilisera souvent la notation abrégée $[x, y, z]$. Des notations mixtes, telles $c(x, [y, z])$ et $c(x, c(y, [z])$ sont également admises.
Remarque. Il est important de distinguer les termes "élément" et "composant". La liste vide n'a ni élément ni composant. Une liste non vide a exactement deux composants directs, la tête qui est un élément, et le reste qui est une liste. La liste $[x, y, z]$ admet les trois éléments x, y et z et les deux composants direct x et $[y, z]$; elle admet en outre les composants indirects y, $[z]$, z et $[\,]$. La liste $[x]$ admet le seul élément x; sa tête est x et son reste est $[\,]$; il n'y a pas de composant indirect.

1.3.4 Listes plates, listes profondes

La notion de liste qui vient d'être introduite donne lieu à un domaine structuré à condition que l'on spécifie l'ensemble des objets qui peuvent être éléments d'une liste.

Etant donné un ensemble de base E non vide, dont les éléments sont des atomes, nous considérerons deux domaines de listes, notés respectivement $\mathcal{L}_-^E$ et $\mathcal{L}^E$.

- Les éléments des listes de $\mathcal{L}_-^E$ appartiennent à E (et ne sont donc pas eux-mêmes des listes). L'ensemble $\mathcal{L}_-^E$ est le domaine des *E-listes plates*.
- Les éléments des listes de $\mathcal{L}^E$ appartiennent à l'ensemble $E \cup \mathcal{L}^E$. L'ensemble $\mathcal{L}^E$ est le domaine des *E-listes*.

On note l'inclusion $\mathcal{L}_-^E \subset \mathcal{L}^E$; cette inclusion est stricte. On appelle *E-liste profonde* une E-liste qui n'est pas plate.

Pour illustrer ces définitions, considérons le cas particulier $E = \{0, 1\}$. La seule E-liste de longueur nulle est la liste vide, qui est une liste plate. Il existe deux listes plates de longueur 1, qui sont [0] et [1], et quatre listes plates de longueur 2, qui sont [0,0], [0,1], [1,0], [1,1]. Le monde des listes profondes est beaucoup plus touffu, comme le montrent quelques exemples simples : [[ ]], [[0]], [[0,1,[0,1]]], [0,[ ]], [[0],0], [[[ ]],[0,[1],[0,[1]]]]. Les trois premières listes sont de longueur 1, les trois dernières sont de longueur 2. Notons aussi trois points importants, qui paraphrasent les définitions :

- $[\,]$ est une E-liste et une E-liste plate; sa longueur est 0.
- Si α est un élément de E et si β est une E-liste plate de longueur n, alors $c(\alpha, \beta)$ est une E-liste plate de longueur $n+1$.
- Si α est un élément de E ou une E-liste et si β est une E-liste de longueur n, alors $c(\alpha, \beta)$ est une E-liste de longueur $n+1$.

Remarque. En termes algébriques, les deux domaines de listes se définissent de manière concise; ce sont les solutions minimales des équations ensemblistes $\mathcal{L}_-^E = \{[\,]\} \cup c(E, \mathcal{L}_-^E)$ et $\mathcal{L}^E = \{[\,]\} \cup c(E \cup \mathcal{L}^E, \mathcal{L}^E)$, respectivement.

On peut décrire une liste non vide en termes de ses éléments ou en termes de ses composants. La seconde solution met en évidence le fait que toute E-liste est aussi un $(E \cup \{[\,]\})$-arbre binaire (la réciproque est fausse). Ces deux représentations sont illustrées à la figure 2.

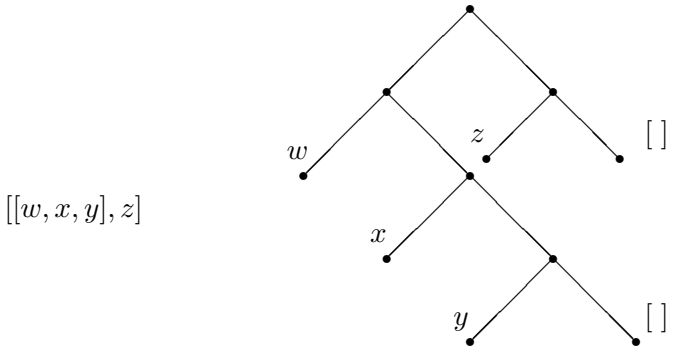


Figure 2: Représentations de la liste $c(c(w, c(x, c(y, [\,]))), c(z, [\,]))$

Remarque. Comme dans le cas des arbres binaires, seules les feuilles sont explicitement étiquetées. Il peut être utile d'attribuer une étiquette implicite à chaque nœud interne. Dans la figure 2, l'étiquette de la racine est la liste elle-même; l'étiquette de son fils gauche est la liste $c(w, c(x, c(y, [\,])))$, c'est-à-dire la liste $[w, x, y]$; l'étiquette du fils droit de la racine est $c(z, [\,])$, c'est-à-dire la liste $[z]$.

Remarque. La présentation que nous avons adoptée ici assimile les listes à des arbres binaires particuliers. Cela n'empêche pas que les listes (profondes) soient souvent utilisées pour modéliser en SCHEME d'autres types d'arbres. Nous reviendrons sur ce point au paragraphe 8.4.

L'intérêt de présenter les listes comme des domaines structurés est de pouvoir raisonner par induction à leur sujet. On a d'abord les principes d'induction suivants :

(Induction superficielle simple sur les listes.)
Si P est vrai de la liste vide
et si dès que P est vrai du reste d'une liste non vide,
P l'est aussi de la liste elle-même,
alors P est vrai de toute liste.

(Induction superficielle complète sur les listes.)
Si dès que P est vrai des suffixes propres [20] *d'une liste,*

[20] Un *suffixe propre* d'une liste non vide est le reste de cette liste, ou un suffixe propre de ce reste; l'unique

P l'est aussi de la liste elle-même,
alors P est vrai de toute liste.

L'induction superficielle est spécialement adaptée aux listes plates. Dans certains cas, on l'utilise aussi pour les listes. Cependant, pour les listes (quelconques), on utilise souvent des principes un peu plus forts; l'idée est la suivante : dans la partie inductive du principe, l'hypothèse est que la propriété est vraie non seulement pour le second composant, qui est toujours une liste, mais aussi pour le premier, quand il s'agit d'une liste. Le symbole P dénote ici une propriété de liste, c'est-à-dire un prédicat à un argument tel que, pour toute liste ℓ, $P(\ell)$ est vrai ou faux. On a alors

(Induction profonde simple sur les listes.)
Si P est vrai de la liste vide;
si pour toute liste non vide dont la tête est atomique,
dès que P est vrai du reste de la liste,
P l'est aussi de la liste elle-même;
si pour toute liste non vide dont la tête est une liste,
dès que P est vrai de la tête et du reste de la liste,
P l'est aussi de la liste elle-même;
alors P est vrai de toute liste.

(Induction profonde complète sur les listes.)
Si dès que P est vrai de la liste vide
et des composants non atomiques d'une liste,
P l'est aussi de la liste elle-même,
alors P est vrai de toute liste.

L'expression de ces principes est un peu lourde parce que $P(\ell)$ n'a de sens que si ℓ est une liste. Si on convient que $P(x)$ est d'office vrai si x n'est pas une liste, on obtient des énoncés plus simples :

(Induction profonde simple sur les listes.)
Si P est vrai de la liste vide;
si dès que P est vrai de la tête et du reste d'une liste non vide,
P l'est aussi de la liste elle-même;
alors P est vrai de toute liste.

(Induction profonde complète sur les listes.)
Si dès que P est vrai des composants d'une liste,
P l'est aussi de la liste elle-même,
alors P est vrai de toute liste.

1.4 Apprendre à programmer avec SCHEME

Programmer n'est sans doute pas, en soi, une activité plus difficile que beaucoup d'autres. Pourtant, on se plaint (depuis au moins trente ans !) de la qualité insuffisante des

suffixe impropre d'une liste, vide ou non, est la liste elle-même.

programmes et de la productivité insuffisante des programmeurs. Nous n'allons pas ici nous engager dans ce thème maintes fois débattu, mais seulement donner quelques idées qui nous ont guidé dans la rédaction de ce texte.

Un premier point est que la programmation, comme beaucoup d'activités, requiert une certaine somme de connaissances préliminaires, sans lesquelles il n'est même pas vraiment possible de *commencer* à pratiquer. Le double choix du style fonctionnel et du langage SCHEME réduit, à notre avis, cette quantité de connaissances prérequises. D'une part, il suffit de maîtriser le petit monde des expressions arithmétiques élémentaires pour comprendre le style fonctionnel. D'autre part, même si le langage SCHEME comporte un certain nombre de notions et de mécanismes difficiles, un sous-ensemble relativement réduit du langage, restreint à des constructions élémentaires, suffit à résoudre d'une manière satisfaisante un grand nombre de problèmes intéressants; dans le même ordre d'idée, la syntaxe de SCHEME est extrêmement simple.

Un deuxième point est que la programmation, c'est-à-dire la construction d'une solution informatique à un problème, requiert une certaine créativité. L'approche que nous suivons ici ne supprime pas cette nécessité, mais rend plus aisée sa satisfaction. Plus concrètement, nous pensons que l'élaboration d'un programme peut se faire méthodiquement et que seules certaines étapes requièrent de la créativité. Nous attachons un soin tout particulier, dans la suite, à circonscrire la partie créative de la résolution d'un problème.

Un troisième point est que la programmation requiert une certaine discipline ... et que l'efficience des outils actuels pourrait malencontreusement induire le programmeur à recourir à la "méthode" dite "essais et erreurs". On croit qu'un essai ne coûte rien : c'est faux, il coûte du temps ![21] De plus, un programme auquel on arrive au terme d'une longue série d'essais, d'erreurs et de corrections a peu de chances d'être entièrement correct, et encore moins de chances d'être raisonnablement "propre". Il ne s'agit pas ici de recommander une approche "puriste", où l'apprentissage de la programmation serait vu comme une activité d'écriture, sans contact avec la machine; simplement, la programmation fait partie des "arts de l'ingénieur", qui requièrent, en dosage approprié, réflexion et expérimentation.

[21] Il coûte du temps d'ordinateur, ce qui, de nos jours, n'est pas grave, mais aussi du temps humain, ressource coûteuse et limitée ...

2 Les bases de SCHEME

2.1 Principe de l'interprète

Le langage SCHEME peut être interprété ou compilé; nous utiliserons surtout l'interprète. L'interaction entre l'interprète et l'utilisateur est en principe très simple; c'est une séquence de quatre étapes, répétée aussi longtemps que l'utilisateur le souhaite :[22]

- L'utilisateur introduit une expression, c'est-à-dire un texte SCHEME respectant certaines règles;
- Le système lit l'expression;
- Le système évalue l'expression;
- Le système affiche la valeur de l'expression (ou un message approprié si l'expression n'a pas de valeur).

Le rôle essentiel de l'interprète consiste donc à *évaluer* les expressions qui lui sont soumises, c'est-à-dire à en calculer la *valeur*. L'interprète SCHEME, que nous appellerons souvent "le système", se comporte un peu comme une calculatrice de poche; une machine de ce type accepte des expressions arithmétiques et produit leurs valeurs numériques.

2.2 Les expressions

On distingue trois sortes d'expressions : les expressions simples, les combinaisons et les formes spéciales.

- Une *expression simple* est une *constante* (numérique ou non) ou une *variable*. Signalons déjà que la valeur d'une constante est cette constante, tandis que la valeur d'une variable peut être un objet quelconque.
- Une *combinaison* est une liste; nous appellerons le premier élément *foncteur* ou *opérateur*; les autres éléments sont les *opérandes*.[23] Les éléments d'une combinaison sont eux-mêmes des expressions (expressions simples, combinaisons, formes spéciales); la valeur du premier élément doit être une fonction, les valeurs des suivants étant alors des arguments pour cette fonction.[24]
- Une *forme spéciale* est une liste dont le premier élément est un *mot-clef* spécifique; le nombre et la nature des autres éléments varient selon le type de la forme spéciale.

[22] Cette séquence indéfiniment répétée s'appelle la *boucle read-eval-print*.

[23] Cette terminologie n'est pas standardisée.

[24] Cette terminologie n'est pas standardisée. Nous souhaitons souligner la différence entre, d'une part, les éléments d'une combinaison (foncteur et opérandes) et les valeurs de ces éléments (fonction et arguments). En pratique, l'usage est moins strict, et on parlera parfois de fonctions et d'arguments en place de foncteurs et d'opérandes.

Syntaxiquement, les expressions simples sont des (représentations de) nombres ou des identificateurs alphanumériques (les mots-clefs sont interdits, de même que certains caractères, les parenthèses notamment). Les *formes*, c'est-à-dire les combinaisons et les formes spéciales, sont des listes.[25]

Avant d'introduire des définitions plus précises concernant notamment quelques formes spéciales importantes, nous donnons quelques exemples d'expressions simples et de combinaisons. Dans les exemples suivants, le symbole `==>` sépare l'expression à évaluer, introduite par l'utilisateur, de la valeur correspondante, produite et affichée par le système. Ce symbole, conventionnel, varie d'un système à l'autre; il inclut souvent un passage à la ligne.

```
3  ==>  3                  +  ==>  # procedure
8/5  ==>  8/5              pi  ==>  3.14159265
-3.14  ==>  -3.14          x  ==>  Error - unbound variable x
```

Les six évaluations ci-dessus concernent des expressions simples. Les constantes (ici, des nombres) s'évaluent en elles-mêmes. Un identificateur est vu par l'interprète comme une variable, à laquelle une valeur est liée. L'interprète renvoie cette valeur, si elle existe; sinon, un message signale "unbound variable" (variable non liée à une valeur). Le symbole `+` est "pré-lié" (lié par le système) à la fonction d'addition; les fonctions ne sont pas affichables explicitement, mais l'interprète signale que la valeur de "`+`" est une fonction. Enfin, on a supposé ici que la variable `pi` avait été liée précédemment à la valeur numérique 3.14159265, tandis que la variable `x` n'est pas liée. Nous considérons maintenant des combinaisons.

```
(+ 2 5)  ==>  7            (/ 2 0)  ==>  Error - 0 within proc /
(+ (* 3 -2) 2) ==> -4      (/ 8 3.0)  ==>  2.66666666
```

Les combinaisons évaluées ci-dessus sont des *formes arithmétiques*. Pour évaluer l'expression `(+ 2 5)`, on évalue d'abord les trois sous-expressions `+`, `2` et `5`, puis on applique la première valeur (la fonction d'addition) aux deux suivantes (les valeurs `2` et `5`), ce qui donne la valeur de l'expression, soit `7`.
Remarque. Si on essaie d'évaluer les listes `(2 3)` et `(x 3)`, on obtient

```
(2 3)  ==>  Error - object 2 not applicable
(x 3)  ==>  Error - variable x unbound
```

Dans les deux cas, le système détecte que, si expression il y a, elle ne peut être simple, puisqu'il y a une parenthèse initiale annonçant une liste. Le premier élément de cette liste n'étant pas un mot-clef, le système suppose que la liste est une combinaison. Dans le premier cas, le système tente d'appliquer `2` à l'argument `3`, ce qui est impossible, `2` n'étant

[25] Le mot "forme" est parfois employé comme synonyme de "expression (évaluable)".

pas une fonction. Dans le second cas, le système reconnaît que le symbole `x` n'a pas de valeur, n'est pas lié à une valeur.[26]

Remarque. La notation SCHEME est plus homogène que les notations mathématiques usuelles, où les foncteurs s'emploient de manière préfixée (comme dans "$\sin x$"), infixée (comme dans "$x+y$") ou postfixée (comme dans "$n!$"), ou encore font l'objet de conventions d'écriture spéciales (comme dans "e^x" et "$\int_{x_0}^{x} f(u)\mathrm{d}u$").

Remarque. La structure arborescente des expressions se prête à une définition dans le formalisme BNF.[27] En se restreignant aux expressions arithmétiques, on a

⟨A-expression⟩ ::= ⟨nombre⟩ | ⟨variable⟩ | ⟨A-combinaison⟩

⟨A-combinaison⟩ ::= `(` ⟨A-op⟩ [⟨A-expression⟩]* `)`

⟨A-op⟩ ::= `+` | `-` | `*` | `/`

On voit qu'une combinaison arithmétique, ou forme arithmétique, est un assemblage composé (dans l'ordre), d'une parenthèse ouvrante, d'un opérateur arithmétique, d'un certain nombre (0, 1 ou plus) d'expressions arithmétiques et d'une parenthèse fermante.

2.3 La forme spéciale `define`

La forme spéciale de mot-clef `define` est utilisée pour établir une liaison entre une variable et une valeur. L'évaluation de la forme spéciale `(define symb` e`)` ne produit pas de *valeur* mais a pour *effet* la liaison de la valeur de l'expression e à la variable `symb`. Dans la suite, nous supposerons que le système affiche "`...`" en réponse à l'évaluation d'une forme spéciale de mot-clef `define`. La session suivante illustre le rôle d'une telle forme :[28]

```
pi  ==>  Unbound variable
(define pi 3.1415926535)  ==>  ...
pi  ==>  3.1415926535
(define inv_pi (/ 1 pi))  ==>  ...
inv_pi  ==>  0.31830988619288864
```

Remarque. On pourrait créer une nouvelle liaison pour `pi`, ce qui rendrait la précédente inaccessible mais ne modifierait en rien la liaison relative à `inv_pi` :[29]

[26] Une telle liaison pourrait être le fait du système lui-même, comme pour le symbole `+`, ou de l'utilisateur (cf. § 2.3).

[27] Pour "Backus Normal Form"; le lecteur non familier avec cette notation peut passer immédiatement au paragraphe suivant.

[28] L'expression `(define symb` e`)` est une forme spéciale dont `define` est le mot-clef. Néanmoins, on parlera, par commodité, de "la forme spéciale `define`", plutôt que de "la forme spéciale de mot-clef `define`". De même, on dit "forme `+`" pour une forme arithmétique dont l'opérateur est l'addition.

[29] On verra plus loin un moyen d'assurer la propagation automatique sur `inv_a` d'une redéfinition de la constante `a`.

```
(define pi 4.0)  ==>  ...
pi  ==>  4.0
inv_pi  ==>  0.31830988619288864
```

2.4 Les symboles et leur double statut

Les textes que l'on peut introduire au clavier et soumettre au système SCHEME sont composés de "mots" (au sens large), séparés les uns des autres par des caractères spéciaux tels l'espacement, le saut à la ligne, les parenthèses, etc.[30] Certains de ces "mots" ont un rôle particulier, notamment les mots-clefs, tels `define` et `if`, les constantes numériques (`2`, `-4/3`, `3.5-2.3i`, `6.02e23`, etc.), les constantes booléennes `#t` et `#f`, ... La plupart des autres "mots", tels `indigo`, `prks9tc`, `iso_9000` et `127-bis`, sont des *symboles*.[31] Les symboles peuvent avoir deux rôles bien distincts. Tout d'abord, ils peuvent être des constantes lexicales (par opposition à d'autres types de constantes comme les nombres et les valeurs booléennes). Ces constantes lexicales n'ont pour le système aucune signification particulière, mais elles ont généralement une signification pour l'utilisateur.[32] Par défaut cependant, le système n'attribue pas à un symbole le statut de constante lexicale, mais celui de variable. Comme nous l'avons vu au paragraphe précédent, la signification d'une variable, ou plus exactement sa valeur, est l'objet qui lui est lié, s'il existe.[33] Pour que le système attribue à un symbole le statut de constante lexicale, on utilise le caractère spécial `'` (lire "quote") : la valeur d'une expression constituée de ce caractère suivi immédiatement d'un symbole est ce symbole. Ceci est illustré par la session suivante :

```
pi  ==>  3.1415926535                    'pi  ==>  pi
pi.+2  ==>  Unbound variable pi.+2       'pi.+2  ==>  pi.+2
indigo  ==>  Unbound variable indigo     'indigo  ==> indigo
```

Remarque. Le langage SCHEME a une syntaxe simple et systématique. Pour ne pas créer une quatrième catégorie syntaxique (en plus des expressions simples, des combinaisons et des formes spéciales), une expression telle que `'pi` a le statut de forme spéciale; c'est en fait une abréviation de la forme spéciale `(quote pi)`. Nous reviendrons plus loin sur la forme spéciale de mot-clef `quote`.

[30] Nous ne donnerons pas ici de manière exhaustive les règles compliquées de l'analyse syntaxique effectuée par le système.

[31] Certains "mots" sont exclus pour des raisons techniques, notamment ceux qui comportent un caractère spécial comme un séparateur. On n'essaiera pas ici d'être précis et exhaustif; notons cependant que l'usage des caractères `( ) [ ] { } ; , " ' ` # -+` dans les symboles est soumis à des restrictions ou totalement interdit. Par exemple, aucun symbole ne peut contenir de parenthèse; d'autre part, `2/b` et `3#a` sont des symboles mais `2/3` et `#3` n'en sont pas.

[32] Nous avons déjà mentionné un exemple : il est plus naturel de représenter les données "couleurs de l'arc-en-ciel" par les symboles "violet", "indigo", "bleu", "vert", "jaune", "orange" et "rouge", plutôt que par les nombres 0, 1, 2, 3, 4, 5 et 6.

[33] Puisqu'en tant que constantes lexicales les symboles n'ont *a priori* aucune signification, il est naturel de les utiliser comme variables, c'est-à-dire de leur attribuer une signification, ou plutôt une valeur, au gré du système ou de l'utilisateur.

Remarque. Le problème de la "citation" n'est pas inhérent à SCHEME, ni même aux langages de programmation en général, comme le montre le petit dialogue suivant :

Q. Dites-moi votre nom !
R. Paul Dupont.
Q. Dites-moi "votre nom" !
R. Votre nom.

2.5 Les listes

Une *liste* est une suite ordonnée et finie d'objets; le nombre d'objets est la *longueur* et les objets eux-mêmes sont les *éléments* de la liste. L'ensemble des expressions composées est inclus dans l'ensemble des listes, qui est lui-même inclus dans l'ensemble des valeurs. Syntaxiquement, une liste est représentée en SCHEME par la suite de ses éléments[34] séparés par des blancs, entourée d'une paire de parenthèses. La seule liste de longueur 0 est la liste vide `()`. Comme on l'a vu au chapitre précédent, le premier élément d'une liste non vide est sa *tête*; les autres éléments forment le *reste* (qui est aussi une liste, éventuellement vide). La tête et le reste sont les deux *composants directs* d'une liste non vide.[35] Voici trois exemples de listes qui sont aussi des expressions évaluables :

```
(- 3) , (+ 3 5) , (define pi 3.14) .
```

Par contre, les listes

```
(0 1 2 3) , (5 (+ (d 4)) ()) , ((1 a b) (2 c d) (3 e f))
```

ne sont pas des expressions évaluables. Enfin, la liste

```
(x y z)
```

n'est une expression évaluable que si `x` est une variable liée à une fonction admettant comme arguments les valeurs de `y` et `z`.

Nous avons vu que, par défaut, l'interprète attribue à un symbole le statut de variable; la forme spéciale de mot-clef `quote` permet de modifier la règle. De la même manière, lorsque l'on évalue, une liste non vide a par défaut le statut d'expression :

```
(2 3)  ==>  Error - bad procedure 2
(x 1 2)  ==>  Unbound variable x
(+ 2 3)  ==> 5
```

On déroge à la règle en utilisant `quote`, ce qu'illustre la session suivante :

[34] ... des représentations de ses éléments ...

[35] Rappelons que l'on ne doit pas confondre les composants directs et les éléments d'une liste. Les deux composants directs de la liste `(a b c)` sont `a` et `(b c)`, tandis que ses trois éléments sont `a`, `b` et `c`. Les composants indirects de `(a b c)` sont `b`, `(c)`, `c` et `()`.

```
'(2 3)  ==>  (2 3)
'(x 1 2)  ==>  (x 1 2)
(quote (2 3))  ==>  (2 3)
(quote (x 1 2))  ==>  (x 1 2)
```

On est souvent amené à considérer des listes dont les éléments sont soumis à certaines restrictions. En particulier, étant donné un ensemble non vide E d'objets qui ne sont pas des listes,[36] on définit formellement le domaine structuré $\mathcal{L}^E$ des listes (ou des E-listes) comme indiqué au paragraphe 1.3.4. L'unique élément de base est la liste vide, notée `()`. Le constructeur est noté `cons` et les deux accesseurs sont notés, pour des raisons historiques, `car` et `cdr`. Rappelons encore que la liste vide n'admet ni composant ni élément; les éléments d'une liste non vide sont des composants de cette liste (direct pour le premier, indirects pour les suivants), mais certains composants d'une liste non vide n'en sont pas des éléments.[37] Il est cependant naturel et commode de construire une liste à partir de ses éléments plutôt qu'à partir de ses deux composants directs; un constructeur auxiliaire nommé `list` concrétise cette possibilité; il prend un nombre quelconque d'arguments, qui sont les éléments de la liste-résultat.[38]

Soient $\alpha, \alpha_1, \ldots, \alpha_n$ des expressions quelconques et β une expression dont la valeur est une liste. En notant $[[e]]$ la valeur d'une expression e, on a les règles suivantes :

- $[[$`(cons` α β`)`$]]$ est une liste dont la tête est $[[\alpha]]$ et le reste est $[[\beta]]$; on a par exemple $[[$`(cons 0 (cons 1 '()))`$]] =$ `(0 1)`; la tête est `0` et le reste est la liste `(1)`. Toute liste non vide est valeur d'une forme `cons`.

- $[[$`(list` α_1 `...` α_n`)`$]]$ est la liste dont les éléments sont (dans l'ordre) $[[\alpha_1]], \ldots, [[\alpha_n]]$. Toute liste est valeur d'une forme `list`.

Les fonctions `cons`,[39] à deux arguments, et `list`, à nombre quelconque d'arguments, sont injectives; deux listes non vides sont égales si elles ont même tête et même reste; deux listes sont égales si leurs éléments sont deux à deux égaux. Notons que l'expression

`(list` α_1 α_2 `...`α_n`)`

équivaut à l'expression

`(cons` α_1 `(cons` α_2 `... (cons` α_n `'()) ... ))`;

[36] Plus précisément, qui ne sont pas la liste vide ou le résultat de l'application du constructeur à des arguments.

[37] Rappelons aussi, avec les notations de SCHEME, l'exemple qui terminait le paragraphe 1.3.3. La liste `(x y z)` admet les trois éléments `x`, `y` et `z` et les deux composants direct `x` et (`yz`); elle admet en outre les composants indirects `y`, (`z`), `z` et (). La liste (`x`) admet le seul élément `x`; sa tête est `x` et son reste est (); il n'y a pas de composant indirect.

[38] Pour être plus précis, `cons`, `car`, `cdr` et `list` sont des variables pré-liées par le système aux fonctions primitives, constructeurs et accesseurs du type "listes".

[39] Par abus de langage, on dit "la fonction `f`" plutôt que "la fonction liée à la variable `f`", exactement comme "Paul Durand" se dit pour "la personne dont le nom est Paul Durand".

la valeur commune est une liste dont les n éléments sont les valeurs des expressions $\alpha_1, \ldots, \alpha_n$.

La *notation pointée* est souvent utilisée pour mettre en évidence la décomposition d'une liste en sa tête et son reste. Si u est une liste non vide dont la tête est représentée par x et dont le reste est représenté par ℓ, alors u admet la représentation `(`x `.` ℓ`)`.

Les huit écritures

```
(0 1 2),      (0 1 . (2)),   (0 . (1 . (2))),   (0 1 . (2 . ())),
(0 . (1 2)),  (0 1 2 . ()),  (0 . (1 2 . ())),  (0 . (1 . (2 . ())))
```

représentent toutes la même liste de longueur 3, dont les éléments sont `0`, `1` et `2`.

Remarque. Nous avons vu au paragraphe 1.3.4 que les E-listes sont des cas particuliers de $(E \cup \{[\ ]\})$-arbres binaires. La notation pointée met ce fait en évidence, comme on le voit notamment en se reportant à la figure 2 : l'arbre binaire représenté sur cette figure correspond à la liste `((w x y) z)`, dont la notation pointée est `((w . (x . (y . ()))) . (z . ()))`. Naturellement, la notation pointée sert à représenter non seulement les listes mais aussi les arbres binaires quelconques; l'égalité

$$[[\texttt{(cons}\ \alpha\ \beta\texttt{)}]] \ = \ \texttt{(}[[\alpha]]\ \texttt{.}\ [[\beta]]\texttt{)}$$

restera vraie même si $[[\beta]]$ n'est pas une liste; les deux membres de l'égalités sont des *paires pointées*. On utilise fréquemment en SCHEME les *expressions symboliques*, c'est-à-dire les $(E \cup \{[\ ]\})$-arbres binaires, où E est l'ensemble des symboles. Le chapitre 8 leur est consacré.

Remarque. L'usage de `quote` s'étend à la notation pointée; on a par exemple

```
(list 2 3)             ==>  (2 3)
(list . (2 3))         ==>  (2 3)
(cons 2 (cons 3 '()))  ==>  (2 3)
(cons 2 (cons '(3)))   ==>  (2 3)
'(2 3)                 ==>  (2 3)
'(2 . (3))             ==>  (2 3)
'(2 . (3 . ()))        ==>  (2 3)
```

L'interprète utilise l'écriture la plus concise pour l'affichage.

Dans le domaine des listes comme dans tout domaine structuré, les accesseurs permettent de construire les fonctions inverses des constructeurs et réciproquement. Si [[`l`]] est une liste non vide, sa tête est l'objet [[`(car l)`]]; son reste est la liste [[`(cdr l)`]]. De même, [[`x`]] et [[`l`]] sont respectivement la tête et le reste de la liste [[`(cons x l)`]]. On a donc les égalités suivantes :

- [[`l`]] = [[`(cons (car l) (cdr l))`]] , si [[`l`]] est une liste non vide;
- [[`(car (cons a l))`]] = [[`a`]] et [[`(cdr (cons a l))`]] = [[`l`]] ,
  si [[`a`]] est un objet quelconque et si [[`l`]] est une liste.

Ces égalités peuvent servir d'axiomes dans un système formel de raisonnement sur les structures de listes.

Voici quelques exemples, illustrant l'usage des constructeurs `list` et `cons` et des accesseurs `car` et `cdr` :

```
(define pi 3.1415926535)  ==>  ...
(define l (list 3 '5 '(6 7) 'a3 'define pi))  ==>  ...
l  ==>  (3 5 (6 7) a3 define 3.1415926535)
(cons (car l) (cdr l))  ==>
        (3 5 (6 7) a3 define 3.1415926535)
(car l)  ==>  3
(cdr l)  ==>  (5 (6 7) a3 define 3.1415926535)
(cdr (car (cdr (cdr l))))  ==>  (7)
(cdaddr l)  ==>  (7)
(car '())  ==>  Error
(cons '(a b) '(6 7))  ==>  ((a b) 6 7)
(list '(a b) '(6 7))  ==>  ((a b) (6 7))
```

On notera les abréviations admises pour les enchaînements de `car` et `cdr`; en particulier, les fonctions `cadr`, `caddr` et `cadddr`, renvoyant respectivement les deuxième, troisième et quatrième éléments d'une liste, sont d'emploi fréquent.

2.6 Booléens, prédicats, forme spéciale `if`

Les *booléens* sont les constantes logiques : `#t` pour *true* (vrai) et `#f` pour *false* (faux). Comme toutes les constantes, elles s'évaluent en elles-mêmes.[40] Une expression est *vraie* si sa valeur est `#t`, et *fausse* si sa valeur est `#f`.

Les *prédicats* sont des fonctions prenant leurs valeurs dans les booléens. Beaucoup de prédicats numériques[41] (`<`, `=`, `zero?`, ...) ou non numériques (`null?`, `equal?`, ...) sont prédéfinis. Les *reconnaisseurs* sont des prédicats à un argument, associés aux différentes catégories d'objets; l'argument est un objet quelconque, la valeur retournée est `#t` si cet objet appartient à la catégorie en question et `#f` sinon. La table de la figure 3 illustre les reconnaisseurs prédéfinis les plus importants.

Des prédicats d'égalité existent pour différents types d'objets. On a en particulier "`=`" pour les nombres, "`eq?`" pour les symboles, "`eqv?`" pour les objets atomiques (nombres, symboles, booléens) et "`equal?`" pour les listes. Voilà quelques exemples :

```
(= (/ 3 2) 3/2 1.5 150e-2 1.5+0i)  ==>  #t
(eq? (list) '())  ==>  #t
(equal? '(a b) (cons 'a (cons 'b '())))  ==>  #t
```

[40] Les expressions `#t` et `'#t` ont toutes deux pour valeur `#t`.

[41] dont les arguments sont des nombres.

Reconnaisseur	*Objet(s) reconnu(s)*	*Expression vraie*	*Expression fausse*
`number?`	`nombres`	`(number? 4.5+.9e-3i)`	`(number? '(1))`
`list?`	`listes`	`(list? '(a . ()))`	`(list? '(() . b))`
`null?`	`liste vide`	`(null? (list))`	`(null? '(()))`
`procedure?`	`fonctions`	`(procedure? +)`	`(procedure? '+)`
`symbol?`	`symboles`	`(symbol? '+)`	`(symbol? +)`
`boolean?`	`booléens`	`(boolean? '#f)`	`(boolean? 'faux)`

Figure 3: Quelques reconnaisseurs

Remarque. Les prédicats d'égalité sont binaires, sauf "`=`" qui admet un nombre quelconque d'arguments.

Soient e_0, e_1 et e_2 trois expressions. La forme spéciale `(if` e_0 e_1 e_2`)` a pour valeur $[[e_2]]$ si $[[e_0]] =$ `#f` et $[[e_1]]$ sinon. La forme spéciale `if` permet donc l'évaluation conditionnelle, qui est un mécanisme essentiel en programmation. Voici quelques exemples d'emploi de ce mécanisme :

```
(if #t 1 2) ==> 1
(if #f 1 2) ==> 2
(if 'any_symbol 1 2) ==> 1
(if 5 1 2) ==> 1
(if (/ 5 0) 1 2) ==> Error - divide by zero
(if #t 1 (/ 2 0)) ==> 1
```

Remarque. La condition $[[e_0]]$ d'une forme `if` est considérée comme vraie dès qu'elle diffère de `#f`. Elle peut donc être d'un type non booléen (nombre ou liste, par exemple). Cette possibilité est parfois utilisée dans les programmes.[42]

2.7 La forme spéciale `lambda`

Le langage SCHEME permet l'utilisation de la notation lambda (cf. § 1.2.5). Dans cette notation, la fonction $x \mapsto x * x$ qui à tout nombre associe son carré s'écrit $\lambda x\ .\ x * x$. La variable x n'a qu'un rôle de marque-place et peut être renommée; $\lambda x\ .\ x * x$ et $\lambda y\ .\ y * y$ définissent naturellement la même fonction.

2.7.1 Définition

En SCHEME, le mot-clef `lambda` caractérise les formes spéciales permettant de définir les fonctions. La valeur de l'expression

```
(lambda (x) (* x x))
```

[42] Certains systèmes assimilent "`#f`" et "`()`" (liste vide); dans la suite, nous refusons cette assimilation mais, pour éviter que la valeur de la forme `(if` e_0 e_1 e_2`)` puisse varier selon le système utilisé, une règle de bonne pratique est de s'interdire le cas où $[[e_0]]$ est la liste vide.

est *fonctionnelle*; il s'agit de la fonction $\lambda x\ .\ x * x$. Les fonctions définies par l'utilisateur s'utilisent de la même manière que les fonctions du système; on a par exemple

```
((lambda (x) (* x x)) (+ 2 3))  ==>  25
((lambda (x y z) (+ (* x y) z)) 4 5 6)  ==>  26
```

Naturellement, il est souvent indiqué de donner un nom à une fonction :[43]

```
square  ==>  Error - unbound variable
(define square (lambda (x) (* x x)))  ==>  ...
square  ==>  # procedure
(square (+ 2 3))  ==>  25
```

Syntaxiquement, une forme spéciale `lambda` est une liste à trois éléments : le mot-clef `lambda`, la *liste des paramètres* et le *corps* de la forme `lambda`. Ce type de forme est omniprésent en SCHEME. On écrit souvent "λ-forme" (lire "lambda forme") au lieu de "forme `lambda`". La liste des paramètres est une liste de variables; le corps est une expression quelconque. Il faut bien observer le rôle particulier des paramètres, et ne pas confondre

1. la valeur d'une λ-forme;
2. la valeur d'une combinaison dont le premier élément est cette λ-forme.

La première valeur est une fonction; la seconde valeur résulte de l'*application* de cette fonction à des arguments qui sont les valeurs des éléments restants de la combinaison. Ceci est illustré dans les sessions suivantes.

```
pi  ==>  3.1415926535
y  ==>  Error - unbound variable
```

Au départ, la variable `pi` est liée, la variable `y` ne l'est pas.

```
(lambda (y) (* y y))  ==>  # procedure
(lambda (pi) (* pi pi))  ==>  # procedure
```

Les deux λ-formes sont équivalentes; le nom du paramètre est indifférent. Les valeurs de ces λ-formes étant — comme toujours — fonctionnelles, elles ne sont pas affichables; il s'agit, dans les deux cas, de la fonction carré, ce qui est illustré par les deux évaluations suivantes.

```
((lambda (y) (* y y)) 12)  ==>  144
((lambda (pi) (* pi pi)) 12)  ==>  144
```

Le processus d'évaluation des combinaisons de ce type provoque une liaison entre le paramètre (ici `y` ou `pi`) et l'argument (ici `12`), mais cette liaison est active pendant le processus d'application seulement; nous reviendrons sur ce processus plus loin.

[43] Rappelons que "donner le nom `var` à un objet" signifie créer une liaison entre la variable `var` et l'objet. Tant que cette liaison est en vigueur, la valeur de la variable `var` est l'objet. Nous (re)voyons ici que cet objet, cette valeur, peut être non seulement un nombre ou une liste, mais aussi une fonction. Le langage SCHEME traite de manière homogène les différents types d'objets. Nous reviendrons plus loin sur ce point important.

2.7.2 Le modèle de substitution

Le concept de λ-forme est central en programmation fonctionnelle; ce concept et celui de récursivité *sont* la programmation fonctionnelle. En fait, l'idée sous-jacente semble élémentaire et résumable en une égalité :

$$(\lambda x, y.M)\,(e_x, e_y) \;=\; M[(x,y)/(e_x,e_y)]\,.$$

Si M est une expression, $M[(x,y)/(e_x,e_y)]$ désigne l'expression obtenue en *substituant* e_x et e_y à x et y dans M, c'est-à-dire en remplaçant toutes les occurrences de x et y par e_x et e_y, respectivement. L'égalité ci-dessus signifie donc que pour appliquer la fonction $(\lambda x, y.M)$ aux arguments e_x, e_y, il suffirait d'évaluer le corps M dans lequel les arguments e_x et e_y sont substitués aux paramètres x et y. La substitution des arguments aux paramètres dans le corps d'une λ-forme s'appelle une *réduction*.
Remarque. Le modèle de substitution brièvement introduit ici est simple, mais ne traduit pas parfaitement la sémantique du système SCHEME. Une variante plus fidèle consiste à substituer au paramètres x et y les *valeurs* des opérandes e_x et e_y.

Considérons deux évaluations en SCHEME :

```
((lambda (x y) (* x (- y 1))) 12 24)  ==>  276
((lambda (x y) (* x (- y 1))) 3 (square 4))  ==>  45
```

Le premier exemple se comprend immédiatement; on a

$$[[\texttt{((lambda (x y) (* x (- y 1))) 12 24)}]] \;=\; [[\texttt{(* 12 (- 24 1))}]]$$

Le second exemple tient compte de la définition de `square` donnée plus haut. L'explication de cet exemple met en jeu des réductions, mais aussi une substitution de nature un peu différente : la fonction carré a été substituée à la variable `square` lors du processus d'évaluation. La technique de calcul que nous venons de présenter, appelée *modèle de substitution*, permet d'expliquer le résultat obtenu, mais une question se pose, qui concerne l'ordre dans lequel les calculs intermédiaires se font. La valeur `45`, résultat de l'évaluation ci-dessus, peut être vue comme la valeur de l'expression[44]

```
(* 3 (- (square 4) 1))
```

obtenue en réduisant la première λ-forme, ou comme la valeur de l'expression

```
((lambda (x y) (* x (- y 1))) 3 16)
```

obtenue en réduisant d'abord la seconde λ-forme, qui a permis la définition de `square`. On observe que le résultat final est le même quel que soit l'ordre choisi. On pourrait démontrer que, si on reste dans le monde des λ-formes, cette indépendance vis-à-vis de l'ordre des réduction est la règle.[45] Nous nous contentons de le vérifier ici sur un exemple plus significatif. L'expression

[44] Nous utilisons une police de caractères différente pour noter ces expressions, qui ne sont pas le résultat d'une évaluation faite par le système SCHEME.

[45] Plus précisément, si une séquence de réduction conduit à une valeur, toute séquence de réduction *qui se termine* conduit à la même valeur.

```
((lambda (w) (* 2 w))
 ((lambda (v) (- 9 ((lambda (u) (+ 3 u)) v))) 5))
```

comporte trois λ-formes; en effectuant une réduction, on arrive donc à l'une des trois expressions suivantes :

```
(* 2 ((lambda (v) (- 9 ((lambda (u) (+ 3 u)) v))) 5))
((lambda (w) (* 2 w)) (- 9 ((lambda (u) (+ 3 u)) 5)))
((lambda (w) (* 2 w)) ((lambda (v) (- 9 (+ 3 v))) 5))
```

En effectuant une réduction supplémentaire, on arrive à l'une des trois expressions suivantes :

```
(* 2 (- 9 ((lambda (u) (+ 3 u)) 5)))
(* 2 ((lambda (v) (- 9 (+ 3 v))) 5))
((lambda (w) (* 2 w)) (- 9 (+ 3 5)))
```

En effectuant une réduction supplémentaire ou, dans le troisième cas, une addition, on arrive à l'une des deux expressions

```
(* 2 (- 9 (+ 3 5)))
((lambda (w) (* 2 w)) (- 9 8))
```

Il est clair que la valeur finale sera `2`, quel que soit l'ordre des opérations. Par contre, l'efficacité du processus d'évaluation peut dépendre de l'ordre des opérations. L'exemple de la forme `(square (+ 3 4))` suffit à le montrer. Si on applique d'abord la fonction carré (valeur de la λ-forme liée à `square`), la valeur cherchée est celle de `(* (+ 3 4) (+ 3 4))`, tandis que si on applique d'abord la fonction d'addition, la valeur cherchée est celle de `(square 7)`. Dans le premier cas, la somme de `3` et `4` sera calculée deux fois, dans le second cas elle ne sera calculée qu'une fois.

Le système SCHEME impose que, dans l'évaluation d'une expression telle que `((lambda (x y)` M`)` e_x e_y`)`, les expressions e_x et e_y soient évaluées *avant* d'être substituées à `x` et `y` dans l'expression M.[46] Plus généralement, l'évaluation de `(f a b)` requiert d'abord la détermination des valeurs [[`f`]], [[`a`]] et [[`b`]] avant l'application de la première aux deux dernières. Par contre, l'ordre dans lequel les trois valeurs [[`f`]], [[`a`]] et [[`b`]] sont calculées est libre. Ce dernier point montre bien que la forme spéciale `if` ne pourrait pas être une fonction : l'ordre d'évaluation des expressions e_0, e_1 et e_2 dans `(if` e_0 e_1 e_2`)` n'est pas libre. Comme on l'a vu au paragraphe précédent, l'expression e_0 est évaluée d'abord; ensuite, en fonction de la valeur obtenue, l'expression e_1 *ou* l'expression e_2 est évaluée.

Une deuxième question se pose si on considère une définition plus compliquée telle que

[46] Notons que la stratégie inverse, dans laquelle les opérandes seraient substitués aux paramètres avant d'être évalués, est parfaitement concevable et présenterait certains avantages. D'ailleurs, même si le système SCHEME utilise la première stratégie, dite "ordre applicatif", des mécanismes existent qui permettent d'utiliser la seconde, dite "ordre normal", dans certains cas. Ces questions sortent du cadre de ce livre introductif.

```
(define f
  (lambda (a b c)          ; 1
    ((lambda (y)           ; 2
       (((lambda (y)       ; 3
           (lambda (x)     ; 4
             (* c x y)))
         a)
        ((lambda (x)       ; 5
           (+ a x y))
         b)))
     c)))
```

Cette définition comporte cinq λ-formes, numérotées de 1 à 5. On observe que `x` et `y` apparaissent dans deux listes de paramètres différentes. Cela n'empêche pas le modèle de substitution de fonctionner, comme le montre l'exemple suivant. Le système SCHEME donne

```
(f 2 3 4)  ==>  72
```

Le modèle de substitution donne le même résultat; nous le montrons ici en utilisant pour les réductions l'ordre *normal*, c'est-à-dire en réduisant les formes plus extérieures d'abord :

```
    (f 2 3 4)
==  ((lambda (y)
       (((lambda (y)
           (lambda (x)
             (* 4 x y)))
         2)
        ((lambda (x)
           (+ 2 x y))
         3)))
     4)
==  (((lambda (y)
         (lambda (x)
           (* 4 x y)))
       2)
      ((lambda (x)
         (+ 2 x 4))
       3))
==  ((lambda (x)
       (* 4 x 2))
     (+ 2 3 4))
==  (* 4 9 2)
==> 72
```

Plus généralement, si on pose

```
(define g
  (lambda (a b c)
    (* c (+ a b c) a)))
```

on peut montrer que [[(f a b c)]] = [[(g a b c)]], quels que soient les nombres a = [[a]], b = [[b]] et c = [[c]].

2.7.3 Exemples de calcul par le modèle de substitution

Le modèle de substitution permet un calcul aisé de la valeur de l'expression suivante :

```
((lambda (a) (* a a))
 ((lambda (a b) (a (* b b)))
  (lambda (a) (* a a))
  (* 2 2)))
```

Pour rendre l'expression plus lisible, la première étape consiste en un renommage des variables.

```
((lambda (x) (* x x))
 ((lambda (a b) (a (* b b))) (lambda (y) (* y y)) (* 2 2)))
((lambda (x) (* x x))
 ((lambda (a b) (a (* b b))) (lambda (y) (* y y)) 4))
((lambda (x) (* x x)) ((lambda (y) (* y y)) (* 4 4)))
((lambda (x) (* x x)) ((lambda (y) (* y y)) 16))
((lambda (x) (* x x)) (* 16 16))
((lambda (x) (* x x)) 256)
(* 256 256)
65536
```

L'exemple suivant se traite de même :

```
((lambda (x y z) (x y z))
 (lambda (x y) (* 2 x y))
 ((lambda (x y) (+ x y)) (+ 1 2) (+ 3 4))
 ((lambda (x) (+ 3 x)) 0))
```

```
((lambda (a b c) (a b c))
 (lambda (x y) (* 2 x y))
 ((lambda (u v) (+ u v)) 3 7)
 ((lambda (z) (+ 3 z)) 0))
((lambda (a b c) (a b c))
 (lambda (x y) (* 2 x y)) (+ 3 7) (+ 3 0))
((lambda (a b c) (a b c)) (lambda (x y) (* 2 x y)) 10 3)
((lambda (x y) (* 2 x y)) 10 3)
(* 2 10 3)
60
```

2.7.4 Exemples de définitions de procédures

Les formes spéciales `define`, `if` et `lambda`, ainsi que les fonctions arithmétiques primitives, permettent la programmation de nombreuses fonctions utiles. Nous avons déjà rencontré (sous d'autres noms) les fonctions

```
(define square_area
  (lambda (c) (* c c)))

(define circle_area
  (lambda (r) (* pi r r)))
```

La deuxième définition présuppose que la variable `pi` ait été liée à (une approximation de) la constante mathématique π, par exemple `3.14159`.
Remarque. Un programme SCHEME est un ensemble de formes `define`. Les objets définis sont le plus souvent des fonctions, mais il s'agit parfois d'objets non fonctionnels, comme des nombres "importants", auxquels on souhaite donner un nom. Les variables correspondantes sont dites "globales". Une convention courante consiste à utiliser pour les variables globales des symboles commençant et se terminant par "`*`"; on écrirait alors

```
(define *pi* 3.14159)
(define square_area (lambda (r) (* *pi* r r)))
```

Il est préférable d'introduire des variables globales seulement pour des objets auxquels on se réfère fréquemment, à divers endroits du programme. Une autre possibilité est de faire de π non pas une constante numérique, mais une fonction numérique sans argument; on écrirait alors

```
(define pi (lambda () 3.14159))
(define circle_area (lambda (r) (* 2 (pi) r)))
```

Notons l'usage des parenthèses : la valeur numérique [[`(pi)`]] est l'application de la fonction [[tt pi]].

L'usage des fonctions sans arguments a un avantage qu'illustre la session suivante :

```
(define pi 3.14)
(define inv_pi (/ 1 pi))
inv_pi  ==>  0.31847
(define pi 3.14159)
inv_pi  ==>  0.31847
```

Comme on l'a vu au paragraphe 2.3, la redéfinition de `pi` n'a pas eu d'effet sur `inv_pi`. Par contre, on a

```
(define pi (lambda () 3.14))
(define inv_pi (lambda () (/ 1 (pi))))
(inv_pi)  ==>  0.31847
(define pi (lambda () 3.14159))
(inv_pi)  ==>  0.31831
```

La redéfinition de `pi` a eu l'effet approprié sur `inv_pi`.

D'autres procédures simples permettent de calculer les surfaces de diverses formes géométriques. On a par exemple

```
(define lateral_cone_area
  (lambda (r a) (* *pi* r a)))
(define base_cone_area
  (lambda (r) (* *pi* r r)))
(define cone_area
  (lambda (r a)
    (+ (lateral_cone_area r a) (base_cone_area r))))
```

Remarque. On pouvait aussi écrire

```
(define base_cone_area
  (lambda (r) (circle_area r)))
```

ou, plus simplement,

```
(define base_cone_area circle_area)
```

On peut aussi, par exemple, écrire une fonction prenant comme argument une liste de trois nombres et renvoyant la somme de leurs racines carrées :

```
(define sum_3_sqrt
  (lambda (l)
    (+ (sqrt (car l)) (sqrt (cadr l)) (sqrt (caddr l)))))
```

On peut généraliser cette fonction en remplaçant la fonction prédéfinie `sqrt` par une fonction numérique unaire quelconque :

```
(define sum_3_f
  (lambda (f l) (+ (f (car l)) (f (cadr l)) (f (caddr l)))))
```

On a

```
(sum_3_f (lambda (x) (* 2 x)) '(1 2 3))  ==>  12
(sum_3_sqrt '(25 36 49))  ==>  18
(sum_3_f sqrt '(25 36 49))  ==>  18
```

Par contre, il semble difficile de généraliser `sum_3_f` en la fonction `+_map`, admettant comme second argument une liste de longueur quelconque. Pour cela et pour la plupart des problèmes courants, on devra recourir à la récursivité (cf. § 4).

2.7.5 La forme `lambda` généralisée

Nous avons déjà rencontré les fonctions prédéfinies `+`, `*` et `list`, qui admettent un nombre quelconque d'arguments. Il en est de même des prédicats arithmétiques `<`, `<=`, `=`, `>` et `>=`. Pour permettre à l'utilisateur de définir de telles fonctions, le langage SCHEME propose une version généralisée de la forme spéciale `lambda`; au lieu d'avoir une liste de paramètres de longueur fixée, telle `(x y z)`, on utilise un simple identificateur comme paramètre. La valeur de `((lambda v` E`)` $e_1 \ldots e_n$`)` s'obtient en évaluant E, la valeur du paramètre `v` étant la liste des valeurs $[[e_1]],\ldots,[[e_n]]$.

La fonction `list`, prédéfinie dans le langage, pourrait être définie comme suit :

```
(define list (lambda v v))
```

On a en effet

```
((lambda v v) 1 2 3 4)  ==>  (1 2 3 4)
```

On peut aussi créer des fonctions de projection, telles

```
(define proj_1 (lambda v (car v)))
(define proj_3 (lambda v (caddr v)))
```

On a alors

```
(proj_1 11 12 13 14)  ==>  11
(proj_3 11 12 13 14)  ==>  13
```

On veillera à distinguer `(lambda v` e`)` et `(lambda (v)` e`)`. En particulier, `proj_1` et `car` ne sont pas équivalents. Rappelons que, si `[[f]]` est une fonction à trois arguments alors `f` et `(lambda (x y z) (f x y z))` ont pour valeurs des fonctions égales.

Enfin, la notation pointée introduite au paragraphe 2.5 permet de définir des fonctions dont le nombre minimal d'arguments est fixé. Par exemple, la fonction `proj_3` ne peut être appliquée sans erreur que s'il y a au moins trois arguments; on peut souligner ce fait en la redéfinissant comme suit :

```
(define proj_3 (lambda (x1 x2 x3 . v) x3))
```

Certaines fonctions prédéfinies, telles la soustraction et la division, admettent au moins un argument; on notera leur comportement particulier :

```
(- 1)  ==>  -1
(- 1 2)  ==>  -1
(- 1 2 3)  ==>  -4
(/ 4)  ==>  1/4
(/ 4 3)  ==>  4/3
(/ 4 3 2)  ==>  2/3
```

2.7.6 Statut "première classe" des procédures

Les procédures, valeurs des λ-formes, héritent de toutes les propriétés essentielles des valeurs usuelles comme les nombres. En particulier, on peut définir une procédure admettant d'autres procédures comme arguments, et renvoyant une procédure comme résultat. De même, on peut lier une procédure à un symbole, en utilisant `define`. Ceci évoque les mathématiques (en analyse moderne, les domaines de fonctions ont le même "statut" que les domaines de nombres), mais contraste avec d'autres langages de programmation usuels, qui ne permettent pas, par exemple, de lier une procédure à une variable (sauf naturellement au moment où la procédure est définie). Nous savons déjà que cette limitation n'existe pas en SCHEME :

```
(define square (lambda (x) (* x x))) ==>  ...
square            ==>  # procedure
(square (+ 4 1))  ==>  25
(define power2 square) ==>  ...
power2            ==>  # procedure
(power2 (+ 4 1))  ==>  25
```

Remarquons aussi l'analogie de traitement de l'opérateur et des opérandes lors de l'évaluation d'une combinaison telle que `(square (+ 4 1))`; on commence par évaluer (peut-être en parallèle) l'opérateur et l'opérande, ce qui donne respectivement la valeur (fonction unaire) $v_0 = \lambda x.x^2$ et la valeur (numérique) $v_1 =$ `5`. On applique alors v_0 à v_1, ce qui revient à évaluer la forme `(* x x)` où `x` est lié à `5`.

Conceptuellement, la valeur-procédure v_0 associée à la variable `square` pourrait être la table (infinie) des couples (n, n^2). Une telle table n'est pas affichable, et SCHEME affichera simplement "`# procedure`". Concrètement, une valeur-procédure est soit une fonction primitive, soit le résultat de l'évaluation d'une λ-forme; ce résultat s'appelle une *fermeture* et comporte les informations nécessaires au système pour évaluer l'application de la valeur-procédure à des arguments.

Un aspect plus spectaculaire du principe consistant à traiter les valeurs fonctionnelles comme les valeurs numériques est la facilité avec laquelle, en SCHEME, on écrit des procédures dites "d'ordre supérieur", dont le domaine et le codomaine comportent eux-mêmes des procédures, éventuellement d'ordre supérieur. Un exemple simple est l'opérateur mathématique de composition fonctionnelle. Il prend comme arguments une fonction f de D_2 dans D_3 et une fonction g de D_1 dans D_2 et leur associe une troisième fonction $f \circ g : x \to f(g(x))$ de D_1 dans D_3.[47] Ceci est illustré dans la session ci-dessous.

```
(define compose
  (lambda (f g)
    (lambda (x) (f (g x)))))  ==>  ...
```

[47] Quand on évalue $(f \circ g)(x)$, c'est-à-dire $f(g(x))$, on applique d'abord g, puis f. Pour cette raison, $f \circ g$ se lit souvent "g rond f" ou "f après g".

```
(compose car cdr)  ==>  # procedure

((compose car cdr) '(1 2 3 4))  ==>  2

((compose (compose car cdr) cdr) '(1 2 3 4))  ==>  3

((compose (compose car cdr) (compose cdr cdr)) '(1 2 3 4)) ==> 4
```

Un exemple plus élaboré est l'opérateur de dérivation qui à toute fonction dérivable de $\mathbb{R}$ dans $\mathbb{R}$ associe une fonction de $\mathbb{R}$ dans $\mathbb{R}$. Pour éviter l'intervention de la notion de limite, nous fixons un incrément `dx`, vu comme argument supplémentaire de l'opérateur. On définit alors

```
(define deriv
  (lambda (f dx)
    (lambda (x)
      (/ (- (f (+ x dx)) (f x)) dx))))
```

La valeur fonctionnelle $v_1 = [[\texttt{(deriv f dx)}]]$ est une bonne approximation de la dérivée de la valeur fonctionnelle $v_0 = [[\texttt{f}]]$. Par exemple, si v_0 est la fonction carré et si $[[\texttt{dx}]] = 0.00001$, alors la fonction v_1 est

$$x \longrightarrow \frac{(x + 0.00001)^2 - x^2}{0.00001}$$

ou encore la fonction

$$x \longrightarrow 2x + 0.00001\,,$$

qui est une très bonne approximation de la fonction double, dérivée de la fonction carré. On a par exemple :

```
((deriv square 0.00001) 10)  ==>  20.00000999942131
```

Le modèle de substitution donne

```
((deriv square 0.00001) 10)

(((lambda (f dx)
    (lambda (x) (/ (- (f (+ x dx)) (f x)) dx))))
  square
  0.00001)
 10)

((lambda (x)
   (/ (- (square (+ x 0.00001)) (square x)) 0.00001))
 10)

(/ (- (square (+ 10 0.00001)) (square 10)) 0.00001)
```

```
(/ (- (square 10.00001) (square 10)) 0.00001)
...
(/ (- 100.0002000001 100) 0.00001)
20.00001
```

Remarque. On n'a pas utilisé ici strictement l'ordre normal.

Remarque. Les algorithmes de calcul arithmétique en nombres décimaux souffrent souvent d'erreurs d'arrondis; c'est pourquoi on a obtenu le résultat `20.00000999942131` au lieu de `20.00001`. L'erreur d'arrondi ne doit pas être confondue avec l'erreur systématique liée au choix de la valeur de `dx`; sans cette erreur systématique, la réponse attendue aurait été `20` et non `20.00001`.

3 Règles d'évaluation

Dans ce chapitre nous synthétisons les règles relatives à l'évaluation des expressions. Le modèle de substitution introduit au chapitre précédent ramène l'évaluation des expressions à l'évaluation des expressions simples, constantes et variables, et à l'application de fonctions prédéfinies à des arguments évalués. Ces mécanismes fondamentaux sont très simples : la valeur d'une constante est elle-même, la valeur d'une variable est l'objet qui lui est lié et le résultat de l'application d'une fonction prédéfinie à des arguments est obtenu en exécutant le code relatif à cette fonction primitive.

L'introduction de la forme spéciale essentielle `lambda` fait apparaître la notion de réduction, ce qui complique la situation. En effet, l'opération de substitution d'un argument [[`e`]] à un paramètre [[`x`]] doit pouvoir se faire même si la variable `x` est liée à une autre valeur v. On ne peut donc plus considérer que l'objet lié à une variable est unique. Le problème fondamental consistant à déterminer l'objet lié à une variable dans une situation donnée devient complexe si de nombreuses λ-formes, éventuellement imbriquées, sont présentes dans le programme que l'on exécute. Ce chapitre a aussi pour but d'introduire la notion d'environnement, utilisée pour gérer le problème du lien entre une variable et une valeur, et le modèle de calcul lié à cette notion.

3.1 Résumé des règles

Le chapitre précédent contenait une définition précise de la syntaxe des expressions, ainsi que quelques indications sur la manière de les évaluer. Nous revenons ici plus précisément sur le processus d'évaluation des expressions.

Rappelons d'abord que les règles d'évaluation des expressions peuvent être récursives : l'évaluation d'une expression peut requérir l'évaluation de sous-expressions. Nous avons vu aussi que le processus d'évaluation renvoie en général une valeur; il peut aussi avoir un effet (outre l'affichage de la valeur), comme la création d'une liaison dans le cas d'une forme `define`.

Le processus d'évaluation comporte une série de règles; la règle à appliquer pour une expression donnée dépend exclusivement du type de l'expression et donc, de sa syntaxe. Pour le processus d'évaluation, les nombres, les booléens et les variables sont des cas de base, tandis que les combinaisons sont des cas inductifs. Les formes spéciales sont des cas de base ou des cas inductifs, cela dépend du type de la forme, déterminé par le mot-clef.

Voilà d'abord les règles relatives aux cas de base.

- L'évaluation d'un nombre donne ce nombre.
- L'évaluation d'un booléen donne ce booléen.
- L'évaluation d'un symbole donne la valeur liée à ce symbole, si elle existe.
- L'évaluation d'une forme spéciale `quote` donne l'expression citée, non évaluée.

- L'évaluation d'une forme spéciale `lambda` donne la valeur fonctionnelle associée; cette valeur est une fermeture, objet à trois composants dont la nature sera précisée sous peu.

Voici une session donnant quelques exemples :

```
pi                      ==>     Error - unbound variable
(define pi 3.14159)     ==>     ...
'pi                     ==>     pi
pi                      ==>     3.14159
#f                      ==>     #f
twice                   ==>     Error - unbound variable
(define twice (lambda (x) (* 2 x)))  ==>  ...
twice                   ==>     # procedure ...
(twice 2)               ==>     4
(twice pi)              ==>     6.28318
car                     ==>     # procedure ...
(lambda (x) (+ x y))    ==>     # procedure ...
xxx                     ==>     Error - unbound variable
'xxx                    ==>     xxx
'(a b)                  ==>     (a b)
'(+ 3 5)                ==>     (+ 3 5)
(quote a)               ==>     a
```

Dans le cas inductif, l'évaluation d'une expression requiert l'évaluation préalable de sous-expressions. On a les règles suivantes :

- L'évaluation d'une combinaison $(e_0\ e_1\ \ldots\ e_n)$ s'effectue comme suit :

  1. Les e_i sont évalués (soient v_i les valeurs);
     v_0 doit être une fonction dont le domaine contient $(v_1, \ldots, v_n)$.
  2. La fonction v_0 est appliquée à $(v_1, \ldots, v_n)$.
  3. Le résultat de l'application est la valeur de la combinaison.

  Remarque. Si l'un des v_i n'existe pas ou si v_0 n'est pas applicable, l'évaluation s'arrête sans fournir de valeur.

- L'évaluation de la forme spéciale `(define symb` e`)` s'effectue comme suit :

  1. L'expression e est évaluée; soit v sa valeur.
  2. La valeur v est liée à la variable `symb`.
  3. Aucune valeur n'est produite.

- L'évaluation de la forme spéciale `(if` $e_0\ e_1\ e_2$`)` s'effectue comme suit :

  1. L'expression e_0 est évaluée; soit v_0 sa valeur.

2. Si v_0 est `#f`, e_2 est évalué, sinon e_1 est évalué;
   dans les deux cas, soit v la valeur produite.
3. La valeur de l'expression conditionnelle est v.

Remarque. Si v_0 ou v n'existe pas, l'évaluation s'arrête sans produire de valeur.

Remarque. Une valeur fonctionnelle n'est pas affichable explicitement; un message informatif (qui n'est pas un message d'erreur) remplace cet affichage.

Remarque. La forme spéciale `if` présente une particularité importante : une des sous-expressions n'est pas évaluée. En fait, e_0 est toujours évaluée mais, des deux sous-expressions e_1 et e_2, une seule sera évaluée : la seconde si $[[e_0]]$ est `#f`, la première sinon. Cette particularité est intéressante en pratique, parce que cela permet à l'expression non évaluée de n'avoir pas de valeur, sans que cela provoque une erreur. Nous verrons au chapitre suivant que ceci est mis à profit lors de la définition d'une fonction récursive. Un exemple d'une telle définition, la fonction `fib`, apparaissait d'ailleurs déjà au premier chapitre; le code comportait l'expression conditionnelle

```
(if  (< n 2)  n  (+ (fib (- n 1)) (fib (- n 2))))
```

Quand la sous-expression

```
(< n 2)
```

est vraie, la sous-expression

```
(+ (fib (- n 1)) (fib (- n 2)))
```

n'est pas évaluée.

Rappel. La valeur $[[e_0]]$ est la *condition*.[48] En général, la condition est booléenne, mais cela n'est pas obligatoire : Une condition qui n'est pas fausse est assimilée à une condition vraie. On a par exemple $[[\texttt{(if 1 2 3)}]] = \texttt{2}$. Il faut cependant que la condition existe, sinon la forme conditionnelle n'a pas de valeur. L'évaluation de l'expression `(if (/ 1 0) 2 3)` donnera lieu à une erreur.

3.2 Mode d'application et environnements

3.2.1 Introduction et exemple

La valeur de `(`e_0 e_1 `...` e_n`)` est le résultat de l'application de la fonction $[[e_0]]$ aux arguments $[[e_1]]$, ..., $[[e_n]]$. Si $[[e_0]]$ est une fonction primitive, le système exécute le code interne associé à cette primitive pour produire ce résultat. Si par contre $[[e_0]]$ est $[[\texttt{(lambda (x1 ... xn)}\ M\texttt{)}]]$, un processus spécial est exécuté; il consiste en l'évaluation du corps M de la λ-forme, étant admis que les variables `x1`, ..., `xn` sont liées aux valeurs $[[e_1]]$, ..., $[[e_n]]$, respectivement. La liaison relative aux paramètres est prise en compte

[48] En pratique, et par abus de langage, l'expression e_0 elle-même est parfois appelée "condition".

lors de cette évaluation seulement; dans le cadre de l'évaluation d'une autre expression, elle sera invisible. Avant de décrire ce processus de manière plus précise, nous donnons un petit exemple.

```
(define y 9)  ==>  ...
x  ==>  Error - unbound variable x
y  ==>  9
((lambda (x y) (+ 2 x y)) 3 7)  ==>  12
x  ==>  Error - unbound variable x
y  ==>  9
```

Dans cette session, les liaisons de `x` et `y` à `3` et `7` ne sont actives que lors de l'évaluation correspondant à la quatrième ligne de cette session, après quoi ces liaisons deviennent inaccessibles, comme le montrent les deux dernières lignes de la session. Une réévaluation ultérieure de la forme `((lambda (x y) (+ 2 x y)) 3 7)` créera des liaisons analogues qui deviendront à leur tour inaccessibles. Par contre, dans la session

```
x  ==>  Error - unbound variable x
y  ==>  9
(define foo
  ((lambda (x y) (lambda (u) (+ u x y))) 3 7))  ==>  ...
x  ==>  Error - unbound variable x
y  ==>  9
(foo 8) ==> 18
x  ==>  Error - unbound variable x
y  ==>  9
(foo 20)  ==>  30
(foo y)  ==>  19
```

les liaisons de `x` et `y` à `3` et `7`, créée lors de la définition de `foo`, sont actives chaque fois que la fonction [[`foo`]] est appliquée à un argument. En dehors de telles applications, ces liaisons sont inaccessibles. Observons que [[`foo`]] est la fonction $x \mapsto x + 10$; lors de l'évaluation de `(foo y)`, le système tient compte de deux liaisons relatives à `y`. Cet exemple montre que plusieurs liaisons relatives à une même variable peuvent coexister à un moment donné.

3.2.2 Notion d'environnement

En l'absence du mécanisme des λ-formes, le problème de la liaison entre variables et valeurs peut être géré par une simple table, ou *cadre*, qui est un ensemble de paires variable-valeur. Ce cadre global comporte les liaisons relatives aux variables prédéfinies et à celles qui ont fait l'objet d'un `define`.

Le processus d'application d'une fermeture à des arguments suscite la création d'un nouveau cadre, composé des paires paramètre-argument. Lors de l'évaluation du corps de

la procédure (compris dans la fermeture) la valeur d'une variable est recherchée d'abord dans le nouveau cadre; elle s'y trouvera si la variable est un paramètre.

D'une manière générale, toute évaluation de variable a lieu dans un *environnement*, c'est-à-dire un pointeur vers une structure dont les nœuds sont des cadres. Quand le processus d'application d'une fermeture commence, un nouvel environnement est créé à partir de l'environnement courant. Ce nouvel environnement comporte un pointeur vers le nouveau cadre, qui lui-même pointe vers l'ancien environnement.[49] Lorsque le processus d'application est terminé, l'ancien environnement est restauré. On conçoit que, lors de l'évaluation d'une expression comportant de nombreuses λ-formes, des environnements compliqués puissent être créés. L'environnement de départ, dans lequel l'utilisateur se trouve quand il entre en session, est l'*environnement global*, comportant seulement le cadre global. Ce cadre comporte les liaisons prédéfinies et celles crées par l'utilisateur au moyen d'un `define`.

3.2.3 Les fermetures et le processus d'application

On a mentionné au paragraphe 2.7.6 que la valeur d'une λ-forme s'appelle une *fermeture* et comporte les informations nécessaires au système pour évaluer toute forme impliquant l'application de la valeur-procédure à des arguments. On peut maintenant préciser que ces informations comportent trois composants : la liste des paramètres, le corps de la λ-forme, et l'*environnement de définition*, c'est-à-dire l'environnement dans lequel la λ-forme a été évaluée. C'est dans cet environnement que les valeurs correspondant aux variables non locales du corps de la λ-forme seront recherchées, chaque fois que la valeur-procédure sera appliquée à des arguments.

On peut maintenant décrire plus finement le processus d'application, qui est un mécanisme essentiel de l'évaluateur. Rappelons d'abord qu'une fonction (mathématique) est un ensemble de couples de valeurs; la première est un élément du domaine, la seconde l'image correspondante. *Appliquer* une fonction (à un nombre adéquat d'arguments de types appropriés) est produire l'image associée à cette suite d'arguments. En SCHEME, pour les fonctions prédéfinies (liées à `+`, `cons`, etc.) le processus d'application est l'exécution du code correspondant à la fonction. Le cas des fonctions définies par l'utilisateur, au moyen de la forme spéciale `lambda`, requiert une définition supplémentaire :

- Appliquer la fermeture v_0 = [[`(lambda (x1 ... xn)` M`)`]] à des arguments $v_1,\ldots,v_n$ consiste d'abord à créer un nouvel environnement, où les valeurs de `x1`,...,`xn` sont respectivement $v_1,\ldots,v_n$, puis à évaluer la forme M dans cet environnement, les valeurs des variables libres éventuellement présentes dans M étant recherchées dans l'environnement inclus dans la fermeture v_0).

Illustrons d'abord cette règle au moyen d'un exemple élémentaire :

```
(define add_3 (lambda (u) (+ 3 u)))  ==>  ...
(add_3 (* 2 4))   ==>  11
```

[49] On verra plus loin que si E' pointe vers E'', E' est l'*environnement de contrôle* de E''.

Ces expressions sont évaluées dans l'environnement global E_0, comportant le seul cadre global C_0. Dans ce cadre se trouvent les liaisons entre les variables prédéfinies (telles `+` et `*`) et leurs valeurs. La première évaluation concerne une forme spéciale `define`. La valeur de la λ-forme est la fermeture `[(u);(+ 3 u);`E_0`]` (E_0 étant l'environnement de définition de la fermeture); elle est liée à la variable `add_3`, dans le cadre global.

La seconde évaluation concerne une combinaison dont l'opérateur est `add_3` et l'opérande est `(* 2 4)`.

La première étape consiste en l'évaluation dans E_0 de l'opérateur et de l'opérande. L'évaluation de l'opérateur donne la valeur qui lui est liée dans E_0. L'évaluation de l'opérande consiste en l'évaluation des composants `*`, `2` et `4` (la valeur de `*` est trouvée dans l'environnement courant E_0) et en l'application de [[`*`]] à `2` et `4`, c'est-à-dire en l'exécution du code effectuant la multiplication de deux nombres; le résultat produit est `8`. La seconde étape consiste en l'application de la fermeture `[(u);(+ 3 u);`E_0`]` à la valeur `8`. Cette application débute par la création d'un nouvel environnement E_1, et donc d'un nouveau cadre C_1 (voir figure 4). Dans ce cadre est placée une liaison entre le paramètre `u` de la fermeture et l'argument `8`. Enfin, le corps `(+ u 3)` est évalué dans l'environnement E_1; la valeur de la constante `3` est `3`; la valeur de la variable locale `u` est trouvée dans le cadre local C_1 et la valeur de la variable non locale `+` est cherchée et trouvée dans l'*environnement d'accès*, qui est l'environnement E_0 compris dans la fermeture. L'application de [[`+`]] à `3` et `8` donne `11`, valeur qui est passée à l'*environnement de contrôle* de E_1, c'est-à-dire à l'environnement E_0 à partir duquel E_1 a été créé, et auquel on revient après que la valeur à produire dans E_1 l'a été. Cette valeur `11` est le résultat de l'évaluation de `((lambda (u) (+ 3 u)) (* 2 4))` dans E_0.

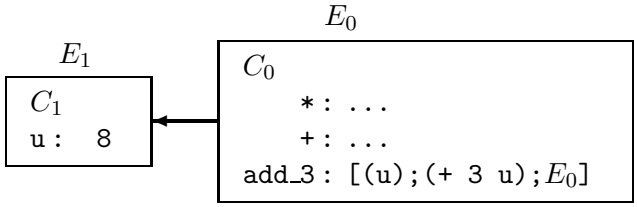


Figure 4: Environnements

L'exemple de l'expression

```
((lambda (w) (* 2 w))
 ((lambda (v) (- 9 ((lambda (u) (+ 3 u)) v))) x))
```

déjà évoquée lors de la présentation du modèle de substitution (§ 2.7.2), sera une deuxième illustration des principes et des notations utilisées dans le modèle des environnements. Nous supposons que cette expression est évaluée dans l'environnement global E_0, où la variable `x` est liée à la valeur `5` suite à l'évaluation de `(define x 5)`. L'expression à évaluer est une combinaison. La valeur de l'opérateur est une fermeture w_0 comportant l'environnement E_0. L'unique opérande est aussi une combinaison, dont l'opérateur a

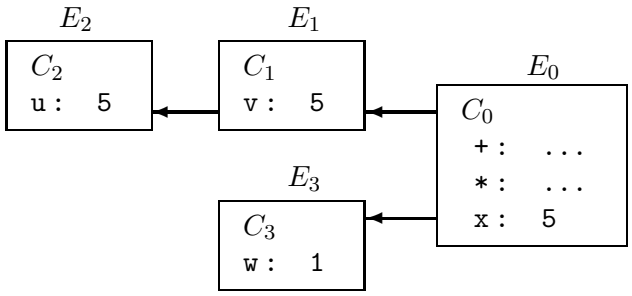


Figure 5: Modèle des environnements

pour valeur une fermeture v_0 comportant l'environnement E_0 et dont l'opérande `x` a pour valeur `5`. Un environnement E_1 est créé, où `v` a pour valeur `5` et dans lequel v_0 est appliqué à [[v]]; le résultat de cette application est la valeur dans E_1 du corps `(- 9 ((lambda (u) (+ 3 u)) v))` contenu dans la fermeture v_0. Pour obtenir la valeur de cette combinaison, il faut notamment évaluer son deuxième opérande, qui résultera de l'application d'une fermeture u_0 (comportant l'environnement E_1) à la valeur `5`, ce qui implique de créer un environnement E_2, où `u` a pour valeur `5` et dans lequel le corps `(+ 3 u)` est évalué, ce qui donne la valeur `8`. La valeur produite dans E_1 est donc `1` $(9 - 8)$. La valeur à produire dans E_0 résulte de l'application à `1` de la fermeture w_0. Pour obtenir cette valeur, un environnement E_3 est créé, où `w` a pour valeur `1` et dans lequel le corps `(* 2 w)` est évalué, ce qui donne `2`, valeur de l'expression de départ dans l'environnement E_0.

La figure 5 représente les divers environnements. Chaque environnement est un pointeur vers un cadre qui porte son nom (cette convention restera d'application dans tous les exemples ultérieurs). Du cadre E_0 partent deux flèches parce que les évaluations à faire dans l'environnement E_0 impliquaient l'application de deux fermetures à des arguments. On observe que les évaluations à faire dans un environnement E_i donné impliquent parfois des valeurs à chercher dans d'autres cadres que le cadre E_i. En fait, l'environnement E_2 comporte non seulement le cadre E_2 mais aussi les cadres E_1 et E_0; c'est dans E_0 qu'est obtenue la valeur liée à la variable `+`, nécessaire pour l'évaluation dans E_2 de la forme `(+ 3 u)`. Nous approfondissons ce point au paragraphe suivant.

3.3 Portée des variables

3.3.1 Variables globales, variables locales, variables libres

Supposons un environnement dont le premier cadre comporte une liaison entre la variable `x` et la valeur v. La valeur de `x` dans cet environnement est naturellement la valeur v. Si ce premier cadre ne comporte pas de liaison relative à la variable `x`, la valeur éventuelle de `x` doit se trouver ailleurs. Nous commençons l'étude de cette question par l'exemple simple :

```
;; Exemple 1
```

```
(define square (lambda (x) (* x x)))  ==>  ...
((lambda (x) (+ 9 (square (+ x 1)))) 3)  ==>  25
```

Le cadre global C_0 comporte des liaisons relatives aux variables prédéfinies `+` et `*`; la première ligne de la session a pour effet d'ajouter à ce cadre une liaison entre la variable `square` et sa valeur, la fermeture `[(x);(* x x);`E_0`]`. La seconde ligne de la session est l'évaluation d'une combinaison dont l'opérateur a pour valeur la fermeture `[(x);(+ 9 (square (+ x 1)));`E_0`]`; l'application de cette fermeture suscite la création d'un environnement E_1, dont le premier cadre C_1 lie `x` à `3`. Ensuite, le corps `(+ 9 (square (+ x 1))` est évalué dans cet environnement. La valeur de `x` est trouvée localement, c'est-à-dire dans le cadre C_1, tandis que les valeurs des variables `+` et `square` sont trouvées dans l'environnement de définition, enclos dans la fermeture, c'est-à-dire E_0. La suite du processus requiert l'évaluation de `(square (+ x 1))` dans l'environnement E_1; [[`square`]] étant une fermeture, il y a création d'un nouvel environnement E_2, dont le premier cadre C_2 lie `x` à la valeur de `(+ x 1)` dans E_1, c'est-à-dire `4`. L'évaluation de `(* x x)` dans E_2 produit la valeur `16`. Enfin, une application de la fonction primitive [[`+`]] donne le résultat final `25`. Les environnements concernés par cette session sont représentés à la figure 6. On notera que E_0 comporte le seul cadre C_0, E_1 comporte les cadres C_1 et C_0, et E_2 comporte les cadres C_2, C_1 et C_0.

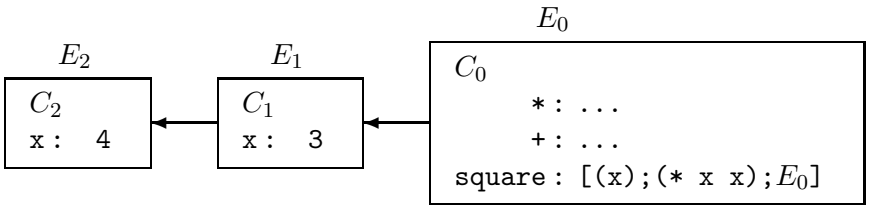


Figure 6: Environnements et variable globale

Dans cet exemple, deux sortes de variables apparaissent : les variables *locales*, dont la valeur dans un environnement donné est trouvée dans le cadre de même indice que cet environnement, et les variables *globales*, dont les valeurs sont trouvées dans l'environnement global.

Il existe aussi des variables *libres*, dont la valeur sera trouvée dans un cadre intermédiaire. Un exemple simple de variable libre apparaît dans la session suivante :

```
;; Exemple 2
((lambda (a) ((lambda (x) (+ a (* x 2))) 2)) 3)  ==>  7
```

Avec les notations de la figure 7, évaluer l'expression complète dans l'environnement global E_0 implique la création de E_1, où `a` est lié à `3`; l'évaluation de `((lambda (x) (+ a (* x 2))) 2)` dans l'environnement E_1 suscite à son tour la création de E_2, où `x` est lié à `2`. Il y a alors l'évaluation du corps `(+ a (* x 2))` dans E_2; dans cette expression (et dans cet environnement), la variable `a` est libre.

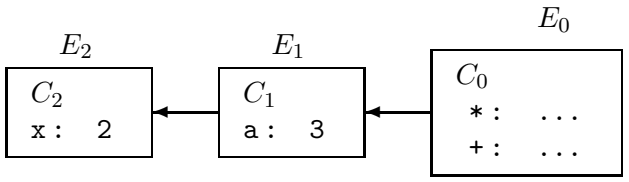


Figure 7: Environnements et variable libre

3.3.2 Conflits de noms

Dans l'exemple 1, la variable `x` est liée dans le cadre C_1 et dans le cadre C_2. Cela ne pose pas de problème : la liaison C_1 est utilisée pour déterminer la valeur de `x` dans l'environnement E_1, puisque, dans cet environnement, on n'a pas accès au cadre E_2; d'autre part, la liaison C_2 est utilisée pour déterminer la valeur de `x` dans l'environnement E_2 : une occurrence d'une variable définie localement est toujours considérée comme locale. Il n'y a donc jamais de conflit entre une variable locale et une variable non locale. Par contre, il peut y avoir conflit entre deux variables libres, ou entre une variable libre et une variable globale, comme le montre l'exemple suivant :

```
;; Exemple 3

(define pi 3.14)  ==>  ...
(define circ (lambda (r) (* 2 pi r)))  ==>  ...

((lambda (pa pe pi po pu)
   (+ (circ pa) (circ pe) (circ pi) (circ po) (circ pu)))
 1 2 4 8 16)  ==>  194.68
```

L'évaluation de l'expression `((lambda ...) 1 2 4 8 16)` dans l'environnement global E_0 implique la création d'un nouvel environnement E_1 (voir figure 8) et l'évaluation du corps de la λ-forme, la combinaison `(+ (circ pa) ... (circ pu))`, dans cet environnement. Les six éléments de cette combinaison (un opérateur et cinq opérandes) sont évalués dans le même environnement E_1. Considérons l'opérande `(circ pi)`; son évaluation requiert celle de `circ`, variable globale dont la valeur est la fermeture `[(r);(* 2 pi r);`E_0`]`, et celle de `pi`, variable locale liée à `4`. L'application de la fermeture [[`circ`]] à `4` requiert la création de l'environnement E_2 et l'évaluation de la forme `(* 2 pi r)` dans cet environnement. Il faut donc connaître la valeur de `pi`, que l'on peut trouver soit dans l'*environnement d'application* (de [[`circ`]] à `4`, donc E_1), soit dans l'*environnement de définition* (de cette même fonction, donc E_0).

Dans le langage SCHEME, c'est toujours l'environnement de définition qui est utilisé en pareil cas; on dit que SCHEME utilise la *règle de portée lexicale,* ou *statique des variables*, par opposition à la *règle de portée dynamique des variables*, adoptée dans certains autres langages. Dans l'exemple qui nous occupe, c'est donc la valeur [[`pi`]] = `3.14` qui sera

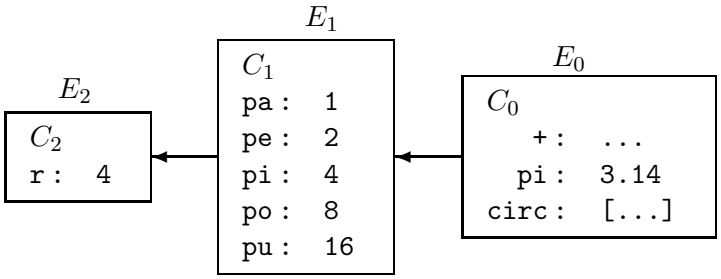


Figure 8: Environnements et conflit de nom

utilisée lors de l'évaluation de `(* 2 pi r)`, et non la valeur `4`. Concrètement, lors de la création de l'environnement E_2, le système note que les valeurs des variables libres sont à chercher non dans l'*environnement de contrôle* E_1 (à partir duquel E_2 est créé) mais dans l'environnement mentionné dans la fermeture [[`circ`]], qui est l'environnement de définition de [[`circ`]], c'est-à-dire E_0. Par rapport à E_2, E_0 est l'*environnement d'accès*.

Le fait, lors de l'application d'une fonction à ses arguments, de privilégier l'environnement de définition a une conséquence générale très importante : la fonction liée à la variable `circ` lors de l'évaluation (dans l'environnement global) de la forme `(define circ ...)` est fixée une fois pour toutes; il suffit de lire le *texte* du programme, c'est-à-dire le fichier contenant les définitions relatives à `pi` et à `circ` pour connaître cette fonction. Si la règle de portée dynamique était d'application, la fonction [[`circ`]] correspondant à ces deux définitions serait la même que précédemment ... sauf dans les cas ou, volontairement ou non, cette fonction serait appliquée dans un environnement où `pi` a une autre valeur. Notons aussi qu'avec la règle de portée statique, les deux expressions

```
((lambda (pa pe pi po pu)
   (+ (circ pa) (circ pe) (circ pi) (circ po) (circ pu)))
 1 2 4 8 16)
```

et

```
((lambda (a e i o u)
   (+ (circ a) (circ e) (circ i) (circ o) (circ u)))
 1 2 4 8 16)
```

ont toujours même valeur, ce qui semble naturel.[50] Observons enfin que, d'une manière générale, la seule façon efficace qu'ait l'esprit humain d'appréhender une entité complexe, qu'il s'agisse d'un programme, d'un moteur de voiture, d'un circuit électronique ou d'une phrase de Cicéron, est de pouvoir décomposer cette entité en composants ayant chacun leur signification propre, et tels que la signification du tout puisse s'obtenir à partir de la signification des composants. Dans le cas d'un programme écrit dans un langage à portée dynamique, cette décomposition est plus difficile à réaliser, puisque la signification d'une définition, ou d'un groupe de définitions, peut dépendre d'éléments extérieurs à ce groupe.

[50] Avec la règle de portée dynamique, nous venons de voir que ce n'était pas le cas.

Remarque. On peut trouver des cas où la règle de portée dynamique serait commode. Considérons les définitions suivantes :

```
(define generic_circ (lambda (r) (* 2 pi r))) ==> ...
(define real_circ (lambda (pi) circ)) ==> ...
```

Dans un système à portée dynamique, on pourrait calculer des circonférences en utilisant selon les circonstances diverses approximations de `pi`, et on aurait

```
((real_circ 3.14) 1) ==> 6.28
```

et

```
((real_circ 3.1416) 1) ==> 6.2832
```

Pour obtenir le même effet en SCHEME, on pourrait redéfinir à chaque fois la variable globale `pi`, mais on a déjà mentionné le danger inhérent aux redéfinitions globales des variables. De manière moins critiquable, on utiliserait en SCHEME le code suivant :

```
(define mk_real_circ
  (lambda (pi)
    (lambda (r) (* 2 pi r))))
```

On aurait alors

```
((mk_real_circ 3.14) 1)  ==>  6.28
((mk_real_circ 3.1416) 1)  ==>  6.2832
```

La différence essentielle est qu'en portée lexicale, toute dépendance est matérialisée par un élément syntaxique, lexical. En SCHEME, la valeur de l'expression `((lambda (pi) circ) x)` est indépendante de la valeur de `x`, puisque l'expression `circ` ne comporte pas d'occurrence du paramètre `pi`. En fait, dès que `x` et `circ` ont une valeur, on a toujours

$$[[\texttt{((lambda (pi) circ) x)}]] = \texttt{circ}$$

3.3.3 Exemple supplémentaire

Un exemple supplémentaire va permettre de mieux voir les implications de la règle de portée statique et lexicale des variables. Tous les environnements dont il est question dans ce paragraphe sont représentés à la figure 9. L'exemple commence par les deux définitions suivantes :

```
(define add_7
  ((lambda (a f)
     (lambda (x) (f a x)))
   7
   +))

(define add_8
```

```
(lambda (x)
  ((lambda (a f) (f a x))
   8
   +)))
```

L'évaluation de la première forme `define` dans l'environnement global E_0 commence par l'évaluation de la combinaison `((lambda (a f) (lambda (x) (f a x))) 7 +)`

La valeur de l'opérateur est la fermeture `[(a f);(lambda (x) (f a x));`E_0`]`, qui doit être appliquée aux valeurs des opérandes, soient `7` et `+`; l'environnement E_1 est créé, où `a` et `f` sont liés à `7` et `+`. Le corps `(lambda (x) (f a x))` est évalué dans E_1, ce qui donne la fermeture `[(x);(f a x);`E_1`]`; enfin, cette valeur est liée à la variable `add_7` dans l'environnement E_0.

L'évaluation de la deuxième forme `define` dans l'environnement global E_0 commence par l'évaluation de la λ-forme

`(lambda (x) ((lambda (a f) (f a x)) 8 +))`

Sa valeur est la fermeture

`[(x);((lambda (a f) (f a x)) 8 +));`E_0`]`;

cette valeur est liée à la variable `add_8` dans l'environnement E_0.

Considérons ensuite la session suivante :

```
(define f *)  ==>  ...
(define a 0)  ==>  ...
(add_7 100)   ==>  107
(add_8 100)   ==>  108
```

Les évaluations des deux formes `define` conduisent à lier dans le cadre C_0 les variables `f` et `a` aux valeurs [[`*`]] et `0`, respectivement (cf. figure 9). La troisième forme à évaluer est une combinaison. La valeur de l'opérateur dans E_0 est la fermeture `[(x);(f a x);`E_1`]`, la valeur de l'opérande est `100`. L'application de la fermeture à la valeur `100` consiste à créer un environnement E_2, dont l'environnement d'accès est E_1 (et l'environnement de contrôle est E_0) où `x` est lié à `100` et dans lequel `(f a x)` est évalué; les valeurs de `f` et `a` sont trouvées dans l'environnement d'accès E_1 et celle de `x` dans l'environnement local E_2. La valeur de `f`, c'est-à-dire [[`+`]], est appliquée aux valeurs de `a` et `x`, c'est-à-dire `7` et `100`, ce qui donne la valeur finale `107` (cf. figure 9). On observe que les liaisons *globales* relatives à `f` et `a` n'ont pas eu d'effet sur cette valeur finale.

La quatrième forme à évaluer est une combinaison. La valeur dans E_0 de l'opérateur est la fermeture `[(x);((lambda (a f) (f a x)) 8 +));`E_0`]`; la valeur de l'opérande est `100`. L'application de la fermeture à la valeur `100` consiste à créer un environnement E_3, dont l'environnement de contrôle et d'accès est E_0; la variable `x` est liée à `100` dans C_3 et le corps `((lambda (a f) (f a x)) 8 +)` est évalué dans E_3. Il s'agit d'une combinaison dont l'opérateur est une λ-forme dont la valeur est la fermeture `[(a f);(f a x);`E_3`]`; les valeurs des opérandes sont `8` et [[`+`]]. L'application de la fermeture aux deux arguments consiste en la création d'un environnement E_4 où `a` et `f` sont liés à `8` et [[`+`]], respectivement, et dans lequel le corps `(f a x)` est évalué. Les valeurs de `f` et `a` sont trouvées localement,

la valeur de `x` est trouvée dans l'environnement d'accès E_3. L'application de [[+]] à `8` et `100` donne la valeur finale `108`.

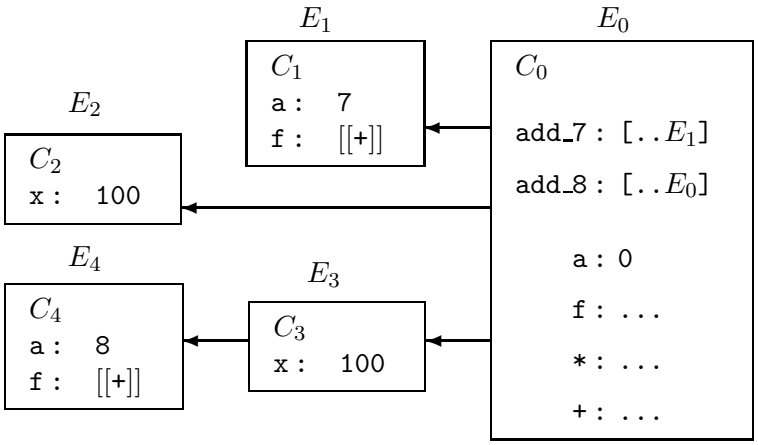


Figure 9: Evaluation de `(add_7 100)` et `(add_8 100)`

Ces exemples mettent en évidence le concept de variable libre; le lecteur est invité à justifier de la même manière la session suivante :

```
((lambda (a f) (add_8 100)) 3 -)  ==>  108
((lambda (a f) (add_7 100)) 4 -)  ==>  107
```

3.3.4 Inférence statique et lexicale des liaisons

La règle de portée statique de SCHEME est dite lexicale parce qu'il est possible d'inférer les liaisons sur base du seul texte de la forme à évaluer et des définitions concernées par cette évaluation; nous précisons ici comment cette inférence se fait.

Les paramètres d'une λ-forme sont liés à des valeurs lorsque la fermeture, valeur de la λ-forme, est appliquée. Seules les occurrences des paramètres dans le corps de la λ-forme sont concernées par ces liaisons; les occurrences extérieures sont des occurrences de variables distinctes, même si elles portent le même nom. Le parenthésage rigoureux des expressions en SCHEME permet en effet d'utiliser le même nom de variable dans des zones distinctes, sans que l'expression devienne ambiguë. Notons aussi que le corps d'une λ-forme peut contenir d'autres λ-formes, dont les paramètres ne portent pas nécessairement des noms distincts. C'est la λ-forme la plus interne qui prime, comme le montraient déjà les exemples donnés plus haut. La règle de portée des variables détermine la portion du programme dans laquelle une liaison variable-valeur est valide. Nous illustrons cette règle au moyen d'un exemple.

Une expression, à évaluer dans l'environnement global, est représentée à la figure 10. Cette expression comporte plusieurs occurrences de `x`. Quatre liaisons relatives à `x` sont susceptibles d'entrer en ligne de compte. Il y a d'abord les trois liaisons dues aux trois

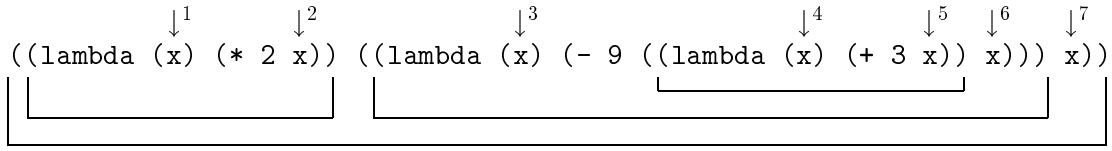


Figure 10: Portée

λ-formes que comporte l'expression. En ce qui concerne une quatrième liaison pertinente éventuelle, on distingue deux cas.

1. Une liaison existe pour `x` dans l'environnement global, suite à l'évaluation d'une forme `(define x ...)`. Cette liaison est aussi à considérer.

2. Dans le cas contraire, certaines occurrences de `x` dans l'expression pourraient n'avoir pas de valeur.

L'expression de la figure 10 est une combinaison à deux éléments; le premier est la λ-forme `(lambda (x) (* 2 x))`, que nous notons Λ_1. Le second élément de la combinaison est lui-même une combinaison à deux éléments; le second élément est `x` tandis que le premier est la λ-forme `(lambda (x) (- 9 ((lambda (x) (+ 3 x)) x)))`, que nous notons Λ_2. Le corps de Λ_2 est une combinaison arithmétique dont le dernier élément est une combinaison dont le foncteur est la λ-forme `(lambda (x) (+ 3 x))`, que nous notons Λ_3. Nous pouvons maintenant décrire les trois λ-formes.

- L'unique paramètre de Λ_1 est identifié par l'exposant 1 (cf. figure 10); l'occurrence repérée par l'exposant 2 se trouve dans le corps de Λ_1 et aussi dans sa *portée*.

- L'unique paramètre de Λ_2 est identifié par l'exposant 3; les occurrences 5 et 6 se trouvent dans le corps de Λ_2 mais seule l'occurrence 6 se trouve dans la portée de Λ_2.

- L'unique paramètre de Λ_3 est identifié par l'exposant 4; l'occurrence 5 se trouve dans le corps de Λ_3 et aussi dans sa portée.

L'occurrence 7 ne se trouve dans le corps d'aucune λ-forme; c'est une variable globale (ou une occurrence globale de l'identificateur `x`).

Une occurrence d'une variable peut se trouver dans le corps de plusieurs λ-formes imbriquées mais elle se trouve dans la portée d'une seule, la λ-forme la plus interne qui contient cette occurrence. On a vu qu'en cas d'application d'une λ-forme à des arguments, les paramètres étaient liés à ces arguments, puis le corps de la λ-forme était évalué. Ce corps contient souvent une ou plusieurs occurrences des paramètres, mais seules celles qui se trouvent dans la portée de la λ-forme sont concernées par ces liaisons.

3.3.5 Deux exercices

Pour s'assurer de sa bonne compréhension du modèle des environnements, le lecteur est invité à évaluer la forme

```
((lambda (x) (* x x))
 ((lambda (a b) (a (* b b)))
  (lambda (a) (* a a))
  (* 2 2)))
```

selon ce modèle (la valeur cherchée est `65 536`).

De même, le lecteur pourra déterminer que la valeur de la forme

```
((lambda (a b c) (a b c))
 (lambda (x y) (* 2 x y))
 ((lambda (x y) (+ x y)) (+ 1 2) (+ 3 4))
 ((lambda (x) (+ 3 x)) 0))
```

est `60`.

3.3.6 Occurrences libres, occurrences liées

Les subtilités inhérentes aux notions de variable et de portée de variable paraissent moins rebutantes si on observe qu'elles ne sont pas spécifiques à la programmation, comme le montrent les exemples suivants. Considérons l'expression mathématique E définie par

$$E \;=_{def}\; 1 - \int_0^x \sin(u)\,\mathrm{d}u\,.$$

Par analogie avec SCHEME, nous pouvons dire que 0 et 1 sont des constantes numériques, tandis que "sin" est une variable globale, préliée à la fonction trigonométrique sinus.[51] Le symbole — bizarre — "$\int \ldots \mathrm{d} \ldots$", tel qu'il apparaît dans la sous-expression "$\int_0^x \sin(u)\,\mathrm{d}u$", est aussi une variable globale, prélié à l'opération d'intégration, c'est-à-dire à une fonction à trois arguments : les deux limites d'intégration et la fonction à intégrer. Si on rationalisait l'écriture mathématique, on écrirait

$$\int (a, b, f)$$

plutôt que

$$\int_a^b f(u)\,\mathrm{d}u\,.$$

Revenons à l'expression mathématique E définie plus haut. Par analogie avec la programmation, il est naturel de considérer que la valeur de E n'existe que dans un

[51] Le mathématicien dira plutôt que "sin" est une constante fonctionnelle, parce qu'il exclut d'emblée la possibilité théorique de renommer la variable; une variable qui fait l'objet d'une liaison permanente et intangible est assimilable à une constante.

environnement où x a une valeur, le fait que u ait une valeur ou non étant non pertinent. L'usage mathématique permet naturellement cela mais il est plus laxiste que SCHEME : si x n'a pas de valeur, on considère que l'expression E est une abréviation de l'expression $\lambda x.E$; sa valeur est donc une fonction de $\mathbb{R}$ dans $\mathbb{R}$. En fait, on ne peut *jamais* confondre une λ-forme `(lambda (x) e)` et le corps e de cette forme.

En ce qui concerne les possibilités de renommage de variables, il y a une forte analogie entre le langage SCHEME et la logique classique. La valeur de vérité de la formule $\forall x\, P(x, y)$ dépend de la valeur de la "variable libre" y (et de l'interprétation de la constante prédicative P) mais pas de la valeur de la "variable liée" x qui peut être renommée : $\forall x\, P(x, y)$ et $\forall z\, P(z, y)$ sont des formules logiquement équivalentes.[52] Le même phénomène s'observe en SCHEME. Pour évaluer l'expression `((lambda (x) (+ x y)) 5)` dans l'environnement global E_0, il sera nécessaire que `y` ait une valeur dans cet environnement courant; par contre, la valeur éventuelle de `x` dans E_0 est sans importance. On peut d'ailleurs remplacer les deux occurrences de `x` par `z`, par exemple.[53]

Remarque. Même si `y` n'est pas lié dans l'environnement global, l'évaluation de

```
(define f (lambda (x) (+ x y)))
```

ne provoque pas d'erreur; par contre, l'évaluation subséquente de `(f 5)`, par exemple, provoquera une erreur.

Remarque. On observe aussi que la valeur de

```
((lambda (u) (+ u 5)) (* x 3))
```

dans l'environnement global dépend de celle de `x` dans cet environnement mais que la valeur de

```
(lambda (x) ((lambda (u) (+ u 5)) (* x 3)))
```

n'en dépend pas.

Soit une λ-forme du type `(lambda (...x...) E)`. Toute occurrence de `x` dans le corps `E` est dite *liée*. Par extension, une occurrence de `x` est liée dans une expression α si cette expression comporte une sous-expression (forme `lambda`) dans laquelle l'occurrence en question est liée.

Exemple. Dans l'expression

```
((lambda (x) (- 9 ((lambda (x) (+ 3 x)) x))) x)
                                          ^   ^
```

[52] Par contre, on ne peut pas récrire $\forall x\, P(x, y)$ en $\forall y\, P(y, y)$; il y aurait *capture* de la variable second argument y, libre dans la première expression et liée dans la seconde.

[53] Sur le plan du renommage des variables, SCHEME, vu sa politique stricte de parenthésage et de notation préfixée, permet certaines écritures que l'usage mathématique interdirait. Par exemple, on peut écrire `((lambda (x) (+ x 5)) x)` au lieu de `((lambda (u) (+ u 5)) x)` alors qu'il n'est pas permis d'écrire $\int_0^x \sin(x)\,\mathrm{d}x$ au lieu de $\int_0^x \sin(u)\,\mathrm{d}u$.

les deux premières occurrences pointées de `x` sont liées, la dernière occurrence de `x` est non liée. Dans l'expression

```
(* 9 ((lambda (x) (+ 3 x)) x) x)
                        ^
```

l'occurrence pointée de `x` est liée, les deux occurrences suivantes de `x` sont non liées. Les variables liées sont habituelles en logique, et aussi en mathématique (indice de sommation, variable d'intégration).

3.4 Procédures `eval` et `apply`

Le processus d'évaluation d'une expression SCHEME est récursif. Il y a des cas de base (constantes, variables, formes `quote`) et des cas inductifs. Pour ces derniers, l'évaluation de l'expression passe par l'évaluation de sous-expressions et aussi, parfois, par le processus d'application d'une valeur fonctionnelle à ses arguments. Le processus d'application lui-même peut requérir de nouvelles évaluations. C'est le cas si la fonction à appliquer est "λ-définie"; il y a lieu alors d'évaluer le corps de la λ-forme dans un nouvel environnement, enrichi des liaisons entre paramètres formels et paramètres réels.

On peut donc décrire précisément le système autour de deux procédures mutuellement récursives `eval` et `apply`. La première prend comme arguments une expression à évaluer et un environnement. La seconde prend comme arguments une valeur fonctionnelle et une liste d'arguments évalués appropriés (en nombre et en type). En fait, on peut construire le système sur cette base.

On peut aussi, pour s'assurer que l'on comprend bien les fonctionnalités du système, le reconstruire sur base de fonctions `eval` et `apply`, qui peuvent être programmées dans un langage quelconque ... y compris SCHEME lui-même. Ce genre d'exercice permet aussi de créer des variantes du vrai système SCHEME et de les expérimenter. Cette démarche requiert que l'on représente les expressions SCHEME à évaluer par des structures de données du langage dans lequel on a programmé `eval` et `apply`. Si on utilise pour cela le langage SCHEME lui-même, on utilisera naturellement la représentation habituelle des expressions en SCHEME; on pourra aussi représenter les environnements par des listes de liaisons, chaque liaison étant une liste (ou une paire) variable-valeur. Par exemple, on devra avoir

```
(eval '(+ 3 4) env)  ==>  7
```

et aussi

```
(eval (list '+ 3 4) env)  ==>  7
```

puisque l'on a

```
(+ 3 4)  ==>  7
```

Construire en SCHEME ce couple de fonctions `eval` et `apply` (et, bien sûr, une série de fonctions auxiliaires) permet d'obtenir un "méta-évaluateur" ou "évaluateur méta-circulaire". C'est une tâche bien moins lourde que la construction d'un évaluateur ordinaire

(écrit par exemple en langage C ou en langage d'assemblage), mais qui sort néanmoins du cadre de ce texte introductif. Il faut cependant noter que les systèmes SCHEME fournissent une procédure `apply`.[54] La procédure `apply` fournie par le système est telle que, si [[`proc`]] est une procédure à n arguments, et si [[`x1`]],...,[[`xn`]] sont des arguments appropriés, alors

$$[[(\texttt{applyproc}(\texttt{listx1...xn}))]] \;=\; [[(\texttt{procx1...xn})]]\,.$$

Cette procédure peut être utile même en programmation élémentaire. Supposons par exemple que l'on veuille construire une fonction qui associe à toute liste de nombres la somme de ses éléments. On peut écrire

```
(define +_list (lambda (l) (apply + l)))
```

On notera la différence entre "`+`" et "`+_list`" :

```
(+ 1 2 3 4) ==> 10
(+_list '(1 2 3 4)) ==> 10
(+ '(1 2 3 4)) ==> Error
```

Cela se généralise à tout opérateur admettant un nombre quelconque d'arguments; si nous définissons

```
(define op_list (lambda (op l) (apply op l)))
```

nous pouvons par exemple calculer la somme et le produit d'une liste de nombres :

```
(+ 1 2 3 4) ==> 10
(op_list + '(1 2 3 4)) ==> 10
(* 1 2 3 4) ==> 24
(op_list * '(1 2 3 4)) ==> 24
```

On peut aussi procéder autrement et définir l'opérateur fonctionnel `gen_op_list` qui transforme une fonction à nombre quelconque d'arguments $x_1, \ldots, x_n$ en la fonction qui prend comme seul argument la liste correspondante $(x_1, \ldots, x_n)$:

```
(define gen_op_list
  (lambda (op)
    (lambda (l) (apply op l))))
```

La transformation inverse se programme facilement, en utilisant la forme `lambda` généralisée :

```
(define inv_gen_op_list
  (lambda (op_list)
    (lambda v (op_list v))))
```

[54] Les systèmes SCHEME fournissent une procédure `eval`, plus ou moins générale, mais nous ne décrirons pas cela ici.

On a alors

```
(op_list + '(1 2 3 4))  ==>  10
(op_list * '(1 2 3 4))  ==>  24
((gen_op_list +) '(1 2 3 4))  ==>  10
((gen_op_list *) '(1 2 3 4))  ==>  24
((inv_gen_op_list +_list) 1 2 3 4)  ==>  10
((inv_gen_op_list (gen_op_list *)) 1 2 3 4)  ==>  24
((inv_gen_op_list (gen_op_list proj_1)) 1 2 3 4)  ==>  1
```

3.5 Autres formes conditionnelles

3.5.1 La forme spéciale `cond`

La forme spéciale `cond` généralise la forme spéciale `if`. La syntaxe habituelle de la forme `cond` est `(cond (`c_1 e_1`) ... (`c_n e_n`))`. Les `(`c_i e_i`)` sont des *clauses*. Le processus d'évaluation est le suivant :

- Les conditions c_i sont évaluées en séquence jusqu'à ce que une valeur $[[c_k]]$ distincte de `#f` soit produite;
- L'expression correspondante e_k est évaluée; la valeur produite $[[e_k]]$ est la valeur de la forme spéciale.

La valeur de la forme `cond` est non définie si toutes les valeurs $[[c_i]]$ s'identifient à `#f`. Une pratique fréquente pour éviter systématiquement ce type de problème consiste à choisir comme dernière condition c_n une expression dont la valeur est toujours `#t`. Dans ce contexte, le symbole spécial `else` peut être utilisé. La forme `(cond (`e_0 e_1`) (else` e_2`))` est donc équivalente à la forme `(if` e_0 e_1 e_2`)`.

Voici quelques exemples :

```
(cond ((> 0 1) (/ 2 0))
      ((> 1 2) (/ 0 1))
      ((> 2 0) (/ 1 2)))  ==>  1/2

(cond ((> 3 5) hi) (else 4))  ==>  4

(cond ('hi 3) (else 4))  ==>  3

(cond ((> 0 1) 5))  ==>  ...

(cond)  ==>  ...
```

3.5.2 La fonction `not`

La valeur de la forme `(not` e`)` est `#t` si la valeur de e est `#f`; elle est `#f` sinon. On peut voir `(not` e`)` comme l'abréviation de `(if` e `#f #t)`. A titre d'exemple, signalons l'existence de deux reconnaisseurs très utiles, `pair?` et `atom?`. Le premier reconnaît tout objet construit au moyen de `cons`, notamment les listes non vides (et d'autres objets plus généraux introduits plus loin); le second reconnaît les objets qui ne sont pas construits au moyen de `cons`, notamment les nombres, les symboles et la liste vide. Certains systèmes prédéfinissent les deux reconnaisseurs, d'autres imposent à l'utilisateur de dériver le second du premier, en évaluant le code

```
(define atom?
  (lambda (x) (not (pair? x))))
```

Remarque. Avec cette définition, [[`(atom? x)`]] = [[`#t`]] signifie que les fonctions [[`car`]] et [[`cdr`]] ne peuvent s'appliquer à [[`x`]]. C'est le cas des objets "atomiques" au sens usuel du terme, mais aussi des fonctions, et d'autres objets composés comme les vecteurs introduits plus loin.

3.5.3 Les formes spéciales `and` et `or`

La valeur de la forme spéciale `(and` e_1 $\ldots e_n$`)` s'obtient comme suit. Les e_i sont évalués dans l'ordre et la première valeur fausse est retournée; les valeurs suivantes ne sont pas calculées. S'il n'y a pas de valeur fausse, la dernière valeur calculée est retournée. La valeur de `(and)` est `#t`.

La valeur de la forme spéciale `(or` e_1 $\ldots e_n$`)` s'obtient comme suit. Les e_i sont évalués dans l'ordre et la première valeur non fausse est retournée; les valeurs suivantes ne sont pas calculées. Si toutes les valeurs sont fausses, `#f` est retourné; la valeur de `(or)` est `#f`.

Les exemples qui suivent illustrent l'usage des formes `and` et `or`

```
(and (= 2 2) (< 2 1) (/ 2 0) (+ 2 3))  ==>  #f
(and (= 2 2) (+ 2 3) (/ 2 0) (< 2 1))  ==>  Error - Div by zero
(and (= 2 2) (< 2 1) (/ 2 0) (+ 2 3))  ==>  #f
(and (= 2 2) (+ 2 3))  ==>  5
(and (+ 2 3) (= 2 2))  ==>  #t
(if (and (= 2 2) (+ 2 3)) 'hi 'ha)  ==>  hi

(or (< 2 1) (= 2 2) (/ 2 0) (+ 2 3))  ==>  #t
(or (< 2 1) (+ 2 3) (/ 2 0) (= 2 2))  ==>  5
(or (< 2 1) (/ 2 0) (+ 2 3) (= 2 2))  ==>  Error - Div by zero
(or (= 2 2) (+ 2 3) (< 2 1))  ==>  #t
(or #f (< 2 1) #f)  ==>  #f
(if (or (+ 2 3) 1 2) 'hi 'ha)  ==>  hi
```

3.5.4 Relations entre `if` et `cond`

La forme `if` est un cas particulier de la forme `cond`; plus précisément, les formes `(cond (`e_0 e_1`) (else` e_2`))` et `(if` e_0 e_1 e_2`)` ont même valeur.

On peut aussi, dans certaines conditions, ramener une forme `cond` à une série de formes `if` emboîtées; il suffit d'utiliser l'équivalence mentionnée ci-dessus pour démontrer par récurrence sur n que la forme

`(cond (`c_0 e_0`) (`c_1 e_1`) ... (`c_{n-1} e_{n-1}`) (else` e_n`))`

a même valeur que la forme

`(if` c_0 e_0 `(if` c_1 e_1 `(if ...(if` c_{n-1} e_{n-1} e_n`)...))).`

4 Procédures récursives

4.1 Préliminaires

La notion de procédure est cruciale en programmation parce qu'elle permet de résoudre élégamment de gros problèmes en les ramenant à des problèmes plus simples. De même, la notion de fonction est cruciale en mathématique parce qu'elle permet de construire et de nommer des objets complexes en combinant des objets plus simples. La définition

$$hypo =_{def} \lambda x, y : \sqrt{x^2 + y^2}$$

ainsi que le programme

```
(define hypo
  (lambda (x y)
    (sqrt (+ (square x) (square y)))))
```

sont des exemples classiques. La définition (opérationnelle) de la longueur de l'hypoténuse suppose les définitions préalables de l'addition, de l'élévation au carré et de l'extraction de la racine carrée.

Remarque. L'ordre dans lequel on *définit* les procédures n'a pas d'importance. On peut définir `hypo` avant de définir `square`, quoique le corps de la λ-forme liée à `hypo` contienne des occurrences de `square`. C'est seulement lors de l'application de `hypo` qu'une valeur sera cherchée pour la variable `square`, dans l'environnement de définition de `hypo`.

L'idée de la récursivité est d'utiliser, dans la définition d'un objet relativement complexe, non seulement des objets antérieurement définis, mais aussi l'objet à définir lui-même. Le risque de "cercle vicieux" existe, mais n'est pas inévitable. Nous avons déjà introduit la récursivité dans le chapitre introductif. Avant d'y revenir plus concrètement ici, nous allons montrer que l'idée de récursivité, loin d'être "moderne", n'est qu'un cas particulier de la notion classique d'équation.

4.2 Récursivité et équations

On peut faire une analogie avec les égalités et les équations. Dans un contexte où f, a et b sont définis, on peut définir x par l'égalité

$$x = f(a, b)\,.$$

Par contre, l'égalité

$$x = f(a, x)$$

ne définit pas nécessairement un et un seul objet x; même si c'est le cas, le procédé de calcul de x peut ne pas exister.

Un procédé de calcul parfois utilisé pour résoudre l'équation $x = f(a, x)$ consiste à construire un certain nombre de termes de la suite

$$x_0,\ x_1 = f(a, x_0),\ x_2 = f(a, x_1),\ \ldots\ ,\ x_{n+1} = f(a, x_n), \ldots$$

Dans certains cas, cette suite converge vers une limite x et, par passage à la limite, on déduit $x = f(a, x)$ de $x_{n+1} = f(a, x_n)$. Cette approche est simple dans son principe mais parfois délicate dans son application : le choix de x_0 n'est pas évident et la convergence de la suite n'est en général pas garantie. Ce procédé est néanmoins souvent utile. Il fournit rapidement, par exemple, une bonne approximation de la solution de l'équation $x = \cos x$.[55]

Remarque. Quand on considère l'équation $x = f(a, x)$, on ne prétend généralement pas "définir" l'objet x mais, plus modestement, un ensemble d'objets, peut-être vide, composé des *solutions* de l'équation. Pour qu'il y ait définition, on doit avoir démontré que l'ensemble comporte un et un seul élément, c'est-à-dire l'existence et l'unicité de la solution.

Le procédé d'approximation peut aussi être utilisé pour calculer des fonctions plutôt que des nombres, et donc pour résoudre des équations fonctionnelles du type

$$f \;=\; \Phi(g, f)\,.$$

L'équation différentielle $y' = f(x, y)$ (avec $y(x_0) = y_0$) peut s'écrire

$$y(x) \;=\; y_0 + \int_{x_0}^{x} f(t, y(t))\,\mathrm{d}t\,,$$

ce qui permet souvent une résolution approchée. Un exemple classique est l'équation $y' = y$, avec la condition initiale $y(0) = 1$. Les approximations successives sont définies par la suite

$$y_0 \;=_{def}\; \lambda x\,.\,1\,,$$
$$y_{n+1} \;=_{def}\; \lambda x\,.\,\left(1 + \int_0^x y_n(t)\,\mathrm{d}t\right)\,.$$

Ce sont les développements finis de MacLaurin; on a successivement :

$$y_0(x) \;=\; 1\,,$$
$$y_1(x) \;=\; 1 + \int_0^x 1\,\mathrm{d}t \;=\; 1 + x\,,$$
$$y_2(x) \;=\; 1 + \int_0^x (1 + t)\,\mathrm{d}t \;=\; 1 + x + x^2/2$$

et, plus généralement,

$$y_n(x) \;=\; \sum_{i=0}^{n} x^i/i!\,.$$

[55] Une calculatrice de poche, en mode "radians", fournit la séquence suivante :
$0, 1, 0.5403, 0.8576, 0.6543, 0.7935, 0.7014, 0.7640, 0.7221, 0.7504, 0.7314, 0.7442, 0.7356, \ldots$;
pour amortir les oscillations, on pose $x_{n+1} = (x_n + \cos x_n)/2$, ce qui donne :
$0, 0.5, 0.6888, 0.7304, 0.7377, 0.7389, 0.7390, \ldots, 0.73908513321516\ldots$

En programmation, on doit associer à toute construction un procédé de calcul clair et précis; on souhaite naturellement qu'il soit aussi raisonnablement efficace.[56] Le mécanisme de définition récursive de fonction respecte en général ces conditions. Une instance du schéma $f = \Phi(g, f)$ que nous avons déjà rencontrée est

$$fact \;=\; \lambda n\,.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * fact(n-1)\,.$$

On définit *fact* en termes d'autres fonctions déjà définies (addition, comparaison, soustraction, multiplication), et de la fonction *fact* elle-même. Il est intéressant de voir que cette écriture donne lieu à une suite d'approximations. Le point de départ est la fonction vide $fact_0$, c'est-à-dire la fonction qui ne comporte aucun couple.[57] On a successivement :

$$\begin{aligned} fact_0 &= \lambda n\,.\,\bot\,,\\ fact_1 &= \lambda n\,.\ \text{if } n = 0 \text{ then } 1 \text{ else } \bot\,,\\ fact_2 &= \lambda n\,.\ \text{if } n = 0 \text{ then } 1 \text{ else if } n-1 = 0 \text{ then } 1 \text{ else } \bot\,,\\ &\ldots \end{aligned}$$

On démontre facilement par récurrence que, pour tout naturel m on a

$$fact_m \;=\; \lambda n\,.\ \text{ if } 0 \le n < m \text{ then } n! \text{ else } \bot\,.$$

On observe qu'il s'agit bien d'une suite d'approximations, chaque élément étant meilleur que le précédent. Ce type d'approximation est très particulier. Un élément de cette suite ne fournit que des réponses exactes, c'est-à-dire des factorielles, mais seulement pour certains éléments du domaine; améliorer une approximation signifie ici étendre le domaine de cette approximation. On remarque aussi que l'extension du domaine se fait de proche en proche. Dans le cas présent, si on peut calculer $fact(x)$, on sait que l'on pourra calculer $fact(x+1)$ au moyen de l'approximation suivante.

En pratique, nous nous limiterons à ce type bien particulier de schéma fonctionnel récursif : l'évaluation de $f(x)$ nécessitera la détermination préalable d'un ensemble de valeurs $f(y_1), \ldots, f(y_n)$ où les $y_1, \ldots, y_n$ sont, en un certain sens, "plus simples" que x.

Si le domaine de la fonction f à définir est tel que tout élément n'admet qu'un ensemble fini d'éléments "plus simples", on conçoit que le risque de "tourner en rond" pourra être évité.

4.3 Quelques exemples

4.3.1 Récursion sur les nombres

Nous donnons d'abord les programmes correspondant aux fonctions définies récursivement dans le chapitre introductif.

[56] En mathématique, toute définition de fonction doit être claire et précise, mais le fait que la fonction f soit correctement définie sur le domaine $\mathbb{Z}$ ne signifie pas que l'on soit capable de calculer $f(r)$ pour tout entier r, ni même pour aucun d'entre eux.

[57] On dit souvent "la fonction définie nulle part". Si l'ensemble de référence est $\mathbb{N}$, on écrira $fact_0(n) = \bot$, ce qui signifie que la valeur de $fact_0(n)$ n'est pas définie. On a, pour toute fonction g, $g(\ldots, \bot, \ldots) = \bot$: une combinaison dont un argument est non défini est elle-même non définie.

La fonction factorielle est définie sur le domaine $\mathbb{N}$ par l'égalité

$$n! \;=\; [\,\text{if } n = 0 \text{ then } 1 \text{ else } n * (n-1)!\,]\,.$$

L'exploitation de cette définition est évidente; on a, par exemple,

$$3! = 3 * 2! = 3 * 2 * 1! = 3 * 2 * 1 * 0! = 3 * 2 * 1 * 1 = 6\,.$$

Le cas de la suite *fib* des nombres de Fibonacci est analogue et déjà connu (cf. § 1.2.1). Voici un nouvel exemple d'exploitation de la définition :

$$\begin{aligned}
&\mathit{fib}(0) = 0\,,\\
&\mathit{fib}(1) = 1\,,\\
&\mathit{fib}(2) = 1 + 0 = 1\,,\\
&\mathit{fib}(3) = 1 + 1 = 2\,,\\
&\mathit{fib}(4) = 2 + 1 = 3\,,\\
&\mathit{fib}(5) = 3 + 2 = 5\,,\\
&\mathit{fib}(6) = 5 + 3 = 8\,,\\
&\ldots
\end{aligned}$$

Dans les deux cas, "plus simple" s'identifie à "plus petit". Sauf dans les cas de base, le calcul de $n!$ repose sur celui de $(n-1)!$ et le calcul de $\mathit{fib}(n)$ sur ceux de $\mathit{fib}(n-1)$ et $\mathit{fib}(n-2)$.

Ceci se traduit aisément en SCHEME :

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))

(define fib
  (lambda (n)
    (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
```

Ces deux programmes permettent respectivement le calcul de la factorielle d'un entier naturel n et le calcul du $n$ième nombre de Fibonacci. Ils ne sont pas destinés à être utilisés avec un argument non naturel et donc rien n'est spécifié pour ce cas, dans lequel le résultat peut même ne pas exister.[58]

Nous avons introduit au paragraphe 1.2.1 les trois fonctions mathématiquement égales $cb1$ et $cb2$, permettant le calcul des coefficients binomiaux; les programmes SCHEME correspondants s'obtiennent immédiatement. On a par exemple

[58] Si on appelle le programme `fact` avec un argument entier négatif, l'exécution ne se termine pas : on dit qu'il y a régression infinie. Concrètement, l'exécution crée une suite du genre $(-4)! = (-4) * (-5)! = (-4) * (-5) * (-6)! = \cdots$, ce qui n'a évidemment ni sens ni intérêt. Si on appelle le programme `fib` avec un argument entier négatif $-n$, le résultat est $-n$, ce qui n'a pas d'intérêt non plus.

```
(define cb1
  (lambda (n p)
    (if (or (= n p) (= p 0))
        1
        (+ (cb1 (- n 1) (- p 1))
           (cb1 (- n 1) p)))))
```

4.3.2 Récursion sur les listes

La récursion s'applique très souvent aux listes. Ecrivons par exemple un programme qui ajoute un nombre [[n]] à tous les éléments d'une liste [[l]] de nombres et renvoie la liste des résultats obtenus :

```
(define add_to_all
  (lambda (n l)
    (if (null? l)
        '()
        (cons (+ n (car l))
              (add_to_all n (cdr l))))))

(add_to_all 2 '(5 3 8))  ==>  (7 5 10)
```

Une variante intéressante permet de traiter des listes comportant des sous-listes de niveau quelconque. Dans ce cas, la récursion opère non seulement sur le reste d'une liste non vide, mais aussi sur la tête, si celle-ci n'est pas un nombre et est donc une liste. On peut utiliser le code suivant :

```
(define add_to_all*
  (lambda (n l)
    (cond ((null? l) '())
          ((number? (car l))
           (cons (+ n (car l))
                 (add_to_all* n (cdr l))))
          (else (cons (add_to_all* n (car l))
                      (add_to_all* n (cdr l)))))))

(add_to_all* 2 '(5 3 12))  ==>  (7 5 14)

(add_to_all* 2 '((5 (() 3)) ((12))))
           ==>  ((7 (() 5)) ((14)))
```

La valeur de `(member x l)` est le premier suffixe de [[l]] qui commence par le symbole [[x]], et `#f` si ce symbole n'appartient pas à la liste :

```
(define member
```

```
(lambda (x l)
  (if (null? l)
      '#f
      (if (eq? x (car l))
          l
          (member x (cdr l))))))
```

On a donc :

```
(member 'x '(a x b x d))  ==>  (x b x d)
(member 'x '(a x))  ==>  (x)
(member 'y '(a x))  ==>  #f
```

Remarque. La définition est en pratique inutile, car `member` est prédéfinie en SCHEME.

Une *table* est une collection de paires dont le premier composant est un symbole et le second une entité se rapportant à ce symbole (une description, une liste de propriétés, etc.). On peut représenter une table par une liste de paires ou de listes dont la tête est un symbole et le reste est l'entité qui s'y rapporte. Dans la (représentation de) table

```
(define *table*
  '((Dubois Pierre 1942) (Dupont Jean 1964) (Durand Paul 1980)))
```

les symboles sont des noms de personnes et les entités associées sont des listes comportant le prénom et l'année de naissance de ces personnes. On définit une fonction [[`assq`]] permettant de retrouver ce qui concerne un symbole donné dans une table :

```
(define assq
  (lambda (symb table)
    (if (null? table)
        #f
        (if (eq? symb (caar table))
            (car table)
            (assq symb (cdr table))))))
```

Notons l'emploi de `caar`, introduit au paragraphe 2.5. On a :

```
(assq 'Dupont *table*)  ==>  (Dupont jean 1964)
(assq 'Martin *table*)  ==>  #f
```

Remarque. La fonction `assq` est prédéfinie en SCHEME, de même que ses variantes `assv` et `assoc`, dans lesquelles le prédicat `eq?` est remplacé par `eqv?` et `equal?`, respectivement.

Considérons encore le cas du produit scalaire de deux vecteurs représentés par deux listes de nombres de même longueur :

```
(define dot_product
  (lambda (u v)
    (if (null? u)
        0
        (+ (* (car u) (car v))
           (dot_product (cdr u) (cdr v))))))
```

On a

```
(dot_product '(1 2 3) '(4 5 6))  ==>  32
```

Remarque. Le programme ne se comporte pas correctement si les deux listes sont de longueurs différentes :

```
(dot_product '(1 2 3) '(4 5))  ==>  Error
(dot_product '(1 2 3) '(4 5 6 7))  ==>  32
```

L'usage de la procédure prédéfinie `error` permet d'attirer l'attention de l'utilisateur sur le problème :

```
(define dot_product
  (lambda (u v)
    (if (null? u)
        (if (null? v)
            0
            (error "second vector too long" v))
        (if (null? v)
            (error "first vector too long" u)
            (+ (* (car u) (car v))
               (dot_product (cdr u) (cdr v)))))))
```

On a alors :

```
(dot_product '(1 2 3) '(4 5))
    ==>  Error - first vector too long (3)
(dot_product '(1 2 3) '(4 5 6 7))
    ==>  Error - second vector too long (7)
```

4.3.3 Schémas de récursion

On peut observer que, souvent, la définition récursive d'une fonction f dont un argument n est un entier naturel consiste d'une part à donner directement la valeur $f(0)$ et, d'autre part, à exprimer $f(n)$ en fonction de $f(n-1)$ (et de n et des autres arguments éventuels) si $n > 0$. De même, très fréquemment, la définition récursive d'une fonction g dont un argument ℓ est une liste consiste d'une part à donner directement la valeur $g(\ell)$ si ℓ est vide et, d'autre part, à l'exprimer en fonction de $g(\ell')$ (et de ℓ et des autres arguments éventuels) si ℓ n'est pas vide et si ℓ' est le reste de ℓ.

Ce mode de définition récursive permet de résoudre facilement la plupart des problèmes usuels, même si des solutions plus complexes sont parfois nécessaires. Nous verrons au chapitre suivant qu'il ne s'agit pas seulement d'un schéma de pensée, mais d'un cadre de programmation très commode.

4.4 Le double rôle de `define`

Comme on l'a vu au paragraphe 3.1, évaluer `(define symb` *exp*`)` consiste à lier à la variable `symb` la valeur de *exp*; le rôle de la forme `define` est donc de donner un nom (`symb`) à une valeur (celle de *exp*). Dans le cas d'une définition récursive, rien ne change techniquement; la valeur de *exp* est une fermeture, liée à `symb`. Du point de vue de l'utilisateur cependant, le rôle de la forme `define` est alors double; il consiste non seulement à *nommer* une fonction mais aussi à la *définir*. Avant l'évaluation de la forme spéciale (définition non récursive)

```
(define square
  (lambda (n) (* n n)))
```

il est déjà possible d'évaluer une combinaison du type

```
((lambda (n) (* n n)) 5)
```

Par contre, avant l'évaluation de la forme spéciale (définition récursive)

```
(define fact
  (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))
```

l'évaluation de la combinaison

```
((lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))) 5)
```

provoque une erreur.

4.5 Le processus de calcul récursif

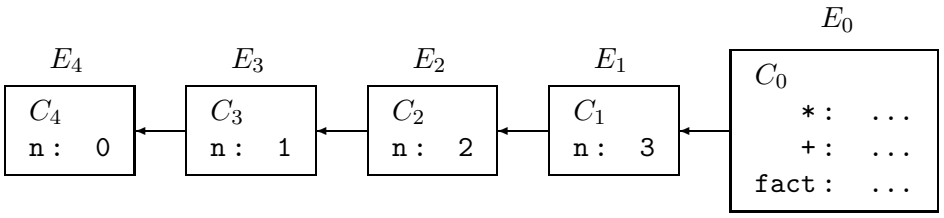
D'après les règles habituelles, on voit que le processus de calcul associé à la définition récursive de la factorielle est relativement "encombrant". On a par exemple, d'après le modèle de substitution

```
(fact 3)
(if (zero? 3) 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
...
(* 3 (* 2 (fact 1)))
...
(* 3 (* 2 (* 1 (fact 0))))
```

```
...
(* 3 (* 2 (* 1 1)))
...
6
```

On conçoit que ce type de calcul requiert un espace-mémoire dont la taille dépend (ici, linéairement) de la valeur de l'argument.

On note aussi que l'introduction de la récursivité n'exige pas de mécanisme particulier pour appliquer une fonction à des arguments : les règles déjà introduites suffisent. En particulier, l'évaluation de `(fact 3)` dans l'environnement global E_0 suscite la structure d'environnements de la figure 11; le tableau explicite les expressions évaluées dans chaque environnement et les valeurs produites. La fermeture liée à `fact` dans l'environnement E_0 indique que E_0 est l'environnement de définition de [[`fact`]].



E_0	`(fact 3)`	`6`
E_1	`(* n (fact (- n 1)))`	`6`
E_2	`(* n (fact (- n 1)))`	`2`
E_3	`(* n (fact (- n 1)))`	`1`
E_4	`1`	`1`

Figure 11: Environnements, application de la fonction `fact`

Il convient d'être attentif au processus de calcul. Celui-ci peut être anormalement long, en temps ou en espace;[59] il peut même être infini, comme le montrent les trois exemples (naïfs) qui suivent :

```
(define f (lambda (x) (f x)))
(define g (lambda (x) (g (+ x 1))))
(define h (lambda (x) (+ (h x) 1)))
```

Sur le plan syntaxique, ces définitions sont parfaitement correctes; il est pourtant clair que l'évaluation des formes `(f 0)`, `(g 1)` et `(h 2)` ne donnera rien ... sauf un gaspillage de ressources ! Le même phénomène de non-terminaison s'était produit pour `(fact -4)`; la procédure `fact` est intéressante, mais l'argument `-4` est inapproprié. L'utilisateur peut

[59] Un processus consommant beaucoup d'espace mémoire consommera aussi, inévitablement, beaucoup de temps, puisque chaque case mémoire devra être créée et/ou visitée; l'inverse n'est pas nécessairement vrai.

éviter ce risque en vérifiant que tout appel pour des valeurs données donne lieu à un nombre fini d'appels subséquents, pour des valeurs "plus simples". Des schémas de programmes récursifs existent, qui garantissent cette propriété de finitude, ou de terminaison. Un autre exemple de processus infini est engendré par la définition

$$f(0) = 1\,, \quad f(x) = 2f(x/2) \text{ si } x > 0\,,$$

dans le domaine des rationnels positifs. On a par exemple

$$f(1) = 2f(1/2) = 4f(1/4) = \cdots = 1024f(1/1024) = \cdots$$

La non-terminaison d'un processus récursif est le risque le plus grave ... mais pas nécessairement le plus pernicieux. Nous avons déjà signalé que le calcul de `(fib n)` se terminait toujours, quel que soit $n \in \mathbb{N}$. Examinons néanmoins ce processus de plus près. Pour $n = 9$, on a :

```
(fib 9)
(+ (fib 8) (fib 7))
(+ (+ (fib 7) (fib 6)) (fib 7))
...
```

Nous avons déjà évoqué ce problème au paragraphe 1.2.6. On démontre facilement par récurrence que le temps de calcul de `(fib n)` est proportionnel à la valeur calculée, ce qui est inacceptable : on a en effet $fib(n) \simeq 1.6^n$.[60] Dans le cas présent, on peut améliorer l'efficacité du calcul en utilisant un autre algorithme, moins naïf, ou en forçant l'évaluateur à mémoriser et à réutiliser les résultats intermédiaires, plutôt qu'à les recalculer. La première approche a été évoquée dans le chapitre introductif, où les bases mathématiques d'algorithmes de calcul en des temps proportionnel à n (fonction *fib_it*), puis à $\log n$, ont été présentés. Voici le programme SCHEME correspondant au premier de ces algorithmes :

```
(define fib_a
  (lambda (n a b)
    (if (zero? n)
        a
        (fib_a (- n 1) b (+ a b)))))
```

On démontre facilement, par récurrence sur n, la propriété $P(n)$ suivante.

> *Si i est un nombre naturel quelconque et*
> *si* `n`, `a` *et* `b` *ont pour valeurs respectives n, $fib(i)$ et $fib(i+1)$,*
> *alors* `(fib_a n a b)` *a pour valeur $fib(n+i)$.*
> *En particulier,* `(fib_a n 0 1)` *a pour valeur $fib(n)$.*

[60] On considère souvent qu'un algorithme est efficace quand le temps d'exécution est borné par une fonction polynomiale en la taille des données. Ici la donnée est un nombre n, dont la taille est $\log n$; l'algorithme est donc *doublement* exponentiel en la taille des données.

Nous reviendrons plus loin sur ce type de récursivité *terminale* ou *dégénérée*, qui se reconnaît par une simple analyse de la syntaxe du programme. Lors de l'évaluation d'un appel, il y a *un seul* appel récursif, et cet appel est *la dernière* action de l'évaluation.[61] Cette analyse permet à l'interprète d'exécuter le programme rapidement et sans consommation inutile de mémoire. Le place mémoire requise par l'exécution du processus est en effet constante. On a par exemple :

```
(fib_a 9 0 1)
(fib_a 8 1 1)
(fib_a 7 1 2)
(fib_a 6 2 3)
(fib_a 5 3 5)
(fib_a 4 5 8)
(fib_a 3 8 13)
(fib_a 2 13 21)
(fib_a 1 21 34)
(fib_a 0 34 55)
34
```

Nous généraliserons plus loin la technique consistant à améliorer un processus de calcul par l'adjonction d'accumulateurs, c'est-à-dire d'arguments supplémentaires (comme `a` et `b` pour `fib_a`) permettant de mémoriser des résultats intermédiaires, et parfois de rendre dégénérée la récursivité. Le temps d'exécution est ici proportionnel à n. Nous verrons plus loin le programme permettant de rendre ce temps proportionnel à $\log n$.

4.6 Récursivité croisée

Le caractère récursif d'une définition peut être indirect; l'expression définissant une fonction f peut contenir des appels à une fonction g, elle-même définie en termes de la fonction f. Un exemple classique très simple est le suivant :

```
(define even?
  (lambda (n)
    (if (zero? n) #t (odd? (- n 1)))))

(define odd?
  (lambda (n)
    (if (zero? n) #f (even? (- n 1)))))
```

Les prédicats `even?` (“pair?”) et `odd?` (“impair?”) sont définis l'un en fonction de l'autre. Voici un exemple du processus de calcul correspondant :

[61] Dans le cas de la factorielle, il y a bien unicité de l'appel récursif, mais, lors de l'évaluation de la forme `(* n (fact (- n 1)))`, cet appel était *l'avant-dernière* action, la dernière étant la multiplication de `(fact (- n 1))` par `n`. La récursivité n'était donc pas terminale.

```
(odd? 4)
(even? 3)
(odd? 2)
(even? 1)
(odd? 0)
#f
```

Rappel. L'ordre dans lequel on définit les procédures n'a pas d'importance. La seule exigence (naturelle) est que les procédures doivent être définies avant d'être appliquées à des arguments. Dans le cas de la récursivité croisée, on évaluera toutes les formes `define` avant d'appliquer l'une des procédures définies.

Remarque. Un bel exemple de récursivité croisée est fourni par l'évaluateur lui-même. Nous avons signalé au paragraphe 3.4 qu'il pouvait être organisé autour de deux procédures, traditionnellement nommées `eval` et `apply`. La procédure `eval` s'appelle elle-même, puisque l'évaluation d'une expression composée requiert en général l'évaluation de sous-expressions; elle appelle aussi `apply`, lors de l'évaluation d'une combinaison. De même, dans le cas où une fonction à appliquer est définie par une λ-forme, la fonction `apply` appelle la fonction `eval`, puisque le résultat de l'application est en fait la valeur du corps de la λ-forme, dans un certain environnement.

5 Récursivité structurelle

Le système SCHEME "accepte" toute définition récursive syntaxiquement correcte, même si la procédure associée donne lieu à des calculs infinis. L'utilisateur doit savoir si, dans un domaine donné, le calcul se terminera toujours. Cette tâche de vérification peut être fastidieuse mais, pour les domaines usuels, des schémas existent dont le respect garantit d'office la terminaison. Les plus utiles de ces schémas sont les schémas *structurels*, basés sur la manière dont les objets du domaine de calcul sont construits. Dans ce cadre, le processus d'évaluation réduit le calcul de $f(v)$ au calcul de $f(v_1), \ldots, f(v_n)$ où les v_i sont des *composants directs* de v. Cette technique est sûre tant que l'on se limite aux domaines dont les objets ont un nombre fini de composants (directs ou non), clairement identifiés.

5.1 Récursivité structurelle sur le domaine des naturels

Conceptuellement, les naturels sont construits à partir de 0 et de la fonction successeur. Le naturel 0 n'a pas de composant; c'est l'unique naturel élémentaire. Tout autre naturel a, par définition, un composant direct, qui est son prédécesseur (cf. § 1.3.1). En conséquence, dans le cas des naturels, la récursivité structurelle consiste à définir $f(0)$ comme une constante, et $f(n)$, avec $n > 0$, comme la valeur d'une expression impliquant un appel à $f(n-1)$ seulement. On peut définir de la sorte des fonctions à plusieurs arguments, mais la récursion "porte" sur un seul de ceux-ci. Voici le schéma de base :

```
(define F
  (lambda (n u1 ... um)
    (if (zero? n)
        (G u1 ... um)
        (H (F (- n 1) (K1 n u1 ... um) ... (Km n u1 ... um))
           n
           u1 ... um))))
```

Les fonctions `G`, `H` et `K1`, ..., `Km` sont supposées déjà définies. On observe que le calcul de `(F 0 u1 ... um)` n'implique pas d'appel récursif, tandis que le calcul de `(F n u1 ... um)` implique un appel récursif qui est du type `(F (- n 1) ...)`, si `n` n'est pas nul; ce dernier implique un appel du type `(F (- n 2) ...)`, et ainsi de suite jusqu'à un appel du type `(F 0 ...)`. Le nombre [[`m`]] d'arguments additionnels (en plus de l'argument sur lequel porte la récursion) est généralement petit. Les cas où il n'y a qu'un argument additionnel et où il n'y en a aucun sont spécialement fréquents; nous en verrons plusieurs exemples. Dans le cas [[`m`]] = `0`, le schéma prend une forme très simple :

```
(define F
  (lambda (n)
    (if (zero? n)
        c
        (H (F (- n 1)) n))))
```

Voici immédiatement quelques exemples d'application élémentaire de la récursivité structurelle sur les naturels.

```
(define harmonic-sum
  (lambda (n)
    (if (zero? n)
        0
        (+ (/ 1 n) (harmonic-sum (- n 1))))))
```

Si $[[\texttt{n}]] = n$, alors $[[\texttt{(harmonic-sum n)}]] = \sum_{i=1}^{n} \frac{1}{i}$.

```
(define mult
  (lambda (n u)
    (if (zero? n)
        0
        (+ u (mult (- n 1) u)))))
```

Si $[[\texttt{n}]] = n$ et $[[\texttt{u}]] = u$, alors $[[\texttt{(mult n u)}]] = n * u$.

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (- n 1))))))
```

Si $[[\texttt{n}]] = n$, alors $[[\texttt{(fact n)}]] = n!$.

```
(define cbin
  (lambda (n u)
    (if (zero? n)
        1
        (/ (* (cbin (- n 1) (- u 1)) u) n))))
```

Si $[[\texttt{n}]] = n$ et $[[\texttt{u}]] = u$, avec $0 \leq n \leq u$ alors $[[\texttt{(cbin n u)}]]$ est le coefficient de x^n dans le développement du binôme $(1 + x)^u$.

Dans chaque cas, le calcul de $[[\texttt{(F n)}]]$ ou de $[[\texttt{(F n u)}]]$ implique une chaîne de $[[\texttt{n}]]$ appels récursifs successifs.

Les schémas sont d'abord des schémas de pensée; ils suggèrent d'exprimer $f(n)$ en termes d'expressions indépendantes de f, mais aussi en terme de $f(n-1)$, si $n > 0$. Les schémas imposent en outre la structure du programme : le programmeur doit seulement définir les fonctions `G`, `H` et `K`. Au lieu d'écrire directement

```
(define cbin
  (lambda (n u)
    (if (zero? n)
        1
        (/ (* (cbin (- n 1) (- u 1)) u) n))))
```

on pourrait récrire le schéma

```
(define F
  (lambda (n u)
    (if (zero? n)
        (G u)
        (H (F (- n 1) (K n u)) n u))))
```

et l'accompagner de définitions auxiliaires :

```
(define G (lambda (u) 1))

(define K (lambda (n u) (- u 1)))

(define H (lambda (r n u) (/ (* r u) n)))

(define cbin F)
```

Remarque. On peut aussi utiliser une variante `FF` du schéma, où les fonctions auxiliaires apparaissent en arguments. On a, par exemple

```
(define FF
  (lambda (G H K)
    (lambda (n u)
      (if (zero? n)
          (G u)
          (H ((FF G H K) (- n 1) (K n u)) n u)))))

(define cbin
        (FF (lambda (u) 1)
            (lambda (r n u) (/ (* r u) n))
            (lambda (n u) (- u 1))))
```

Une autre possibilité est la suivante :

```
(define FFF
  (lambda (G H K n u)
    (if (zero? n)
        (G u)
        (H (FFF G H K (- n 1) (K n u)) n u)))))

(define cbin
  (lambda (n u)
    (FFF (lambda (u) 1)
         (lambda (r n u) (/ (* r u) n))
         (lambda (n u) (- u 1))
         n
         u)))
```

5.2 Les listes

Nous avons déjà mentionné la correspondance naturelle entre les E-listes et les arbres dont les feuilles sont (étiquetées par) des éléments de E. Un exemple concret est le domaine $\mathcal{L}^{Al}$ des *listes littérales*, où Al est l'ensemble des lettres de l'alphabet (Fig. 12). Rappelons que seules les feuilles sont explicitement étiquetées, les nœuds internes ayant une étiquette implicite.[62]

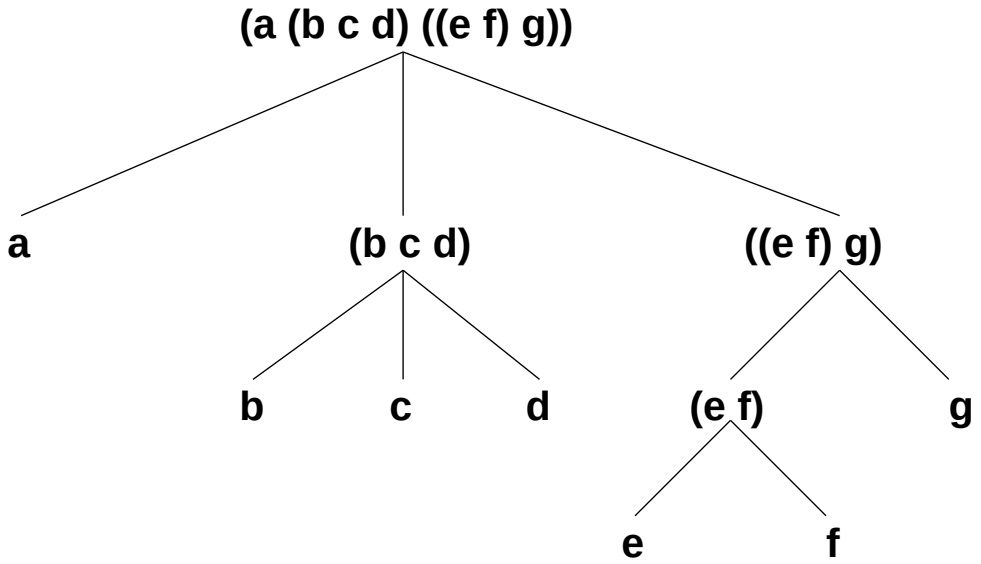


Figure 12: Arbre et liste littérale

5.3 Récursivité superficielle sur les listes

5.3.1 Le schéma de récursion superficielle

Le principe d'induction superficielle sur les listes donne immédiatement lieu au schéma de récursion superficielle sur les listes :

```
(define F
  (lambda (l u1 ... um)
    (if (null? l)
        (G u1 ... um)
        (H (F (cdr l) (K1 l u1 ... um) ... (Km l u1 ... um))
           l
           u1 ... um))))
```

Les fonctions `G`, `H` et `K1`, ..., `Km` sont supposées déjà définies. Calculer la valeur de l'expression `(F l ...)` n'implique pas d'appel récursif quand [[`l`]] est la liste vide; sinon,

[62] Les arbres dont les nœuds internes sont étiquetés (indépendamment des feuilles) forment un autre type de donnée, auquel pourrait correspondre un autre type de liste.

l'évaluation implique une chaîne d'appels récursifs dont la longueur est celle de la liste [[l]]. Comme dans le domaine des nombres naturels, le cas particulier où il n'y a pas d'argument additionnel est d'emploi fréquent :

```
(define F
  (lambda (l)
    (if (null? l)
        c
        (H (F (cdr l)) l))))
```

5.3.2 Exemples élémentaires

Donnons immédiatement quelques exemples d'emploi du schéma de récursion superficielle :

```
(define length
  (lambda (l)
    (if (null? l)
        0
        (+ 1 (length (cdr l))))))

(define append
  (lambda (l v)
    (if (null? l)
        v
        (cons (car l) (append (cdr l) v)))))

(define reverse
  (lambda (l)
    (if (null? l)
        '()
        (append (reverse (cdr l))
                (list (car l))))))

(define map
  (lambda (f1 l)
    (if (null? l)
        '()
        (cons (f1 (car l)) (map f1 (cdr l))))))
```

Observons que la fonction `append` permet de concaténer deux listes, tandis que la fonction `reverse` (qui utilise la fonction `append`) permet de retourner une liste.[63]

Notons aussi que la procédure `map` admet comme premier argument une procédure (à un argument), qui est appliquée à tous les éléments de la liste qui constitue le deuxième argument. Cette fonction `map` s'obtient en instanciant le schéma

[63] Nous employons ici une tournure de langage fréquente et commode, mais abusive : en programmation fonctionnelle, on ne "retourne" pas une liste : on construit une autre liste, version retournée de la première.

```
(define F
  (lambda (l u)
    (if (null? l)
        (G u)
        (H (F (cdr l) (K l u))
           l
           u))))
```

par les paramètres fonctionnels suivants :

```
(define G (lambda (u) '()))

(define H (lambda (r l u) (cons (u (car l)) r)))

(define K (lambda (l u) u))

(define map (lambda (f1 l) (F l f1)))
```

Remarque. La fonction `map` prédéfinie en SCHEME est plus générale; outre l'égalité [[`(map f (list a1 ... an))`]] = [[`(list (f a1) ... (f an))`]], on a aussi, par exemple, [[`(map g (list a1 ... an) (list b1 ... bn))`]] = [[`(list (g a1 b1) ... (f an bn))`]]. On a notamment

```
(map + '(1 2 3) '(10 20 30))  ==>  (11 22 33)
(map + '(1 2) '(10 20) '(100 200))  ==>  (111 222)
```

D'autres fonctions prédéfinies existent; signalons notamment `append` qui réalise la concaténation d'un nombre quelconque de listes :

```
(append)  ==>  ()
(append '(a b c))  ==>  (a b c)
(append '(a b) '() '(c d) '(e))  ==>  (a b c d e)
```

Remarque. La fonction `reverse` permet de comprendre en quoi le schéma de récursion qui vient d'être présenté est "superficiel". La figure 13 comporte à gauche une liste littérale [[`l`]] et à droite la liste [[`(reverse l)`]]; on voit que le retournement est "superficiel" : il ne concerne que les branches du premier niveau dans l'arbre.

5.3.3 Les listes sans répétition

Les listes sans répétition sont souvent utilisées pour représenter des ensembles. Nous reviendrons plus longuement sur les ensembles au paragraphe 10.6; seuls quelques exemples simples sont donnés ici. La fonction `add_elem` ajoute un élément dans une liste, s'il ne s'y trouve pas déjà. Elle utilise la fonction auxiliaire `member` (§ 4.3.2) :

```
(define add_elem
  (lambda (x l)
    (if (member x l) l (cons x l)))))
```

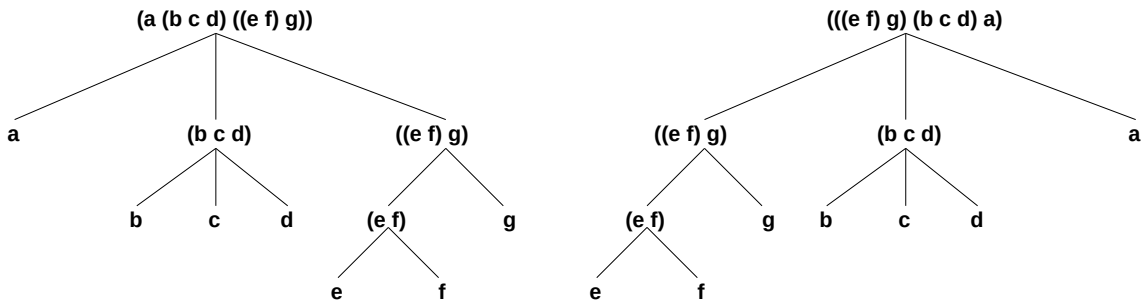


Figure 13: Retournement superficiel d'une liste littérale

La fonction `union` réalise la réunion de deux listes sans répétition :

```
(define union
  (lambda (u v)
    (if (null? u) v (add_elem (car u) (union (cdr u) v)))))
```

Sur le même modèle, on a aussi les fonctions d'intersection et de différence :

```
(define inter
  (lambda (u v)
    (if (null? u)
        '()
        (if (member (car u) v)
            (add_elem (car u) (inter (cdr u) v))
            (inter (cdr u) v)))))

(define diff
  (lambda (u v)
    (if (null? u)
        '()
        (if (member (car u) v)
            (diff (cdr u) v)
            (add_elem (car u) (diff (cdr u) v))))))
```

On a par exemple

```
(define *s1* '(a c d f h))
(define *s2* '(a b e f g))

(union *s1* *s2*)  ==>  (c d h a b e f g)
(inter *s1* *s2*)  ==>  (a f)
(diff  *s1* *s2*)  ==>  (c d h)
```

Remarque. Les deux dernières fonctions peuvent se récrire en utilisant `cond`; on a par exemple

```
(define diff
  (lambda (u v)
    (cond ((null? u) '())
          ((member (car u) v) (diff (cdr u) v))
          (else (add_elem (car u) (diff (cdr u) v))))))
```

5.3.4 Le tri par insertion et l'ordre lexicographique

Une application classique dans le domaine des listes est celle du tri. Elle est pertinente dans le cas de toute liste dont les éléments forment un ensemble totalement ordonné. En programmation fonctionnelle, on ne développera pas d'algorithmes de tri spécifiques à tel ou tel ensemble ordonné : il sera plus commode de considérer que seul le prédicat représentant cet ordre sera spécifique; on le passera comme argument supplémentaire (fonctionnel) à un algorithme de tri qui sera générique, c'est-à-dire valable pour des domaines variés (listes de nombres, de lettres, de mots, de phrases, d'arbres, etc.). On a le programme suivant :

```
(define sort
  (lambda (l comp)   ;; comp: argument procedural
    (if (null? l)
        '()
        (insert (car l) (sort (cdr l) comp) comp))))
```

Remarque. Le point-virgule (éventuellement répété) introduit un commentaire, qui se termine en fin de ligne.

Le schéma habituel s'applique sans surprise. La version triée d'une liste vide est la liste vide. La version triée d'une liste non vide `l` est le résultat de l'insertion à sa place de la tête `(car l)` dans la version triée du reste `(cdr l)` de la liste. La fonction auxiliaire `insert` prend trois arguments : un objet `x`, à insérer dans une liste triée `l`, l'ordre des objets étant spécifié par le prédicat binaire `comp`. Ici aussi, l'application du schéma récursif est aisée :

```
(define insert
  (lambda (x l comp)
    (if (null? l)
        (list x)
        (if (comp x (car l))
            (cons x l)
            (cons (car l)
                  (insert x (cdr l) comp))))))
```

Le cas de base est, comme d'habitude, évident. Le cas inductif concerne l'insertion d'un objet dans une liste triée non vide. Il y a deux sous-cas, dont seulement un donne lieu à un appel récursif : si `x` précède le premier élément de la liste `l`, la tête du résultat est `x` et le reste est `l`. Sinon, la tête du résultat est la tête de `l`, tandis que son reste est le résultat

de l'insertion de `x` dans le reste de `l`. En fait, ce code est suffisamment simple pour se passer de commentaires ! Voici quelques essais :

```
(insert 3 '(0 2 3 3 5 7 8 9) <)  ==>  (0 2 3 3 3 5 7 8 9)
(sort '(8 3 5 7 2 3 9 0) <=)  ==>  (0 2 3 3 5 7 8 9)
(sort '(8 3 5 7 2 3 9 0) >=)  ==>  (9 8 7 5 3 3 2 0)
```

Vérifions que `insert`, par exemple, est bien une instance du schéma de récursion superficielle. Dans le cas de deux arguments additionnels, ce schéma est

```
(define F
  (lambda (l u1 u2)
    (if (null? l)
        (G u1 u2)
        (H (F (cdr l) (K1 l u1 u2) (K2 l u1 u2))
           l
           u1
           u2))))
```

Les instances correspondantes sont

```
(define G (lambda (u1 u2) (list u1)))

(define H
  (lambda (r l u1 u2)
    (if (u2 u1 (car l))
        (cons u1 l)
        (cons (car l) r))))

(define K1 (lambda (l u1 u2) u1))

(define K2 (lambda (l u1 u2) u2))

(define insert (lambda (x l comp) (F l x comp)))
```

Vérifions maintenant que le même programme, appelé avec un prédicat de comparaison lexicale, permet de trier des listes de mots représentés par des chaînes de caractères (les chaînes de caractères constituent un type de données primitif en SCHEME). On a

```
(sort '("bb" "ac" "acb") string<=?) ==> ("ac" "acb" "bb")
```

Etant donné un ensemble E, on note E^* l'ensemble des suites d'éléments de E. Si E est totalement ordonné par une relation notée $\preceq$, on peut toujours définir sur E^* un ordre total noté $\preceq^*$ et qui étende l'ordre $\preceq$.[64] On introduit d'abord deux notations. Si α est

[64] Deux éléments $a, b \in E$ sont aussi éléments de E^*; le fait que $\preceq^*$ étende $\preceq$ signifie que l'on a $a \preceq^* b$ si et seulement si on a $a \preceq b$.

une suite non vide, $hd(\alpha)$ désigne le premier élément de α et $tl(\alpha)$ désigne la suite α privée de son premier élément.[65] De plus, on pose $a \prec b =_{def} (a \preceq b \wedge a \neq b)$ et $a \prec^* b =_{def} (a \preceq^* b \wedge a \neq b)$. On définit alors $\preceq^*$ comme suit. Soient α et β deux suites distinctes.

- $\alpha \preceq^* \alpha$.
- $\alpha \prec^* \beta$ ou $\beta \prec^* \alpha$.
- Si α est vide, alors $\alpha \prec^* \beta$.
- Si α et β sont non vides et si $hd(\alpha) \prec hd(\beta)$, alors $\alpha \prec^* \beta$.
- Si α et β sont non vides et si $hd(\alpha) = hd(\beta)$, alors $\alpha \prec^* \beta$ si et seulement si $tl(\alpha) \prec^* tl(\beta)$.

L'ordre total $\preceq^*$ est l'*extension lexicographique* de l'ordre total $\preceq$.[66]

On définit maintenant une procédure `lex`, prenant comme arguments (procéduraux) une relation d'ordre total strict sur un certain domaine E et la relation d'égalité sur ce domaine. Cette procédure renvoie la version lexicographique (ordre total strict) sur le domaine E^*.

```
(define lex
  (lambda (str-ord iden)  ;; arg proceduraux
    (lambda (u v)
      (cond
        ((null? u) #t)
        ((null? v) #f)
        ((str-ord (car u) (car v)) #t)
        ((iden (car u) (car v))
         ((lex str-ord iden) (cdr u) (cdr v)))
        (else #f)))))
```

Définissons deux cas particuliers fréquents. Le premier concerne les listes de nombres, le second les listes de chaînes de caractères :

```
(define numlex (lex < =))
(define alpha (lex string<=? string=?))
```

On peut maintenant trier des listes de nombres ou de mots :

[65] Les symboles *hd* et *tl* abrègent les mots "head" et "tail".

[66] Si $\preceq$ est l'ordre alphabétique sur les 26 lettres de l'alphabet latin, sa version lexicographique est l'ordre du dictionnaire, donc l'ordre lexicographique usuel, pour les langues utilisant l'alphabet latin.

```
(sort '((2 3) (1 4) (2 3 2) (1 3)) numlex)
  ==>  ((1 3) (1 4) (2 3) (2 3 2))

(sort '(("acb" "ac") ("ac" "acb") ("ac")) alpha)
  ==>  (("ac") ("ac" "acb") ("acb" "ac"))
```

Observons à nouveau l'économie de code rendue possible par une approche réellement fonctionnelle de la programmation, et en particulier par l'usage de procédures admettant des arguments procéduraux et/ou renvoyant un résultat procédural.

5.3.5 Récursivité sur les suites

Certaines fonctions prédéfinies admettent un nombre illimité d'arguments et la forme `lambda` généralisée permet à l'utilisateur de créer lui-même de telles fonctions. La suite des arguments s'apparentant à une liste, il est naturel de penser que les fonctions à nombre illimité d'arguments peuvent être définies récursivement.

Etant donné une fonction $f : D^2 \to D$, on définit une extension f^+ de f sur le domaine $\bigcup_{i>0} D^i$ en posant

$$\begin{aligned} &f^+(x_1) = x_1\,,\\ &f^+(x_1, x_2) = f(x_1, x_2)\,,\\ &f^+(x_1, x_2, x_3) = f(x_1, f(x_2, x_3))\,,\\ &\ldots \end{aligned}$$

On devine la récursivité sous-jacente, qui se résume à l'égalité

$$f^+(x_0, x_1, \ldots, x_n) = f(x_0, f^+(x_1, \ldots, x_n))\,.$$

Si de plus la fonction binaire f admet un neutre à gauche e, c'est-à-dire un élément de D tel que $f(e, x) = x$ pour tout $x \in D$, on peut créer l'extension f^* de f^+ en posant $f^*() = e$. En SCHEME, la fonction `apply` et la forme `lambda` généralisée permettent de définir ces extensions comme suit :

```
(define f*
  (lambda v
    (if (null? v)
        e
        (f (car v) (apply f* (cdr v))))))

(define f+
  (lambda (x . v)
    (if (null? v)
        x
        (f x (apply f+ v)))))
```

On a admis ici l'égalité $e = [[\mathsf{e}]]$.

On peut aussi définir directement les opérateurs d'extensions `*_ext` et `+_ext`; ici, pour le premier opérateur, le premier argument est la fonction binaire à étendre et le second est son neutre à gauche. On a :

```
(define *_ext
  (lambda (f e)
    (lambda v
      (if (null? v)
          e
          (f (car v) (apply (*_ext f e) (cdr v)))))))

(define +_ext
  (lambda (f)
    (lambda (x . v)
      (if (null? v)
          x
          (f x (apply (+_ext f) v))))))
```

Ces deux fonctions sont *a priori* étonnantes et méritent quelques commentaires; nous raisonnons ici sur la seconde. En dépit des apparences, la fonction `+_ext` n'est pas définie récursivement. Son argument est une fonction et nous n'avons pas muni le domaine des fonctions d'une structure permettant de dire qu'une fonction est "plus simple" qu'une autre. En fait, c'est la valeur fonctionnelle retournée par `+_ext` qui est définie récursivement. Cela apparaît mieux si on nomme cette fonction; nous verrons au chapitre 9 comment on peut donner des noms locaux à certains objets. Il est intéressant d'observer que, sur le plan syntaxique, ces définitions semblent faire un usage vicieux de la récursivité; par exemple, `(+_ext f)` est défini en fonction de `(+_ext f)` et non d'un terme `(+_ext g)`, où [[g]] serait "plus simple" que [[f]]. Notons à ce propos que la complexité du processus d'évaluation de la forme `(+_ext f)` ne dépend pas de `f`; par contre, la complexité du processus d'évaluation de la forme `(apply (+_ext f) v)` dépend de `v`; c'est donc la valeur de `v` (une liste) qui doit devenir plus simple (c'est-à-dire plus courte) à chaque appel.

Pour illustrer l'usage des opérateurs d'extension, rappelons la définition de l'opérateur de composition fonctionnelle :

```
(define compose
  (lambda (f g)
    (lambda (x) (f (g x)))))  ==>  ...
```

On a par exemple

```
((compose car cdr) '(1 2 3 4))  ==>  2
((compose (compose car cdr) (compose cdr cdr)) '(1 2 3 4)) ==> 4
```

On voit que pour composer quatre fonctions, trois interventions de l'opérateur `compose` sont nécessaires. Les opérateurs d'extension permettent de simplifier l'écriture. On a :

```
(define id (lambda (x) x))    ;; transformation identique
(define compose* (*_ext compose id))
(define compose+ (+_ext compose))

((compose*) 'x)   ==>  x
((compose+ cdr) '(1 2 3 4))  ==>  (2 3 4)
((compose* car cdr) '(1 2 3 4))  ==>  2
((compose+ car cdr cdr cdr) '(1 2 3 4))  ==>  4
```

Un autre exemple intéressant est celui de la fonction exponentielle `expt`, prédéfinie en SCHEME : on a

```
(expt 2 10)  ==>  1024
```

Observons que la fonction exponentielle admet 1 comme neutre à droite (on a $x^1 = x$ pour tout x) mais n'admet pas de neutre à gauche; on utilisera donc exclusivement l'extension `(+_ext expt)`. Notons aussi que l'exponentielle n'est pas associative. On a

```
((+_ext expt) 3)  ==>  3
((+_ext expt) 3 2)  ==>  9
((+_ext expt) 2 3 2)  ==>  512
((+_ext expt) 2 3 2 1)  ==>  512
((+_ext expt) 2 3 2 1 5)  ==>  512
```

On aurait pu aussi définir directement `expt+` :

```
(define expt+
  (lambda (x . v)
    (if (null? v)
        x
        (expt x (apply expt+ v)))))

(expt+ 2 3 2 1 5)  ==>  512
```

On pouvait encore utiliser une fonction auxiliaire et écrire

```
(define expt_list
  (lambda (l)
    (if (null? (cdr l))
        (car l)
        (expt (car l) (expt_list (cdr l))))))

(define expt+ (inv_gen_op_list expt_list))
```

On utilise ici l'opérateur `inv_gen_op_list` introduit au paragraphe 3.4.

Remarque. La récursivité sur les suites est intéressante, mais nous voyons ici qu'elle n'est pas indispensable : une suite d'arguments peut toujours être remplacée par une liste d'arguments. Au point de vue des performances, ce remplacement est préférable.

A titre d'exemple supplémentaire, créons la fonction de tri `numsort_*` qui renvoie la liste de ses arguments numériques triés par ordre croissant. Un moyen simple passe par l'emploi de la fonction générique `sort` introduite au paragraphe 5.3.4. On a

```
(sort '(8 3 5 7 2 3 9 0) <=)  ==>  (0 2 3 3 5 7 8 9)

(define <=sort_* (lambda (l) (sort l <=)))  ==>  ...

(define numsort_* (lambda v (<=sort_* v)))  ==>  ...

(numsort_* 8 3 5 7 2 3 9 0)  ==>  (0 2 3 3 5 7 8 9)
```

On pouvait aussi utiliser l'opérateur `inv_gen_op_list`

```
(define numsort_* (inv_gen_op_list <=sort_*))
```

On pouvait enfin écrire immédiatement

```
(define numsort_* (lambda v (sort v <=))
```

Cela suggère la généralisation suivante :

```
(define sort_*
  (lambda (comp . v) (sort v comp)))  ==>  ...

(sort_* <= 8 3 5 7 2 3 9 0)  ==>  (0 2 3 3 5 7 8 9)
(sort_* >= 8 3 5 7 2 3 9 0)  ==>  (9 8 7 5 3 3 2 0)
```

5.4 Récursivité profonde sur les listes et les arbres

La récursivité profonde ne s'applique qu'à des listes dont les éléments sont des atomes ou des listes du même type.[67] Concrètement, le schéma superficiel est modifié comme suit : un appel `(F l ...)` peut donner lieu non seulement à un appel récursif du type `(F (cdr l) ...)` mais aussi à un appel similaire portant sur le `car`, du type `(F (car l) ...)`, à condition, naturellement, que `[[(car l)]]` soit une liste. On observera que ceci est calqué sur le principe d'induction profonde introduit au paragraphe 1.3.4. Voilà le schéma général :

```
(define F
  (lambda (l u1 ... um)
    (cond ((null? l) (G u1 ... um))
```

[67] La récursivité profonde s'applique donc aux liste littérales introduites au paragraphe précédent.

```
          ((atom? (car l))
           (H (F (cdr l) (K1 l u1 ... um) ... (Km l u1 ... um))
              l
              u1 ... um))
          (else    ;; (car l) est une liste
           (J (F (car l) (A1 l u1 ... um) ... (Am l u1 ... um))
              (F (cdr l) (D1 l u1 ... um) ... (Dm l u1 ... um))
              l
              u1 ... um)))))
```

Comme d'habitude, notons que le nombre `m` d'arguments additionnels est généralement faible; le cas où il est nul donne lieu à la version simplifiée suivante :

```
(define F
  (lambda (l)
    (cond ((null? l) c)
          ((atom? (car l)) (H (F (cdr l)) l))
          (else (J (F (car l)) (F (cdr l)) l)))))
```

Remarque. On exclut (provisoirement) de rencontrer dans la liste `l` des objets qui ne soient ni des listes ni des atomes.

A titre de premier exemple, voici une fonction qui calcule la frondaison d'une liste littérale, c'est-à-dire la liste des feuilles de l'arbre sous-jacent :

```
(define flat_list
  (lambda (l)
    (cond ((null? l) '())
          ((atom? (car l))
           (if (null? (car l))
               (flat_list (cdr l))
               (cons (car l) (flat_list (cdr l)))))
          (else
           (append (flat_list (car l))
                   (flat_list (cdr l)))))))
```

Voici quelques exemples d'utilisation :

```
(flat_list 'a)  ==>  Error - 5 passed as argument to car
(flat_list '())  ==>  ()
(flat_list '(a (b c) ((((d))) e) b))  ==>  (a b c d e b)
```

Les listes considérées ici ont pour éléments des objets atomiques d'un certain type (par exemple, des lettres) et aussi d'autres listes. Le parallèle entre principes d'induction et schéma de récursion permet d'étendre le domaine $\mathcal{L}^{Al}$ des listes littérales en le domaine $\mathcal{L}_+^{Al}$ des arbres littéraux. Les schémas précédents s'adaptent facilement. Dans le cas simplifié, le schéma

```
(define F
  (lambda (l)
    (cond ((null? l) c)
          ((atom? (car l)) (H (F (cdr l)) l))
          (else (J (F (car l)) (F (cdr l)) l)))))
```

devient

```
(define F
  (lambda (l)
    (cond ((null? l) c)
          ((atom? l) (G l))
          (else (J (F (car l)) (F (cdr l)) l)))))
```

Le programme de calcul de la frondaison d'un arbre littéral est :

```
(define flat_tree
  (lambda (l)
    (cond ((null? l) '())
          ((atom? l) (list l))
          (else
           (append (flat_tree (car l))
                   (flat_tree (cdr l)))))))
```

Ce programme admet maintenant les arguments atomiques; pour le reste, son comportement ne change pas :

```
(flat_tree 'a)  ==>  (a)
(flat_tree '())  ==>  ()
(flat_tree '(a (b c) ((((d))) e) b))  ==>  (a b c d e b)
```

Voilà enfin le programme de retournement profond d'un arbre littéral :

```
(define deeprev
  (lambda (l)
    (cond ((null? l) '())
          ((atom? l) l)
          (else
           (append (deeprev (cdr l))
                   (list (deeprev (car l))))))))
```

L'appel récursif porte sur le `car` *et* sur le `cdr`. On observera le comportement du programme à la figure 14.

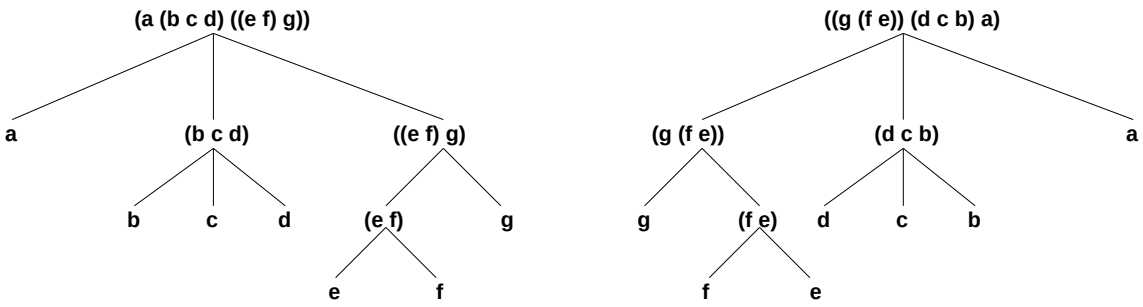


Figure 14: Retournement profond d'une liste littérale

5.5 Remarque sur les schémas de programmes

Suivre un schéma "à la lettre", c'est-à-dire se limiter strictement à instancier les paramètres qu'il contient, rend la programmation particulièrement méthodique et sûre. On peut cependant, pour diverses raisons, garder "l'esprit" d'un schéma sans respecter la lettre (sa syntaxe précise). Un exemple classique est le programme `flat_tree_a`, version équivalente mais plus efficace de `flat_tree` :

```
(define flat_tree_a
  (lambda (l u)
    (cond ((null? l) u)
          ((atom? l) (cons l u))
          (else
           (flat_tree_a
             (car l)
             (flat_tree_a (cdr l) u))))))
```

On montre facilement (par induction sur la structure de `l`) que les valeurs de `(flat_tree_a l u)` et de `(append (flat_tree l) u)` sont égales pour toutes listes `l` et `u`; en particulier, `(flat_tree_a l '())` et `(flat_tree l)` ont même valeur. Par contre, on peut noter que la fonction `flat_tree_a` n'est pas une instance du schéma, même si elle s'en inspire nettement.

5.6 Récursivité structurelle complète et mixte

Les schémas de récursion introduits jusqu'ici correspondent à des principes d'induction simple. Le principe de la récursivité structurelle est que la structure du programme est calquée sur celle des données. On exprime $f(x, \ldots)$ en termes des éléments de l'ensemble $\{f(y, \ldots) : y \prec x\}$, où $y \prec x$ signifie que y est un composant direct de x. On peut naturellement généraliser cela de diverses manières :

- Admettre aussi les composants indirects :
  `(- n 1)`, mais aussi `(- n 2)`, `(/ n 2)`, ...
  `(cdr l)`, mais aussi `(cddr l)`, ...

- Admettre plusieurs appels récursifs.
- Admettre les appels imbriqués.

La première généralisation correspond au passage de l'induction simple à l'induction complète. Pour toutes ces généralisations, la terminaison reste garantie ... mais pas l'efficacité, comme nous avons eu plusieurs fois l'occasion de l'observer. Notons aussi qu'admettre les composants indirects peut légèrement compliquer le test d'existence du composant, toujours indispensable. Il convient d'éviter les erreurs grossières, telles les évaluations de `(f (- n 2))` avec `n` ≤ 1, et de `(f (cddr l))` où `l` comporte moins de deux éléments. Nous avons déjà rencontré des exemples d'emploi de la récursivité structurelle complète, notamment le programme naïf `fib` (voir §§ 1.2.6 et 4.3) et l'exponentielle rapide `exp`, dont voici le code :

```
(define exp
  (lambda (m n)
     (cond ((zero? n) 1)
           ((even? n) (exp (* m m) (/ n 2)))
           ((odd? n) (* m (exp m (- n 1)))))))
```

Le principe de la récursivité structurelle *mixte* est de faire porter la récursivité sur plusieurs arguments. On peut par exemple exprimer $f(x1, x2, \ldots)$ en termes de $\{f(y1, x2, \ldots), f(x1, y2, \ldots), f(y1, y2, \ldots) : y1 \prec x1 \wedge y2 \prec x2\}$. L'induction porte sur plusieurs arguments; si $f(a, b)$ dépend de $f(c, d)$, alors $c \prec a \wedge d \preceq b$ ou $c \preceq a \wedge d \prec b$. La terminaison reste garantie. L'exemple le plus classique est sans doute la version naïve de calcul du plus grand commun diviseurs de deux entiers strictement positifs par la règle d'Euclide :

```
(define gcd
  (lambda (x y)
    (cond ((= x y) x)
          ((> x y) (gcd (- x y) y))
          ((< x y) (gcd x (- y x))))))
```

Un moyen simple de se convaincre que l'évaluation de `(gcd x y)` se termine toujours si les valeurs des opérandes sont des entiers strictement positifs est d'observer que tout appel récursif subséquent se fait avec des arguments dont la somme est strictement inférieure à la somme des arguments initiaux. Une démarche analogue s'applique au programme récursif suivant :

```
(define enum
  (lambda (p q) (if (> p q) '() (cons p (enum (+ p 1) q)))))
```

En effet, `(enum p q)` s'évalue sans appel récursif si [[p]] > [[q]]; sinon, tout appel récursif subséquent se fait avec des arguments dont la différence est strictement inférieure à la différence des arguments initiaux.

5.7 La séparation fonctionnelle

On peut "dérécursiver" le schéma classique

$$fact =_{def} \lambda n\,.\,[\text{if } n = 0 \text{ then } 1 \text{ else } n * fact(n-1)]$$

en l'écriture

$$fact_m =_{def} \lambda n\,.\,[\text{if } n = 0 \text{ then } 1 \text{ else } n * fact_{m-1}(n-1)]\,.$$

La récursivité revient (sous forme dégénérée) pour définir globalement la famille des fonctions $fact_m$:

```
(define f
  (lambda (m c)
    (if (zero? m)
        c
        (f (- m 1)
           (lambda (n)
             (if (= n 0) 1 (* n (c (- n 1))))))))))
```

La fonctionnelle[68] `f` admet un paramètre numérique `m` et un paramètre fonctionnel `c`; quand leurs valeurs respectives sont m et $fact_p$, la valeur de `(f m c)` est $fact_{p+m}$; en particulier on a :

```
(define fact0 'emptyfunction)

(define fact (lambda (n) ((f (+ n 1) fact0) n)))
```

On observe que $fact_m$ a pour domaine $\{0, 1, \ldots, m-1\}$. Sur ce domaine, on a $fact_m(n) = n!$. On a par exemple

```
(define fact8 (f 8 fact0))
(fact8 7)  ==>  5040
(fact8 8)  ==>  Error
(fact 8)   ==>  40320
```

On a séparé

- le *calcul* de la fonction $fact_p$
- l'*application* de cette fonction à un argument.

[68] On donne parfois ce nom aux fonctions "d'ordre supérieur" dont l'un des arguments, ou le résultat, est de nature fonctionnelle.

On a $fact(n) = fact_p(n)$ si $p > n$; on choisit $p = n + 1$.

La *séparation fonctionnelle* n'apporte rien dans le cas très simple de la fonction factorielle mais, dans le cadre général de la définition récursive de fonctions, cette technique sera parfois très utile ! Plus généralement, l'introduction de paramètres fonctionnels et/ou de fonctionnelles auxiliaires définies récursivement est une technique importante.

Examinons de plus près le processus de calcul lié aux instances du principe de séparation fonctionnelle :

```
(fact 8)
((f 9 fact0) 8)
((f 8 (lambda (n) (if (= n 0) 1 (* n (fact0 (- n 1)))))) 8)
((f 8 fact1) 8)
...
((f 0 fact9) 8)
(fact9 8)
(* 8 (* ...  1))
...
(* 8 5040)
40320
```

La partie fonctionnelle du développement s'achève avant le début du calcul arithmétique proprement dit, c'est-à-dire avant les multiplications.

5.7.1 Application : le double comptage

Le problème du *double comptage*, quoique très simple, permet une première illustration de la technique de séparation fonctionnelle. On souhaite parcourir une liste et dénombrer d'une part les éléments possédant une certaine caractéristique et, d'autre part, ceux ne la possédant pas. On peut par exemple dénombrer les "a" et les "non-a" dans une liste de lettres, et fournir la liste des deux résultats :

```
(count '(a b a c d a c) 'a)  ==>  (3 4)
;; 3 occurrences de "a", 4 autres occurrences
```

La solution élémentaire consiste à utiliser deux fonctions auxiliaires, une fonction `c-yes` pour compter les "a", une fonction `c-no` pour compter les "non-a", et à regrouper les deux résultats partiels dans une liste :

```
(define count0 (lambda (l s) (list (c-yes l s) (c-no l s))))

(define c-yes
  (lambda (l s)
    (cond ((null? l) 0)
          ((eq? (car l) s) (+ 1 (c-yes (cdr l) s)))
          (else (c-yes (cdr l) s)))))
```

```
(define c-no
  (lambda (l s)
    (cond ((null? l) 0)
          ((eq? (car l) s) (c-no (cdr l) s))
          (else (+ 1 (c-no (cdr l) s))))))
```

L'inconvénient est que la liste `l` est parcourue deux fois lors de l'évaluation de `(count l s)`.

Un "remède" aussi naïf que catastrophique consiste en l'utilisation du schéma habituel :[69]

```
(define count1
  (lambda (l s)
    (if (null? l)
        (list 0 0)
        (if (eq? (car l) s)
            (list (1+ (car (count1 (cdr l) s)))
                  (cadr (count1 (cdr l) s)))
            (list (car (count1 (cdr l) s))
                  (1+ (cadr (count1 (cdr l) s))))))))
```

L'inefficacité est catastrophique, parce que chaque appel avec `l` non vide donne lieu à deux appels avec `(cdr l)` : le temps est donc exponentiel en la longueur de la liste. On peut remédier à cela simplement, par la séparation fonctionnelle :

```
(define c2
  (lambda (l s c)
    (if (null? l)
        c
        (if (eq? (car l) s)
            (c2 (cdr l)
                s
                (lambda (u)
                  (c (list (1+ (car u)) (cadr u)))))
            (c2 (cdr l)
                s
                (lambda (u)
                  (c (list (car u) (1+ (cadr u))))))))))

(define id (lambda (v) v))

(define count2 (lambda (l s) ((c2 l s id) '(0 0))))
```

[69] Notons à cette occasion les fonctions primitives `1+` et `-1+`; les expressions `(1+ a)` et `(-1+ a)` sont équivalentes aux expressions `(+ a 1)` et `(- a 1)`, respectivement.

Si ℓ est la longueur de la liste `l`, les temps d'exécution de `(count0 l)` et de `(count2 l)` sont proportionnels à ℓ. Celui de `(count1 l)` est proportionnel à 2^ℓ. Une bonne compréhension de ce qui précède passe par une spécification claire de la fonction auxiliaire `c2` :

> *Si* [[`s`]] *et* [[`l`]] *sont respectivement un objet et une liste d'objets,*
> *si* [[`c`]] *est une fonction qui à toute liste de deux nombres* $(n\ m)$
> *associe la liste de deux nombres* $(n{+}c_a\ \ m{+}c_d)$,
> *alors* [[`(c2 l s c)`]] *est la fonction qui à toute liste de deux nombres* $(n\ m)$
> *associe la liste* $(n{+}c_a{+}\ell_a\ \ m{+}c_d{+}\ell_d)$,
> *où* ℓ_a *est le nombre d'éléments de* [[`l`]] *égaux à* [[`s`]] *et*
> *où* ℓ_d *est le nombre d'éléments de* [[`l`]] *distincts de* [[`s`]].

On verra d'autres solutions pour ce problème au chapitre suivant.

5.7.2 Application : le produit d'une liste de nombres

Un autre problème élémentaire donnant lieu à l'application de la technique de séparation fonctionnelle est celui du produit d'une liste de nombres. Le schéma habituel donne la solution suivante :

```
(define prodlist0
  (lambda (l)
    (if (null? l) 1 (* (car l) (prodlist0 (cdr l))))))
```

Si un des éléments de `l` est nul, le résultat est 0 ... et toutes les multiplications ont été effectuées en vain. Un raffinement simple consiste à s'arrêter dès qu'un 0 est rencontré :

```
(define prodlist1
  (lambda (l)
    (cond ((null? l) 1)
          ((zero? (car l)) 0)
          (else (* (car l) (prodlist1 (cdr l)))))))
```

Si cependant le facteur 0 ne survient qu'en fin de liste, on aura quand même effectué de nombreuses multiplications inutiles. En fait, la question est, si un facteur est nul, comment éviter *toutes* les multiplications ? La séparation fonctionnelle permet de résoudre ce problème simplement :

```
(define pl2_c
  (lambda (l c)
    (cond ((null? l) c)
          ((zero? (car l)) (lambda (v) 0))
          (else (pl2_c (cdr l)
                       (lambda (u) (* (car l) (c u))))))))

(define prodlist2 (lambda (l) ((pl2_c l id) 1)))
```

Ici aussi, une bonne compréhension du comportement de la fonction principale passe par une spécification claire de la fonction auxiliaire `pl2_c` :

> *Si* [[`l`]] *est une liste de nombres, si* k_c *est un nombre*
> *et si* [[`c`]] *est la fonction qui à tout nombre* n *associe le produit* $k_c n$*,*
> *alors la fonction* [[`(pl2_c l c)`]] *associe à tout* n *le produit* $\ell k_c n$*,*
> *où* ℓ *est le produit des éléments de* [[`l`]]*.*

On verra une autre solution au chapitre 7, basée sur un principe différent.

6 Conception de programme

Au terme de ces premiers chapitres, nous avons rencontré les principaux mécanismes de SCHEME. Ceux-ci suffisent à résoudre un grand nombre de problèmes. Cependant, sauf peut-être indirectement, par le biais d'exemples, nous n'avons pas abordé le point essentiel de l'apprentissage de la programmation, qui est *Comment concevoir un programme ?*. Dans ce paragraphe, nous donnons quelques indications.

6.1 Première étude

6.1.1 L'énoncé

Pour fixer les idées, partons d'un problème très simple :

> *Ecrire une fonction* `cube_sum_square` *qui,*
> *à toute liste de nombres, associe*
> *le cube de la somme des carrés de ses éléments.*

L'écriture d'une fonction `f` peut nécessiter le recours à des fonctions auxiliaires, qui doivent alors être spécifiées.

6.1.2 Analyse, structuration, solution

Bien étudier l'énoncé fourni, ainsi que soigner la rédaction des énoncés spécifiant les fonctions auxiliaires éventuelles, n'est jamais une perte de temps. Dans notre exemple, le fragment

> ... le cube de la somme des carrés ...

suscite "naturellement" l'idée de recourir à des fonctions auxiliaires `cube` et `sum_square`, dont les spécifications sont les suivantes :

> *Ecrire une fonction* `cube` *qui à tout nombre*
> *associe le cube de ce nombre.*
>
> *Ecrire une fonction* `sum_square` *qui, à toute liste de nombres,*
> *associe la somme des carrés de ses éléments.*

Vu le caractère élémentaire du problème posé, donnons sans plus attendre le code de la fonction principale et des deux fonctions auxiliaires :

```
(define cube_sum_square
  (lambda (l)
    (cube (sum_square l))))

(define cube
```

```
  (lambda (x) (* x x x)))

(define sum_square
  (lambda (l)
    (if (null? l)
        0
        (+ (* (car l) (car l))
           (sum_square (cdr l))))))
```

6.1.3 Variantes

Si "naturelle" que soit cette solution, on peut toujours imaginer d'autres solutions, correspondant à d'autres décisions concernant le choix des fonctions auxiliaires, ou à l'utilisation d'autres algorithmes. Il est intéressant ici de considérer explicitement d'autres politiques en ce qui concerne le choix des fonctions auxiliaires. Observons d'abord que, *a priori*, la décision de créer une fonction auxiliaire `cube` ne s'imposait pas vraiment, parce que son code est vraiment "trop simple". Soulignons quand même un fait important : *définir* une fonction est une chose, la *nommer* en est une autre. Il aurait été parfaitement raisonnable de renoncer à nommer la fonction cube, et donc d'utiliser une fonction anonyme. La solution serait alors

```
(define cube_sum_square
  (lambda (l)
    ((lambda (x) (* x x x)) (sum_square l))))

(define sum_square ...
```

Par contre, renoncer aussi à cette fonction anonyme et donc écrire

```
(define cube_sum_square
  (lambda (l)
    (* (sum_square l) (sum_square l) (sum_square l))))
```

aurait été un gaspillage de ressources, conduisant à calculer la valeur de l'expression `(sum_square l)` trois fois. Le même raisonnement ne s'applique pas à l'élévation au carré présente à l'avant-dernière ligne du code de `sum_square` car l'évaluation de `(car l)` est très peu coûteuse.[70]

Remarque. Lors de l'application de la fonction [[`cube_sum_square`]] définie par

```
(define cube_sum_square
  (lambda (l)
    ((lambda (x) (* x x x)) (sum_square l))))
```

[70] Remplacer `(* (car l) (car l))` par `((lambda (x) (* x x)) (car l))` ne serait donc pas utile.

à un argument approprié [[l]], on doit évaluer d'abord [[(sum_square l)]], créer un environnement étendu par une liaison de x à cette valeur, puis évaluer (* x x x) dans cet environnement étendu. Cela correspond à dire, "soit v la valeur de (sum_square l); on renvoie la valeur de (* x x x), où [[x]] = v". Il se fait que SCHEME permet de traduire littéralement ce procédé :

```
(define cube_sum_square
  (lambda (l)
    (let ((x (sum_square l))) (* x x x))))
```

La forme spéciale let sera vue au paragraphe 9.1.

6.1.4 Eliminer une fonction auxiliaire ?

On pourrait aussi envisager d'éliminer le nom sum_square, voire la fonction elle-même. Notons d'emblée que, contrairement à cube, la fonction sum_square a été définie récursivement ... ce qui impose de lui donner un nom. Si on souhaite "cacher" ce nom, il faut utiliser une technique particulière, introduite au chapitre suivant.[71] Plus radicalement, le programmeur aurait pu envisager de programmer directement cube_sum_square, sans recourir à des fonctions auxiliaires (nommées ou anonymes), et en particulier sans utiliser sum_square. L'idée est *a priori* défendable, mais sera en principe rejetée après la courte étude suivante. Pour exprimer

$$[[\texttt{(cube\_sum\_square l)}]] \;=\; (x_0^2 + x_1^2 + \cdots + x_n^2)^3$$

en termes de

$$[[\texttt{(cube\_sum\_square (cdr l))}]] \;=\; (x_1^2 + \cdots + x_n^2)^3\,,$$

il faut utiliser la règle

$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3\,,$$

où l'on constate que b est précisément [[(sum_square (cdr l))]], ce qui montre que la fonction sum_square est indispensable.

6.1.5 Généralisation et réutilisation

Le fait que le recours à une fonction auxiliaire ait été reconnu nécessaire, comme dans le cas de sum_square, ne signifie pas obligatoirement que la fonction doit être programmée immédiatement. Il convient de se demander d'abord si cette fonction n'apparaît pas "naturellement" comme un cas particulier d'une fonction plus générale. Si c'est le cas, mieux vaut programmer la fonction plus générale; la probabilité que l'on puisse réutiliser

[71] Supposons que la fonction principale cube_sum_square ne soit qu'un petit fragment d'un gros logiciel, développé en équipe. Il faudra alors que chaque programmeur obtienne du chef d'équipe l'autorisation de nommer toute fonction autre que la fonction principale qu'il est chargé de construire, sinon il y aurait risque d'interférence avec les fonctions programmées et nommées par d'autres membres de l'équipe. En pratique, le programmeur "cachera" les noms des fonctions auxiliaires qu'il aura jugé opportun de développer.

une fonction auxiliaire, dans un contexte différent de celui qui a amené sa spécification et sa programmation, est d'autant plus élevée que cette fonction est générale. Dans le cas présent, il est naturel de généraliser la fonction

$$(x_1, \ldots, x_n) \mapsto \sum_{i=1}^{n} x_i^2$$

en la fonction

$$(f, (x_1, \ldots, x_n)) \mapsto \sum_{i=1}^{n} f(x_i)\,,$$

ce qui donne lieu au code suivant :

```
(define cube_sum_square
  (lambda (l) (cube (+_map square l))))

(define cube (lambda (x) (* x x x)))

(define square (lambda (x) (* x x)))

(define +_map
  (lambda (f l)
    (if (null? l)
        0
        (+ (f (car l))
           (+_map f (cdr l))))))
```

On peut aller plus loin dans la voie de la généralisation; en effet, l'opérateur somme apparaît ici comme la généralisation à un nombre quelconque d'arguments de l'opérateur binaire d'addition. Tout opérateur binaire admettant un élément neutre est susceptible d'être étendu de la sorte; nous connaissons déjà la multiplication, dont le neutre est `1`, et la concaténation, dont le neutre est `()`. (Rappelons que la fonction prédéfinie `append` calcule la concaténation d'un nombre quelconque de listes.) Cela justifie que l'on généralise la fonction

$$(f, (x_1, \ldots, x_n)) \mapsto \sum_{i=1}^{n} f(x_i)\,,$$

en la fonction

$$(\omega, \nu, (x_1, \ldots, x_n)) \mapsto \Omega_{i=1}^{n} f(x_i)\,.$$

Dans cette notation, ω est un opérateur binaire et Ω est la généralisation de ω définie comme suit :

$$\begin{aligned}
&\Omega(()) = \nu\,,\\
&\Omega((x_1)) = x_1\,,\\
&\Omega((x_1, x_2)) = x_1\ \omega\ x_2\,,\\
&\Omega((x_1, x_2, x_3)) = x_1\ \omega\ (x_2\ \omega\ x_3)\,,\\
&\ldots
\end{aligned}$$

Avec cette nouvelle généralisation, la solution du problème original devient :

```
(define cube_sum_square
  (lambda (l) (cube (gen_map + 0 square l))))

(define cube (lambda (x) (* x x x)))

(define square (lambda (x) (* x x)))

(define gen_map
  (lambda (omega nu f l)
    (if (null? l)
        nu
        (omega (f (car l))
               (gen_map omega nu f (cdr l))))))
```

On peut voir tout de suite l'intérêt de la généralisation. Supposons que l'on veuille définir la fonction qui à toute liste de nombres associe le carré du produit de ces nombres augmentés de 5; la réponse est maintenant immédiate :

```
(define square_product_add5
  (lambda (l) (square (gen_map * 1 add5 l))))

(define add5 (lambda (n) (+ n 5)))
```

On peut aussi définir aisément les fonctions `reverse_concat_duplic` et `duplic_concat_reverse` telles que, par exemple

```
(reverse_concat_duplic '((a b c) (1 2) (x y)))
  ==>  (y x y x 2 1 2 1 c b a c b a)

(duplic_concat_reverse '((a b c) (1 2) (x y)))
  ==>  (c b a 2 1 y x c b a 2 1 y x)
```

Le code est

```
(define reverse_concat_duplic
  (lambda (l) (reverse (gen_map append '() duplic l))))

(define duplic_concat_reverse
  (lambda (l) (duplic (gen_map append '() reverse l))))

(define duplic (lambda (l) (append l l)))
```

Il est fréquent d'utiliser `gen_map` avec comme premier argument `append`; cela justifie la définition

```
(define append_map
  (lambda (f l) (gen_map append '() f l)))
```

On peut aussi définir `append_map` directement :

```
(define append_map
  (lambda (f l)
    (if (null? l)
        '()
        (append (f (car l)) (append_map f (cdr l))))))
```

ou encore en utilisant `apply` :

```
(define append_map
  (lambda (f l) (apply append (map f l))))
```

Cette fonction sera réutilisée dans la suite; notons aussi la variante

```
(define union_map
  (lambda (f l)
    (if (null? l)
        '()
        (union (f (car l)) (union_map f (cdr l))))))
```

D'autres cas particuliers sont fréquents. Notons d'abord que les expressions `(gen_map cons '() f l)` et `(map f l)` ont même valeur; on a déjà mentionné `+_map`, et on définit aussi `*_map` de telle sorte que `(gen_map * 1 f l)` et `(*_map f l)` aient même valeur.

Deux cas intéressants sont `and_map` et `or_map` mais, `and` et `or` étant des formes spéciales et non des fonctions susceptibles d'être passées en arguments, on doit définir ces opérateurs directement :

```
(define and_map
  (lambda (f l)
    (if (null? l)
        #t
        (and (f (car l)) (and_map f (cdr l))))))

(define or_map
  (lambda (f l)
    (if (null? l)
        #f
        (or (f (car l)) (or_map f (cdr l))))))
```

ou encore

```
(define and_map
  (lambda (f l)
    (or (null? l)
        (and (f (car l)) (and_map f (cdr l))))))
```

```
(define or_map
  (lambda (f l)
    (and (not (null? l))
         (or (f (car l)) (or_map f (cdr l))))))
```

6.1.6 Filtrage et transformation

La fonction `gen_map` est un opérateur très général pour traiter les listes élément par élément. Il est parfois nécessaire de filtrer une liste avant l'application de `gen_map`, pour éliminer les éléments auxquels le traitement ne peut pas ou ne doit pas s'appliquer. On utilise dans ce but la fonction `filter` définie comme suit :

```
(define filter
  (lambda (p? l)
    (cond ((null? l) '())
          ((p? (car l)) (cons (car l) (filter p? (cdr l))))
          (else (filter p? (cdr l))))))
```

Le prédicat unaire `p?` est le filtre; les éléments de `l` qui ne le vérifient pas sont omis. On peut alors intégrer le filtrage à la fonction `gen_map` :

```
(define gen_map_filter
  (lambda (omega nu f p? l)
    (gen_map omega nu f (filter p? l))))
```

ou encore :

```
(define gen_map_filter
  (lambda (omega nu f p? l)
    (cond ((null? l) nu)
          ((p? (car l))
           (omega (f (car l))
                  (gen_map_filter omega nu f p? (cdr l))))
          (else (gen_map_filter omega nu f p? (cdr l))))))
```

Remarque. En utilisant déjà la construction `let` évoquée plus haut, on a aussi :

```
(define gen_map_filter
  (lambda (omega nu f p? l)
    (if (null? l)
        nu
        (let ((rec (gen_map_filter omega nu f p? (cdr l))))
          (if (p? (car l))
              (omega (f (car l)) rec)
              rec)))))
```

D'autres généralisations sont possibles. Le filtrage introduit ici est un préfiltrage, puisque les éléments sont testés avant l'application de `f`. On pourrait ajouter un postfiltrage, qui s'exercerait après cette application. A l'opposé, une particularisation intéressante est `map_filter` :

```
(define map_filter
  (lambda (f p? l)
    (gen_map_filter cons '() f p? l)))
```

que l'on pouvait aussi définir directement :

```
(define map_filter
  (lambda (f p? l)
    (cond ((null? l) '())
          ((p? (car l))
           (cons (f (car l)) (map_filter f p? (cdr l))))
          (else (map_filter f p? (cdr l))))))
```

Remarque. Les expressions `(map_filter id p? l)` et `(filter p? l)` sont équivalentes si [[`id`]] est la fonction identique.

Remarque. La facilité avec laquelle les procédures écrites en SCHEME se généralisent et se particularisent est un atout important du langage.

6.2 Deuxième étude

6.2.1 L'énoncé

On a vu au paragraphe 2.7.6 que la valeur renvoyée par une procédure, ainsi que les arguments d'une procédure, pouvaient eux-mêmes être des procédures. L'exemple classique est celui de la procédure `compose`, qui prend comme arguments deux procédures et renvoie leur composition. La récursivité permet de définir d'autres procédures d'ordre supérieur. On peut notamment généraliser `compose` en une fonction `compose_list`, spécifiée comme suit :

> *Ecrire une fonction* `compose_list` *qui, à toute liste de fonctions unaires composables, associe leur composée.*[72]

Par exemple, [[`(compose_list (list f g h))`]] sera [[`f`]] ∘ [[`g`]] ∘ [[`h`]], la fonction résultant de la composition de [[`f`]], [[`g`]] et [[`h`]].

Remarque. Rappelons que $(f \circ g)(x)$ est par définition $f(g(x))$; on applique d'abord g, puis f.

[72] Rappelons que l'opérateur de composition de fonctions est associatif mais pas commutatif. La fonction identité est neutre pour cet opérateur.

6.2.2 Solution directe, solution par réutilisation

Le schéma de récursion superficielle sur les listes fournit immédiatement une solution :

```
(define compose_list
  (lambda (f_list)
    (if (null? f_list)
        (lambda (x) x)
        (compose (car f_list) (compose_list (cdr f_list))))))
```

On peut éviter le recours explicite à `compose` :

```
(define compose_list
  (lambda (f_list)
    (if (null? f_list)
        (lambda (x) x)
        (lambda (x)
          ((car f_list) ((compose_list (cdr f_list)) x))))))
```

On a par exemple

```
((compose_list (list car cdr cdr)) '(1 2 3 4 5))  ==>  3
```

On peut aussi noter que le problème posé, quoiqu'entièrement différent du problème posé au paragraphe précédent, se résout immédiatement grâce à la fonction auxiliaire très générale introduite dans ce paragraphe précédent :

```
(define compose
  (lambda (f g) (lambda (x) (f (g x)))))

(define id (lambda (x) x))

(define compose_list
  (lambda (f_list)
    (gen_map compose
             (lambda (x) x)
             (lambda (x) x)
             f_list)))
```

6.2.3 L'itérateur

Un cas particulier intéressant est celui où toutes les fonctions à composer sont égales. On appelle $n$ième itérée de la fonction f de D dans D la composée de n fonctions égales à f.

> *Ecrire une fonction* `iter` *tel que pour tout naturel* `n`,
> `(iter n)` *soit la fonction qui à toute fonction* `f`
> *composable avec elle-même associe la* `n`*ième itérée de* `f`.

La solution est immédiate. On peut écrire

```
(define iter
  (lambda (n)
    (lambda (f)
      (if (zero? n)
          (lambda (x) x)
          (compose f
                   ((iter (- n 1)) f))))))
```

Remarque. Il pourrait paraître abusif (ou prétentieux ...) de qualifier cette solution d'immédiate. Avec un peu d'habitude, elle l'est néanmoins, dans la mesure où elle se commente très simplement. Dans le texte suivant, chaque ligne correspond à une ligne du programme précédent.

> *On définit* `iter`,
> *une fonction à un argument* `n`;
> *la fonction* `(iter n)` *associe à* `f`
> *si* `n` *vaut 0,*
> *la fonction identité,*
> *sinon la composée de* `f`
> *et de* $\texttt{f}^{\texttt{n}-1}$.

Une variante de cette solution est

```
(define iter
  (lambda (n)
    (lambda (f)
      (lambda (x)
        (if (zero? n)
            x
            (f (((iter (- n 1)) f) x)))))))
```

On peut aussi réutiliser une fonction précédente :

```
(define iter
  (lambda (n)
    (lambda (f) (compose_list (n_list n f)))))

(define n_list
  (lambda (n x)
    (if (= n 0) '() (cons x (n_list (- n 1) x)))))
```

Voici quelques exemples d'utilisation :

```
(define add1 (lambda (x) (+ x 1)))
(define add7 ((iter 7) add1))
(add7 8)  ==>  15
```

```
(define square (lambda (x) (* x x)))
(define power8 ((iter 3) square))
(power8 2)  ==>  256
```

6.3 Troisième étude

Le lecteur est sans doute déjà convaincu de l'intérêt des schémas récursifs, dont une surprenante variété de programmes sont de simples instances. On soupçonne néanmoins, à juste titre d'ailleurs, que de nombreux problèmes demandent une démarche plus créative qu'une simple instantiation de schéma. Nous verrons d'ailleurs dans la suite du livre quelques exemples de tels problèmes. Toutefois, c'est une erreur méthodologique de croire trop tôt qu'un problème ne peut être résolu simplement, ce que nous illustrons par deux exemples classiques.

6.3.1 Deux énoncés classiques

Ecrire une fonction `subsets`
qui calcule la liste des sous-ensembles d'un ensemble donné.

Ecrire une fonction `partitions`
qui calcule la liste des partitions d'un ensemble donné.

6.3.2 Solution du premier problème

Il n'est pas très difficile d'énumérer la liste des sous-ensembles d'un ensemble donné. On a par exemple, pour l'ensemble $\{a, b, c\}$, les sous-ensembles suivants :

$$\{\}\,,\ \{a\}\,,\ \{b\}\,,\ \{c\}\,,\ \{a, b\}\,,\ \{a, c\}\,,\ \{b, c\}\,,\ \{a, b, c\}\,.$$

Sur base de cet exemple, et guidé par l'ordre (pourtant non imposé) dans lequel les sous-ensembles ont été énumérés, on pourrait juger opportun de générer séparément les sous-ensembles de 0, de 1, de 2 et de 3 éléments. C'est parfaitement possible, mais il y a mieux — c'est-à-dire plus simple — à faire : envisager l'utilisation directe d'un schéma récursif. On représentera naturellement un ensemble par une liste sans répétition.[73] Pour ce faire, on se demande comment la liste des sous-ensembles de $\{a, b, c\}$, par exemple, pourrait être construite au départ de la liste des sous-ensembles de $\{b, c\}$, c'est-à-dire

$$\{\}\,,\ \{b\}\,,\ \{c\}\,,\ \{b, c\}\,.$$

On observe d'abord que les sous-ensembles de $\{b, c\}$ sont aussi des sous-ensembles de $\{a, b, c\}$, mais que la réciproque n'est pas vraie; les sous-ensembles manquants sont

$$\{a\}\,,\ \{a, b\}\,,\ \{a, c\}\,,\ \{a, b, c\}\,.$$

[73] Les ensembles $\{a, b\}$ et $\{b, a\}$ sont en fait le même ensemble, tandis que les deux listes `(a b)` et `(b a)` sont distinctes.

On observe ensuite que les sous-ensembles nouveaux sont les anciens dans lesquels on a inséré l'élément nouveau a.

Soulignons avec insistance que ces deux observations elles-mêmes sont des évidences; le seul moyen de "passer à côté" est d'omettre d'envisager l'usage des schémas récursifs. Notons enfin que le cas de base est évident : l'ensemble vide admet un seul sous-ensemble, lui-même. On obtient immédiatement le code suivant :

```
(define subsets
  (lambda (e)
    (if (null? e)
        '(())
        (append (subsets (cdr e))
                (insert_in_all (car e) (subsets (cdr e)))))))

(define insert_in_all
  (lambda (x le)
    (if (null? le)
        '()
        (cons (cons x (car le))
              (insert_in_all x (cdr le))))))
```

La fonction auxiliaire `insert_in_all` prend comme arguments un objet `x` et une liste de listes `ll`. Elle renvoie une liste de listes, dont les éléments sont ceux de `ll` préfixés de `x`.

Cette "solution" est correcte mais très inefficace : l'appel `(subsets e)`, quand `e` n'est pas vide, provoque deux appels récursifs à `(subsets (cdr e))`. On y remédie comme suit :

```
(define subsets
  (lambda (e)
    (if (null? e)
        '(())
        ((lambda (le)
           (append le (insert_in_all (car e) le)))
         (subsets (cdr e))))))
```

Pour évaluer `(subsets e)`, on évalue une fois `(subsets (cdr e))` et on lui applique la fonction anonyme définie par la λ-forme. On pourrait choisir de donner un nom, par exemple `expand`, à cette fonction, à condition d'ajouter en argument supplémentaire l'élément nouveau, valeur de `(car e)`. On obtient alors la variante suivante :

```
(define subsets
  (lambda (e)
    (if (null? e)
        '(())
        (expand (car e) (subsets (cdr e))))))
```

```
(define expand
  (lambda (x le)
    (append le (insert_in_all x le))))
```

Remarque. Cette technique consistant à introduire une fonction auxiliaire, souvent anonyme, dans le seul but d'utiliser plus d'une fois un résultat intermédiaire, est d'usage très fréquent. Comme nous l'avons déjà signalé au paragraphe 6.1.3, la forme spéciale de mot-clef `let` a été introduite à cet usage dans le langage SCHEME; dans le cas présent, on aurait pu écrire

```
(define subsets
  (lambda (e)
    (if (null? e)
        '(())
        (let ((le (subsets (cdr e))))
           (append le (insert_in_all (car e) le))))))
```

6.3.3 Solution du second problème

La fonction `partitions` est *a priori* plus difficile à écrire parce que la structure de l'ensemble des partitions d'un ensemble donné est moins apparente que celle de l'ensemble des sous-ensembles. Rappelons qu'une partition d'un ensemble E est simplement un partage de cet ensemble, c'est-à-dire une famille de sous-ensembles non vides de E, disjoints deux à deux et dont la réunion forme E. Les cinq partitions de $\{a, b, c\}$ sont

$$\{\{a\},\{b\},\{c\}\},\ \{\{a\},\{b,c\}\},\ \{\{b\},\{a,c\}\},\ \{\{c\},\{a,b\}\},\ \{\{a,b,c\}\}.$$

La bonne question est, à nouveau, comment obtenir les partitions de $\{a, b, c\}$ au départ de celles de $\{b, c\}$. Il faut *penser* à se poser cette question;[74] la résoudre est simple, en principe du moins. Les partitions de $\{b, c\}$ sont

$$\{\{b\},\{c\}\},\ \{\{b,c\}\}.$$

On observe qu'une partition de $\{a, b, c\}$ est obtenue au départ d'une partition de $\{b, c\}$ selon deux techniques :

1. En insérant le singleton $\{a\}$ comme partie supplémentaire;
   $\{\{b\},\{c\}\}$ donne $\{\{a\},\{b\},\{c\}\}$;
   $\{\{b,c\}\}$ donne $\{\{a\},\{b,c\}\}$.

2. En insérant l'élément a dans une partie existante;
   $\{\{b\},\{c\}\}$ donne $\{\{a,b\},\{c\}\}$ et $\{\{b\},\{a,c\}\}$;
   $\{\{b,c\}\}$ donne $\{\{a,b,c\}\}$.

[74] Du moins au début; pour l'habitué, se poser cette question est un réflexe !

On obtient aisément le code suivant :

```
(define partitions
  (lambda (e)
    (if (null? e)
        '(())
        (append (procede_1 (car e) (partitions (cdr e)))
                (procede_2 (car e) (partitions (cdr e)))))))
```

qui, pour la même raison d'efficacité que précédemment, est remplacé par

```
(define partitions
  (lambda (e)
    (if (null? e)
        '(())
        ((lambda (lp)
           (append (procede_1 (car e) lp)
                   (procede_2 (car e) lp)))
         (partitions (cdr e))))))
```

Observons que l'ensemble vide admet une seule partition; cette partition ne comporte aucune partie et est donc l'ensemble vide lui-même.[75]

La première fonction auxiliaire est immédiate car il suffit de réutiliser la fonction `insert_in_all` introduite plus haut; on a

```
(define procede_1
  (lambda (x lp)
    (insert_in_all (list x) lp)))
```

Pour la seconde fonction auxiliaire, il y a aussi possibilité de réutilisation. En effet, on doit, pour chaque partition de `lp`, appliquer le procédé 2, ce qui donne une liste de partitions, puis concaténer toutes ces listes. On a donc

```
(define procede_2
  (lambda (x lp)
    (append_map (lambda (p) (split x p)) lp)))
```

La fonction `append_map` apparaît dans la première étude. La fonction `split` réalise le procédé 2 proprement dit, pour une partition. Comme toujours lorsque l'on spécifie une fonction auxiliaire, il convient de le faire de la manière la plus générale possible. Le second argument ne sera donc pas nécessairement une partition, mais une quelconque liste de listes. On aura non seulement

```
(split 'a '((b) (c))) ==> (((a b) (c)) ((b) (a c)))
(split 'a '((b c))) ==> (((a b c)))
```

[75] La définition d'une partition spécifie que les parties éventuelles qui la composent sont des ensembles non vides, mais ne spécifie pas qu'il doit y avoir au moins une partie.

mais aussi, par exemple,

```
(split '2 '((1 2) () (3)))
==>  (((2 1 2) () (3)) ((1 2) (2) (3)) ((1 2) () (2 3)))
```

La construction de la fonction `split` n'est pas immédiate ... mais elle le devient si nous employons la tactique habituelle : comment obtient-on `(split x ll)` à partir de `(split x (cdr ll))` ? Un exemple est toujours éclairant :

```
(split '0 '(() (3)))       ;; (split x (cdr ll))
==>  (((0) (3)) (() (0 3)))
(split '0 '((1 2) () (3))) ;; (split x ll)
==>  (((0 1 2) () (3)) ((1 2) (0) (3)) ((1 2) () (0 3)))
```

Le premier élément du résultat est à créer de toutes pièces; c'est [[`(cons (cons x (car ll)) (cdr ll))`]]. Le reste s'obtient en remplaçant dans [[`(split x (cdr ll))`]] (liste de listes de listes) chaque élément [[`ss`]] (liste de listes) par [[`(cons (car ll) ss)`]]. Le code est maintenant immédiat :

```
(define split
  (lambda (x ll)
    (if (null? ll)
        '()
        (cons (cons (cons x (car ll)) (cdr ll))
              (map (lambda (ss) (cons (car ll) ss))
                   (split x (cdr ll)))))))
```

On peut à présent utiliser la fonction `partition` :

```
(partitions '(a b c))  ==>
(((a) (b) (c)) ((a) (b c)) ((a b) (c)) ((b) (a c)) ((a b c)))
```

Réduire le cas d'une liste non vide `l` au cas de `(cdr l)` est l'essentiel du travail d'application du schéma de récursion, mais résoudre le cas de la liste vide est tout aussi important. "Si la base s'effondre, le sommet ne restera pas inébranlable". Ce morceau de sagesse populaire est d'application ici; le lecteur peut s'en rendre compte en essayant la "variante" de `partitions` dans laquelle le résultat pour le cas vide — l'expression `'(())` — aurait été remplacé par l'expression `'()`. Une erreur de ce type peut anéantir tout un développement.

6.3.4 Approche descendante, approche ascendante

Nous avons fait usage du style de développement "de haut en bas" (ou "top-down"). On a écrit d'abord la fonction principale `partitions`; cela nous a amené à spécifier et à utiliser deux sous-fonctions (fonctions auxiliaires), nommées `procede_1` et `procede_2`. On a ensuite écrit ces fonctions, ce qui nous a amené à spécifier et à utiliser d'autres

sous-sous-fonctions. Nous privilégions ce style dans tout l'ouvrage, ce qui revient à écrire une fonction principale avant les fonctions auxiliaires que la fonction principale utilise. L'intérêt du style "top-down" est clair : les fonctions auxiliaires sont introduites en cas de besoin seulement, et spécifiées en fonction de ces besoins.

Le style opposé, "de bas en haut" (ou "bottom-up") expose au risque d'écrire des fonctions auxiliaires inutiles ou mal adaptées, sur lesquelles on devrait revenir ultérieurement, après avoir écrit des fonctions les utilisant. Ce style présente néanmoins deux avantages. Quand le développeur n'est pas directement influencé par un besoin précis, il écrira souvent des fonctions auxiliaires plus générales, voire toute une bibliothèque de fonctions auxiliaires qui seront réutilisées intensivement.

Le compromis que nous adoptons est le suivant. On utilise le style "top-down", mais en veillant à généraliser les fonctions auxiliaires et à bien les documenter, ce qui favorise la réutilisation de ces fonctions. Dans le programme `partition`, nous récupérons ainsi `append_map` et `insert_in_all`, qui avaient été spécifiées et écrites dans un autre contexte. Nous avons veillé aussi à fournir de `split` une version plus générale que strictement nécessaire (surtout au point de vue de sa spécification), de manière à favoriser une réutilisation éventuelle.

6.4 Quatrième étude

Les trois études que nous avons présentées sont éclairantes mais peu représentatives des problèmes réels, c'est-à-dire des problèmes auxquels l'informaticien est généralement confronté. Considérons à présent un problème réel, induit par le petit discours suivant.

> *Je souhaite emprunter de l'argent, pour acheter une maison et une voiture. J'ai contacté divers organismes prêteurs qui m'ont proposé différentes combinaisons de délais et de taux; d'autres n'annoncent pas de taux mais directement le montant de la mensualité. Dans les rares cas où le taux et la mensualité étaient annoncés, j'ai recalculé la mensualité moi-même ... et abouti à un montant inférieur à celui exigé. Curieusement, la différence tend à être plus importante pour les taux "voiture" que pour les taux "maisons". D'où viennent ces divergences systématiquement en ma défaveur ? Comment puis-je vérifier, et comparer différentes propositions ?*

6.4.1 Clarifier le problème

L'informaticien doit d'abord transformer ce discours[76] en l'énoncé d'un problème clair. Pour cela, il doit d'abord connaître les différentes méthodes de calcul; il pourra ensuite les programmer, puis créer des programmes de comparaison permettant de confronter les conditions de deux organismes utilisant des méthodes différentes. Le point de départ de notre démarche est un rappel succinct de la théorie classique des intérêts composés.

[76] qui a été effectivement tenu à l'auteur, et qui a donné lieu au petit travail décrit dans la suite de ce paragraphe.

La notion de taux d'intérêt est centrale entre prêteurs et emprunteurs. Le système habituellement admis est simple. L'emprunteur reçoit du prêteur une somme S et s'engage à la rembourser, accrue des intérêts, sous forme de "périodicités" (annuités ou mensualités) en général constantes. On fixe un taux t (souvent exprimé en pourcentage; 0.5 % par mois signifie un taux mensuel de 0.005) et un nombre de périodes n. Supposons pour fixer les idées que la période est le mois et que le taux fixé est mensuel. Déterminer le montant M de la mensualité constante n'est pas très difficile. On observe d'abord qu'une somme S, au terme d'un nombre n de mois, vaudra

$$S' \;=\; S(1+t)^n\,; \tag{IC1}$$

c'est la formule classique des intérêts composés, qui donne la somme à rembourser par l'emprunteur si le remboursement s'opère en un seul versement, à l'échéance. Dans le cas de remboursements mensuels constants, la formule devient

$$S' \;=\; M(1+t)^{n-1} + M(1+t)^{n-2} + \cdots + M(1+t) + M\,; \tag{IC2}$$

le $i$ème terme $M(1+t)^{n-i}$ représente la valeur à l'échéance (au terme du $n$ième mois) de la mensualité M payée au terme du $i$ème mois, et qui s'est donc valorisée pendant $(n-i)$ mois. On utilise la formule bien connue

$$\sum_{i=0}^{n-1} b^i = \frac{b^n-1}{b-1}\,,$$

où $b = 1+t$, pour effectuer la somme des valorisations des n mensualités et, par élimination de S' entre IC1 et IC2, on tire

$$S(1+t)^n = M\frac{(1+t)^n-1}{t} \tag{IC3}$$

ou encore

$$M = \frac{St(1+t)^n}{(1+t)^n-1} \tag{IC4}$$

qui permet le calcul de la mensualité constante. Le programme SCHEME correspondant à l'égalité IC4 est élémentaire; il s'écrit

```
(define periodicite
  (lambda (S t n)
    (/ (* S t (expt (+ 1 t) n))
       (- (expt (+ 1 t) n) 1))))
```

La valeur [[`(periodicite S t n)`]] donne le montant périodique constant à rembourser, pour les données `S`, `t` et `n`.[77]

[77] Dans certains systèmes anciens, l'identificateur `t` est assimilé au booléen `#t`; il est alors nécessaire de choisir un autre nom de variable.

Remarque. Ce programme comporte un facteur d'inefficacité : il prévoit que l'expression $(1+t)^n$ sera calculée deux fois. Cela n'a rien de catastrophique, mais on pourrait y remédier en utilisant une fonction auxiliaire anonyme. On aurait alors la variante de gauche ou, en anticipant sur l'introduction de la forme spéciale `let`, celle de droite :

```
(define periodicite
  (lambda (S t n)
    ((lambda (aux)
       (/ (* S t aux)
          (- aux 1)))
     (expt (+ 1 t) n))))
```

```
(define periodicite
  (lambda (S t n)
    (let ((aux
           (expt (+ 1 t) n)))
      (/ (* S t aux)
         (- aux 1)))))
```

La fonction `periodicite` permet de calculer la mensualité en fonction de la somme à emprunter, du taux d'intérêt mensuel et de la durée du prêt en mois; elle ne permet pas de calculer directement, par exemple, le taux en fonction de la somme empruntée, de la mensualité et du nombre de mois. En pratique, l'emprunteur qui connaît la mensualité requise, ou sa capacité maximale de remboursement, peut se poser les trois questions suivantes :

Etant donné que ma capacité de remboursement mensuel est de M francs,

- à quel taux mensuel maximal t puis-je emprunter la somme S, remboursable en n mois ?
- quelle somme maximale puis-je emprunter au taux mensuel t, pour n mois ?
- en combien de mois minimum puis-je rembourser la somme S empruntée au taux mensuel t ?

Quoique ces questions ne couvrent pas l'intégralité du discours perplexe qui nous a été soumis, elles constituent un bon point de départ et notre premier objectif sera de les programmer.

6.4.2 Inversion d'une fonction réelle

Les trois fonctions à programmer sont visiblement les inverses de la fonction `periodicite` par rapport à chacun de ses trois arguments. On voit la nécessité de pouvoir inverser une fonction de plusieurs variables et il sera donc indiqué de spécifier ce sous-problème d'inversion puis d'en programmer la solution. Il sera clairement plus économique de poser le problème dans un cadre général; cela permettra de développer un seul programme, qui sera utilisé pour répondre aux trois questions ci-dessus, et peut-être à beaucoup d'autres.

Le problème de l'inversion d'une fonction réelle de plusieurs variables réelles suscite d'emblée deux questions : comment inverse-t-on une fonction et comment tient-on compte des arguments non concernés par l'inversion. Inverser la fonction

$$f : \mathbb{R}^3 \to \mathbb{R} : (x_1, x_2, x_3) \mapsto f(x_1, x_2, x_3)$$

par rapport à x_2 consiste à construire une fonction

$$g : \mathbb{R}^3 \to \mathbb{R} : (x_1, u, x_3) \mapsto g(x_1, u, x_3)$$

telle que

$$g(x_1, f(x_1, x_2, x_3), x_3) = x_2 \quad \text{et} \quad f(x_1, g(x_1, u, x_3), x_3) = u\,,$$

pour toutes valeurs adéquates de x_2 et u. Le problème est mathématiquement difficile, puisque l'inverse n'existe pas toujours, mais devient plus simple dans le cas où la fonction à inverser est continue et strictement monotone (croissante ou décroissante) par rapport à l'argument (ici x_2) sur lequel porte l'inversion. On va aussi admettre que, comme dans l'exemple qui nous occupe, tous les arguments sont strictement positifs.

On devrait en principe écrire un opérateur d'inversion distinct pour chaque nombre n d'arguments et pour chaque rang $i \in \{1, \ldots, n\}$ de l'argument sur lequel porte l'inversion, ce qui semble tout à fait impraticable. Rien ne nous empêche, d'une part, de permuter les arguments de sorte que l'argument x sur lequel porte l'inversion soit toujours le premier et, d'autre part, de regrouper les autres arguments en une liste ℓ. Supposons croissante en x la fonction $(x, \ell) \mapsto f(x, \ell)$. Comment déterminer la fonction inverse $(u, \ell) \mapsto g(u, \ell)$? La méthode naïve, mais raisonnablement efficace, consiste à calculer $g(u, \ell)$ par approximations successives : on crée une suite $(x_0, x_1, \ldots)$ de nombres telle que chaque terme approxime $g(u, \ell)$, l'approximation devenant satisfaisante à partir d'un certain rang. L'idée de base est que, si $f(x_n, \ell)$ excède u, alors x_n excède $g(u, \ell)$ et on choisira $x_{n+1} < x_n$; si u excède $f(x_n, \ell)$, on choisira $x_{n+1} > x_n$. Cette idée demande à être précisée. D'une part, il faut préciser la manière dont on choisit les x_i successifs et, d'autre part, il faut choisir une condition d'arrêt. La technique de la *bissection* consiste à maintenir un intervalle $[m, M]$ dans lequel la valeur recherchée se trouve, et à rétrécir cet intervalle à chaque itération. Au départ, l'intervalle est très grand, par exemple $m = 10^{-6}$ et $M = 10^7$. A chaque étape, on calcule la moyenne μ des bornes de l'intervalle puis la valeur $f(\mu, \ell)$. En fonction de cette valeur, on décide de s'arrêter (si $f(\mu, \ell) = u$)[78] ou de continuer soit avec l'intervalle $[m, \mu]$, soit avec l'intervalle $[\mu, M]$. Chaque étape a pour effet de réduire de moitié la longueur de l'intervalle et on peut s'arrêter dès que cette longueur devient suffisamment petite. On introduit une constante ε telle que l'arrêt survient quand la longueur de l'intervalle descend en dessous de ε.
Remarque. La moyenne la plus fréquemment utilisée est la moyenne arithmétique mais on a parfois intérêt, lorsque tous les nombres impliqués sont strictement positifs, à préférer la moyenne géométrique; c'est le cas ici.[79]

On définit d'abord les trois variables globales spécifiant respectivement la borne inférieure de l'intervalle initial, sa borne supérieure et le seuil de tolérance; on définit aussi une fonction calculant la moyenne (géométrique) de deux nombres :

[78] Tester une égalité de ce type n'a guère de sens; il serait préférable de tester une "proximité"; nous n'entrons pas ici dans des considérations du ressort de l'analyse numérique.

[79] Rappelons que la moyenne arithmétique de deux nombres a et b est $\frac{a+b}{2}$; si a et b sont strictement positifs, leur moyenne géométrique est $\sqrt{ab}$.

```
(define *min* 1.e-6)
(define *max* 1.e+7)
(define *eps* 1.e-12)
(define mu (lambda (a b) (sqrt (* a b))))
```

On peut alors programmer l'inversion proprement dite. Nous considérons ici le cas de fonctions croissantes :

```
(define inv+
  (lambda (f)
    (lambda (u l) (i+ f u l *min* *max* *eps*))))

(define i+
  (lambda (f u l x0 x1 eps)
    (cond ((prox x0 x1 eps) (mu x0 x1))
          ((prox (f (mu x0 x1) l) u eps) (mu x0 x1))
          ((< (f (mu x0 x1) l) u) (i+ f u l (mu x0 x1) x1 eps))
          (else (i+ f u l x0 (mu x0 x1) eps)))))

(define prox (lambda (a b eps) (< (abs (- a b)) eps)))
```

L'opérateur `inv+` prend comme argument une fonction `f` croissante en son premier argument et renvoie la fonction inverse. Elle utilise les fonctions auxiliaires `prox` et `i+`, qu'il convient de spécifier. Pour simplifier les écritures, on notera la valeur d'un objet SCHEME par le nom de cet objet écrit en italique; pour la valeur d'une combinaison, on utilisera la notation mathématique habituelle.[80] Le prédicat *prox* prend comme arguments deux réels a et b et un réel positif ε; il renvoie vrai si $|a - b| < \varepsilon$. En ce qui concerne i^+, on a

> *Si la fonction* $f : (\mathbb{R} \times \mathbb{R}^k) \to \mathbb{R}$ *est croissante en son premier argument et si l'unique solution* x *de l'équation* $f(x, \ell) = u$ *appartient à l'intervalle* $[x_0 : x_1]$, *alors* $i^+(f, u, \ell, x_0, x_1, \varepsilon)$ *est un nombre* x' *proche de* x, *au sens que* x' *ne s'écarte pas de* x *de plus de* $\varepsilon/2$ *ou que* $f(x', \ell)$ *ne s'écarte pas de* u *de plus de* ε.

On définit de manière analogue un opérateur `inv-` pour inverser les fonctions décroissantes, qui fera appel à l'opérateur auxiliaire `i-`; ce dernier ne diffère de `i+` que par la permutation des comparateurs `<` et `>` dans les conditions des clauses contenant les appels récursifs. Un dernier point intéressant consiste à modifier `i+` de manière à éviter l'évaluation multiple des expressions `(mu x0 x1)` et `(f (mu x0 x1) l)`. Nous utilisons la technique déjà employée pour la variante de la fonction `periodicite`; cela permet d'abord d'éviter l'évaluation multiple de `(mu x0 x1)` :

```
(define i+
  (lambda (f u l x0 x1 eps)
```

[80] On écrira donc, par exemple, $h(a, (x + y)/2)$ au lieu de [[`(h a (/ (+ x y) 2))`]].

```
    ((lambda (aux1)
       (cond ((prox x0 x1 eps) aux1)
             ((prox (f aux1 l) u eps) aux1)
             ((< (f aux1 l) u) (i+ f u l aux1 x1 eps))
             (else (i+ f u l x0 aux1 eps))))
     (mu x0 x1))))
```

On peut réutiliser la même technique pour éviter la double évaluation de l'expression `(f aux1 l)`; on obtient ainsi :

```
(define i+
  (lambda (f u l x0 x1 eps)
    ((lambda (aux1)
       ((lambda (aux2)
          (cond ((prox x0 x1 eps) aux1)
                ((prox aux2 u eps) aux1)
                ((< aux2 u) (i+ f u l aux1 x1 eps))
                (else (i+ f u l x0 aux1 eps))))
        (f aux1 l)))
     (mu x0 x1))))
```

Ici aussi, la construction `let` pourra être utilisée.

6.4.3 Solution du problème simplifié

Nous pouvons maintenant répondre aux trois questions que peut se poser l'emprunteur connaissant sa capacité mensuelle de remboursement, sous l'hypothèse simplificatrice que la méthode de calcul correcte est d'application; cela revient à programmer les inverses `somme`, `taux` et `nombre_periodes` de la fonction `periodicite` par rapport à ses arguments `S`, `t` et `n`, respectivement. On rappelle d'abord le code de la fonction `periodicite` :

```
(define periodicite
  (lambda (S t n)
    (/ (* S t (expt (+ 1 t) n))
       (- (expt (+ 1 t) n) 1))))
```

Les trois programmes d'inversion sont construits de la même manière :

```
(define periodicite<-somme
  (lambda (S tn) (periodicite S (car tn) (cadr tn))))
(define somme<-periodicite (inv+ periodicite<-somme))
(define somme
  (lambda (M t n) (somme<-periodicite M (list t n))))

(define periodicite<-taux
  (lambda (t Sn) (periodicite (car Sn) t (cadr Sn))))
```

```
(define taux<-periodicite (inv+ periodicite<-taux))
(define taux
  (lambda (S M n) (taux<-periodicite M (list S n))))

(define periodicite<-nombre_periodes
  (lambda (n St) (periodicite (car St) (cadr St) n)))
(define nombre_periodes<-periodicite
  (inv- periodicite<-nombre_periodes))
(define nombre_periodes
  (lambda (S t M) (nombre_periodes<-periodicite M (list S t))))
```

On notera la technique très simple utilisée pour regrouper en un seul argument les deux arguments de `periodicite` qui ne sont pas concernés par l'inversion. On observera aussi que la fonction `periodicite` est croissante en ses deux premiers arguments et décroissante dans le troisième; on a donc utilisé `inv+` dans les deux premiers cas et `inv-` dans le troisième.

6.4.4 Première cause de divergence

Les programmes proposés devraient en principe résoudre le problème : ils permettent de calculer la mensualité en fonction du taux mensuel et réciproquement. Néanmoins, ces programmes élémentaires n'expliquent en rien les nombreuses divergences défavorables entre les résultats fournis par ces programmes et les conditions proposées par les organismes prêteurs.

Pour expliquer ces divergences, il a fallu connaître les techniques de calcul utilisées. Elles consistent, d'une part, à "simplifier" le rapport existant entre les données annuelles et les données mensuelles (technique dite "mois / année")[81] et, d'autre part, à "simplifier" la formule de calcul des mensualités (technique dite "du taux de chargement"); seule la première sera étudiée dans ce paragraphe. La technique "mois / année" se pratique de deux manières. La première consiste à considérer que le taux mensuel (utilisé) est le douzième du taux annuel (annoncé); c'est systématiquement défavorable à l'emprunteur car on a

$$(1 + t_a) = (1 + t_m)^{12} > 1 + 12t_m \,,$$

d'où l'on conclut immédiatement que le taux mensuel correspondant à un taux annuel donné est moindre que le douzième de ce taux annoncé. Les passages du taux annuel au taux mensuel (exact) et réciproquement se programment aisément :

```
(define tm<-ta (lambda (ta) (- (expt (+ ta 1) 1/12) 1)))

(define ta<-tm (lambda (tm) (- (expt (+ tm 1) 12) 1)))
```

[81] En profitant du fait que les remboursements sont souvent mensuels alors que le taux annoncé est annuel.

L'organisme prêteur qui applique un taux mensuel effectif égal au douzième de son taux annuel nominal utilise en fait un taux annuel effectif supérieur au taux nominal annoncé; pour savoir quel taux annuel lui sera réellement appliqué, le client pourra utiliser le programme suivant :

```
(define ta<-ta_1 (lambda (t) (ta<-tm (/ t 12))))

(ta<-ta_1 0.06)  ==>  .0616778
```

On voit par exemple qu'un taux nominal de 6 % correspond à un taux réel de 6.168 %.

La seconde manière d'appliquer la technique "mois / année" consiste à calculer l'annuité sur base du taux annuel annoncé, puis à considérer que la mensualité est le douzième de l'annuité. A nouveau, ceci est clairement désavantageux pour l'emprunteur, qui rembourse chaque mensualité avec une avance de 1 à 11 mois, c'est-à-dire avec, en moyenne, cinq mois et demi d'avance. Ici aussi, le client souhaitera connaître le taux réel correspondant au taux nominal annoncé par l'organisme pratiquant cette méthode. On peut obtenir le programme adéquat `ta<-ta_2` en calculant successivement, au départ du taux annuel nominal, l'annuité, la mensualité, le taux mensuel réel et le taux annuel réel; autrement dit, `ta<-ta_2` pourrait se réduire à la composition des fonctions que l'on appellerait naturellement `annu<-ta_2`, `mensu<-annu`, `tm<-mensu` et `ta<-tm`. Il est tentant de réaliser ces compositions au moyen de la fonction `compose_list` introduite au paragraphe 6.2.2 mais un problème technique apparaît : certaines fonctions à composer comportent plus d'un arguments, les valeurs retournées dépendant non seulement du taux mais aussi de la somme empruntée et de la durée du prêt.

Une première solution consiste à supprimer artificiellement ces arguments surnuméraires en les remplaçant par des variables globales `*S*` (somme empruntée) et `*n*` (durée du prêt en années).[82] En réutilisant diverses fonctions précédemment introduites, on a :

```
(define annu<-ta_2 (lambda (t) (periodicite *S* t *n*)))

(define mensu<-annu (lambda (a) (/ a 12.0)))

(define tm<-mensu (lambda (M) (taux *S* M (* 12 *n*))))

(define ta<-tm (lambda (tm) (- (expt (+ tm 1) 12) 1)))

(define ta<-ta_2
 (compose_list (list ta<-tm tm<-mensu mensu<-annu annu<-ta_2)))
```

A titre d'illustration, nous considérons le cas d'un taux annuel nominal de 6 % et d'une somme d'un million, à rembourser en 20, 10 ou 3 ans.

[82] Rappelons ici la convention introduite au paragraphe 2.7.4 : les identificateurs correspondant à des variables globales définies par l'utilisateur commencent et se terminent par le caractère "*".

```
(define *S* 1000000)  ==>  ...
(define *n* 20)  ==>  ...
(ta<-ta_2 .06)  ==>  .063523
(define *n* 10)  ==>  ...
(ta<-ta_2 .06)  ==>  .066294
(define *n* 3)  ==>  ...
(ta<-ta_2 .06)  ==>  .079255
```

On constate immédiatement que cette deuxième manière d'appliquer la méthode "mois / année" est encore plus défavorable à l'emprunteur que la première manière, surtout si la durée du prêt est faible.

L'usage des variables globales est à éviter car elles nuisent à la modularité et à la transparence du code. Il est tout à fait contraire à l'esprit de la programmation fonctionnelle de permettre que la valeur [[`(ta<-ta_2 t)`]] d'une combinaison ne soit pas entièrement déterminée par les valeurs [[`ta<-ta_2`]] et [[`t`]] de l'opérateur et de l'opérande. Il est donc impératif d'éliminer `*n*` mais on peut se permettre de conserver `*S*` car la valeur [[`(ta<-ta_2 t)`]] est en fait indépendante de `*S*`. Par contre, cette dernière valeur influe sur les résultats retournés par certaines fonctions auxiliaires; cela sera toutefois sans inconvénient si ces fonctions sont "cachées". Contrairement à ce que l'on pourrait croire, le fait de réintroduire la durée du prêt comme argument explicite de certaines fonctions ne nous empêche pas de continuer à utiliser `compose_list`. On a le code suivant :

```
(define make_ta<-ta_2
  (lambda (n)
    (compose_list
      (list (lambda (tm) (- (expt (+ tm 1) 12) 1)) ; ta<-tm
            (lambda (M) (taux *S* M (* 12 n)))     ; tm<-mensu
            (lambda (a) (/ a 12.0))                ; mensu<-annu
            (lambda (t) (periodicite *S* t n))))))  ; annu<-ta_2
```

On notera que `make_ta<-ta_2` est une fonction qui à tout nombre positif (représentant la durée du prêt) associe une fonction équivalente à `ta<-ta_2` donnée plus haut. En reprenant les mêmes exemples que précédemment, on obtient

```
(define *S* 1234567)  ==>  ... ;; somme quelconque
((make_ta<-ta_2 20) .06)  ==>  .063523
((make_ta<-ta_2 10) .06)  ==>  .066294
((make_ta<-ta_2  3) .06)  ==>  .079255
```

La technique consistant à écrire un "générateur de fonction" permet d'éviter les variables globales gênantes sans que l'on doive modifier le nombre d'arguments des fonctions impliquées. Nous aurons encore l'occasion d'écrire des générateurs de fonctions.

6.4.5 Deuxième cause de divergence

Le développement qui précède rend compte des divergences observées dans le cas des prêts immobiliers mais pas de celles, généralement plus graves, observées dans le cas des prêts à la consommation. A nouveau, une petite enquête permet de mettre en évidence le moyen astucieux qu'utilisent certains organismes prêteurs pour annoncer des taux avantageux. Il est bien clair que l'emprunteur d'une somme S remboursera, le plus souvent en plusieurs fois, une somme totale S' supérieure à S. La différence $S' - S$ est une mesure du prix du crédit. On définit le *taux de chargement* comme le quotient $t_c =_{def} (S' - S)/nS$.[83] L'emprunteur d'une somme S remboursable en n mois au taux de chargement mensuel t_c rembourse chaque mois la somme $M = S'/n = (1/n + t_c)S$. La notion de taux de chargement ressemble à celle de taux d'intérêt. Plus le crédit est cher, plus le taux de chargement est élevé et un crédit gratuit correspond à un taux de chargement nul. En outre, le taux de chargement se confond avec le taux d'intérêt dans deux cas extrêmes. Le premier est celui d'un prêt de durée unitaire : il y a donc un seul remboursement de $S' = (1+t_m)S = (1+t_c)S$; le second est celui d'un prêt perpétuel, dans lequel l'emprunteur paie uniquement les intérêts, soit un versement mensuel permanent, égal à St_m ou St_c. Par contre, pour toute durée n bornée plus grande que 1, un taux de chargement donné correspond à un taux d'intérêt mensuel supérieur, la différence dépendant de la durée du prêt. En réutilisant des fonctions antérieurement définies on obtient aisément les fonctions permettant de calculer la mensualité à partir de la somme à emprunter, du taux de chargement ou du taux d'intérêt et de la durée du prêt en mois :

```
(define mensualite<-tm periodicite)

(define tm<-mensualite taux)

(define mensualite<-tc
  (lambda (S tc n) (* (+ (/ 1 n) tc) S)))

(define tc<-mensualite
  (lambda (S M n) (- (/ M S) (/ 1 n))))
```

On en tire immédiatement les fonctions de conversion :

```
(define tc<-tm
  (lambda (tm n)
    (tc<-mensualite *S* (mensualite<-tm *S* tm n) n)))

(define tm<-tc
  (lambda (tc n)
    (tm<-mensualite *S* (mensualite<-tc *S* tc n) n)))
```

[83] Seul le taux de chargement mensuel semble être utilisé en pratique.

On calcule d'abord quel taux d'intérêt mensuel correspond à un taux de chargement de 0.5 %, pour des durées de prêt de 1, 12, 30 et 240 mois; on obtient

```
(tm<-tc 0.005   1)  ==>  0.005000
(tm<-tc 0.005  12)  ==>  0.009080
(tm<-tc 0.005  30)  ==>  0.009265
(tm<-tc 0.005 240)  ==>  0.007719
```

On calcule de même quel taux de chargement correspond à un taux d'intérêt mensuel de 0.5 % :

```
(tc<-tm 0.005   1)  ==>  0.005000
(tc<-tm 0.005  12)  ==>  0.002733
(tc<-tm 0.005  30)  ==>  0.002646
(tc<-tm 0.005 240)  ==>  0.002998
```

On voit que la confusion entre les deux notions de taux peut coûter cher !

7 Accumulateurs et processus itératifs

On a montré au paragraphe 4.5 comment le programme `fib`, inspiré directement de la définition récursive de la suite de Fibonacci, était trop inefficace en pratique; le programme `fib_a`, introduit dans le même paragraphe, permettait d'éliminer cet inconvénient, grâce à deux arguments supplémentaires, destinés à mémoriser temporairement des résultats intermédiaires utiles. Nous avions aussi noté que, dans le programme `fib_a`, la récursivité intervenait sous forme dite *dégénérée* ou *terminale*. Le processus de calcul présente dans le cas de la récursivité dégénérée une structure très simple; on dit qu'il est *itératif*. Dans ce chapitre, nous étudions ces notions de manière plus approfondie.

7.1 Le principe de l'accumulateur

Un accumulateur est un argument supplémentaire, lié à des résultats intermédiaires à mémoriser. La notion de résultat intermédiaire est à prendre au sens le plus large : il s'agit de toute valeur dont la connaissance est mise à profit par le processus de calcul pour déterminer le résultat final.

Considérons une fois de plus le processus de calcul lié à la fonction classique `fact` (§ 4.3), ou plutôt sa simulation par le modèle de substitution :

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
(* 4 (* 3 (* 2 (* 1 1))))
(* 4 (* 3 2))
(* 4 6)
==> 24
```

On voit que ce processus comporte une première phase d'expansion, pendant laquelle aucune multiplication n'est effectuée, suivie d'une phase de réduction, consistant essentiellement en l'évaluation des multiplications. Vu l'associativité de la multiplication, le processus

```
(fact 4)
(* 4 (fact 3))
(* (* 4 3) (fact 2))
(* (* 4 3 2) (fact 1))
(* (* 4 3 2 1) (fact 0))
(* 4 3 2 1)
==> 24
```

serait équivalent. Il peut se simplifier en

```
(* 1 (fact 4))
(* 4 (fact 3))
```

```
(* 12 (fact 2))
(* 24 (fact 1))
(* 24 (fact 0)) ==> 24
```

L'intérêt est que chaque état (ou étape) du processus est caractérisé par deux paramètres seulement. Cela représente, en espace mémoire, une nette économie par rapport au processus initial. Etant donné un programme, il est simple, en principe, d'étudier le processus de calcul associé : il suffit de bien connaître l'ensemble des règles du langage de programmation utilisé, ce que l'on appelle la *sémantique opérationnelle* du langage. La démarche inverse, qui consiste à reconstituer le programme sur base du processus associé, reste facile dans le cas présent. On va définir une fonction récursive à deux arguments, nommée `fact_a`, telle que les appels successifs se feront avec des arguments égaux aux valeurs successives des deux paramètres du processus. Cette idée est "naturelle" dans la mesure où le second paramètre (qui deviendra le premier argument) évoque immédiatement notre schéma habituel de récursion sur les nombres naturels. Le premier paramètre (qui deviendra le second argument) répond à notre définition de l'accumulateur : ses valeurs successives sont clairement des résultats intermédiaires. Par le modèle de substitution, on a :

```
(fact_a 4 1)
(fact_a 3 4)
(fact_a 2 12)
(fact_a 1 24)
(fact_a 0 24) ==> 24
```

La définition de `fact_a` est maintenant évidente :[84]

```
(define fact_a
  (lambda (n a)
    (if (zero? n) a (fact_a (- n 1) (* a n)))))
```

On peut à présent redéfinir la procédure `fact` comme suit :

```
(define fact
  (lambda (n) (fact_a n 1)))
```

La fonction `fact_a` est devenue une fonction auxiliaire; à ce titre, il est *indispensable* de la spécifier, ce que l'on fera de la manière suivante :

> *La fonction* `fact_a` *prend comme arguments*
> *un entier naturel* n *et un nombre* a*;*
> *elle renvoie comme résultat le produit* $n!a$*.*

[84] Observons que le processus associé à la boucle `while n > 0 do (a,n) := (a*n,n-1)` (construction classique de nombreux langages de programmation) est analogue à celui associé à l'évaluation de la forme `(fact_a n a)`.

Signalons d'emblée qu'une phrase telle que "`fact_a` est une version accumulante de `fact`" est trop vague pour être d'une quelconque utilité.[85]

7.2 Autres exemples numériques

La solution "accumulante" que nous venons de commenter n'est pas la seule possible pour le calcul de la factorielle. Plutôt que de considérer des résultats partiels du type $n * (n-1) * \cdots$, on pourrait accumuler des produits du type $1 * 2 * \cdots$. Une solution alternative consiste en la fonction `fact_b`, spécifiée comme suit :

> *Si les valeurs de* `n`, `i` *et* `b` *sont* n, i $(1 \leq i \leq n+1)$ *et* $(i-1)!$,
> *alors la valeur de* `(fact_b n i b)` *est* $n!$.

On peut écrire

```
(define fact_b
  (lambda (n i b)
    (if (> i n) b (fact_b n (+ i 1) (* b i)))))

(define fact
  (lambda (n) (fact_b n 1 1)))
```

Remarque. Notre spécification pourrait paraître incomplète, puisqu'elle ne précise pas la valeur de `(fact_b n i b)` quand [[`b`]] n'est pas $([[\texttt{i}]]-1)!$. L'objection n'est pas valable parce qu'en usage normal ce cas ne se produit jamais, ni au premier niveau, ni lors d'appels récursifs subséquents. Par contre on ne pourrait se contenter de dire que $[[\texttt{(fact\_b n 1 1)}]] = [[\texttt{n}]]!$.

L'emploi d'un accumulateur permet de diminuer significativement l'espace mémoire requis par l'exécution du processus de calcul de la factorielle. Un gain nettement plus significatif est possible dans le cas de la suite de Fibonacci, que nous avons déjà rencontré. Le code "naïf", directement inspiré de la définition mathématique (§ 1.2.6) a pu être nettement amélioré, en fait par l'adjonction de deux accumulateurs permettant de mémoriser les deux derniers termes calculés de la suite de Fibonacci. Le code de la fonction `fib_a` a été donné au paragraphe 4.5, ainsi que sa spécification et la description du processus de calcul associé.[86]

De manière analogue, la fonction "exponentielle rapide", dont le code a été donné au paragraphe 5.6, admet une variante accumulante :

[85] Il serait encore plus vain d'écrire "La fonction `fact_a` prend comme arguments un entier naturel n et le nombre 1; elle renvoie comme résultat le nombre $n!$". Tout d'abord, cette phrase n'est qu'une paraphrase du code de `fact` : elle est donc inutile. En outre, il va de soi que seul le premier appel se fait avec $a = 1$; lors des appels récursifs, on a en général $a \neq 1$. Enfin, le lecteur d'une telle phrase est censé en admettre la véracité comme un acte de foi; ce n'est pas le cas pour la spécification correcte, qui se démontre aisément par récurrence sur n. Nous aurons encore l'occasion dans la suite de montrer comment il convient de spécifier les fonctions accumulantes.

[86] Ce processus est itératif et analogue à celui associé à la boucle
`while n > 0 do (n,a,b) := (n-1,b,a+b)`.

```
(define exp
  (lambda (m n) (exp_a m n 1)))

(define exp_a
 (lambda (m n a)
  (cond ((zero? n) a)
        ((even? n) (exp_a (* m m) (/ n 2) a))
        ((odd? n) (exp_a m (- n 1) (* m a))))))
```

On a $[[\texttt{(exp\_a m n a)}]] = [[\texttt{a}]][[\texttt{m}]]^{[[\texttt{n}]]}$, d'où $[[\texttt{(exp\_a m n 1)}]] = [[\texttt{m}]]^{[[\texttt{n}]]}$ en particulier.
Considérons encore l'exemple du coefficient binomial :

```
(define cbin
 (lambda (n u)
  (if (zero? n)
      1
      (/ (* (cbin (- n 1) (- u 1)) u) n))))
```

dont voici la version accumulante :

```
(define cbin
 (lambda (n u) (cbin_a n u 1)))

(define cbin_a
 (lambda (n u a)
  (if (zero? n)
      a
      (cbin_a (- n 1) (- u 1) (/ (* u a) n)))))
```

Voici un exemple d'exécution :

```
(cbin 4 6)      ==>  15
(cbin_a 4 6 1)  ==>  15
```

La spécification de `cbin_a` est laissée au lecteur.

7.3 Accumulateurs et traitement de listes

Nous avons déjà rencontré (§ 5.3) la fonction `reverse` :

```
(define reverse
  (lambda (u)
    (if (null? u)
        '()
        (append (reverse (cdr u))
                (list (car u))))))
```

A cause de l'emploi de la fonction `append` (d'efficacité linéaire en son premier argument), la fonction `reverse` est d'efficacité quadratique. Donnons un exemple d'application, en utilisant le modèle de substitution :

```
(reverse '(0 1 2))
(append (reverse '(1 2)) '(0))
(append (append (reverse '(2)) '(1)) '(0))
(append (append (append (reverse '()) '(2)) '(1)) '(0))
(append (append (append '() '(2)) '(1)) '(0))
(append (append '(2) '(1)) '(0))
(append '(2 1) '(0)) ==> (2 1 0)
```

On peut aisément obtenir une version accumulante linéaire :

```
(define reverse_a
  (lambda (u a)
    (if (null? u)
        a
        (reverse_a (cdr u) (cons (car u) a)))))
```

Le modèle de substitution donne :

```
(reverse_a '(0 1 2) '())
(reverse_a '(1 2) '(0))
(reverse_a '(2) '(1 0))
(reverse_a '() '(2 1 0)) ==> (2 1 0)
```

On a la spécification suivante :

> *Pour toutes listes* `u` *et* `a`,
> `(append (reverse u) a)` *équivaut à* `(reverse_a u a)`.

En particulier, `(reverse u)` équivaut à `(reverse_a u '())`.

Un autre exemple, déjà rencontré (§§ 5.4-5.5), est celui du calcul de la liste des feuilles d'une liste littérale ou d'un arbre. On construit immédiatement un programme simple et correct pour ce calcul :

```
(define flat_list
  (lambda (l)
    (cond ((null? l) '())
          ((atom? (car l))
           (if (null? (car l))
               (flat_list (cdr l))
               (cons (car l) (flat_list (cdr l)))))
          (else
           (append (flat_list (car l)) (flat_list (cdr l)))))))
```

On a par exemple

```
(flat_list '(a (b c) ((((d))) e) b))  ==>  (a b c d e b)
```

Ce programme donne cependant lieu à un processus de calcul relativement complexe, suite à l'utilisation de la fonction auxiliaire `append`. Pour y remédier, on définit une fonction auxiliaire `flat_list_a`, plus générale. Elle prend un second argument `a`, un accumulateur destiné à contenir un résultat intermédiaire (la valeur de `a` sera donc une liste littérale plate, c'est-à-dire une liste de lettres). Cette fonction auxiliaire est spécifiée par l'égalité

$$[[\texttt{(flat\_list\_a l a)}]] \;=\; [[\texttt{(append (flat\_list l) a)}]]\ ,$$

valable pour toute liste littérale `l` et pour toute liste `a`. On a le code suivant :

```
(define flat_list_a
  (lambda (l a)
    (cond
      ((null? l) a)
      ((atom? (car l))
       (if (null? (car l))
           (flat_list_a (cdr l) a)
           (cons (car l) (flat_list_a (cdr l) a))))
      (else
       (flat_list_a (car l) (flat_list_a (cdr l) a))))))
```

On a par exemple

```
(flat_list_a '(a (b c) ((((d))) e) b) '())  ==>  (a b c d e b)
(flat_list_a '(a (b (c) d)) '(1 2 3))  ==>  (a b c d 1 2 3)
```

De la spécification générale donnée plus haut, on tire

`(flat_list l)` équivaut à `(flat_list_a l '())`.

L'usage de la fonction auxiliaire `flat_list_a` procure un gain de temps et d'espace à l'exécution, quoique le processus associé ne soit pas itératif.

7.4 Conception de programme, cinquième étude

Nos premières études de cas (§ 6) avaient pour but de montrer comment l'énoncé du problème à résoudre suggérait, de manière plus ou moins directe, divers moyens de solution; on a vu aussi comment choisir entre ces moyens, et comment organiser le travail de conception du programme, notamment par la spécification et la définition de quelques fonctions auxiliaires appropriées. Dans les trois premiers cas, l'usage direct ou indirect des schémas de programme s'est révélé utile. Nous traitons ici un problème très simple mais néanmoins susceptible de recevoir des solutions variées. On verra en particulier que la notion de schéma de programme doit s'utiliser avec souplesse et discernement, de même que celle d'accumulateur.

7.4.1 L'énoncé

> *Ecrire une fonction* `lpref` *prenant comme argument une liste* `l`
> *et retournant la liste des préfixes de* `l`.

Cette spécification est claire; il est cependant nécessaire de préciser la notion de préfixe, en décidant par exemple que la liste vide et la liste `u` elle-même sont des préfixes de `u`.[87]

7.4.2 Première solution

Notre troisième étude de cas avait montré que l'usage direct du schéma de programme concernant les listes (ici, les listes plates) pouvait conduire à une solution simple, claire et raisonnablement efficace. Nous décidons donc d'emblée que le calcul de `(pref l)` passera par celui de `(pref (cdr l))`. On a donc immédiatement un "squelette" de solution :

```
(define lpref1
  (lambda (l)
    (if (null? l)
        C                          ;; cas de base
        (F (lpref1 (cdr l)) l))))  ;; cas inductif
```

Clairement, la liste vide admet un seul préfixe, elle-même. En outre, les préfixes d'une liste `l` non vide sont, d'une part, la liste vide et, d'autre part, les préfixes de `(cdr l)`, chacun d'eux étant préfixé de `(car l)`. On obtient immédiatement le code suivant :

```
(define lpref1
  (lambda (l)
    (if (null? l)
        (list l)
        (cons '() (insert_in_all (car l) (lpref1 (cdr l)))))))
```

La fonction auxiliaire `insert_in_all` a été introduite au paragraphe 6.3.2. Elle prend comme arguments un objet `x` et une liste de listes `ll`; elle renvoie une liste de listes, dont les éléments sont ceux de `ll` préfixés de `x`. On observe que les fonctions `insert_in_all` et `lpref1` sont des instances du schéma habituel.[88] Voici un exemple d'utilisation :

```
(lpref1 '(a b c))  ==>  (() (a) (a b) (a b c))
```

Le processus associé à `lpref1` n'est pas itératif, pas plus d'ailleurs que celui associé à `insert_in_all`. A titre d'illustration, on va étudier la possibilité de produire une version accumulante de ces fonctions. On rappelle d'abord le code de `insert_in_all` :

[87] Un autre choix serait également acceptable; le point important est d'expliciter ce choix.

[88] On aurait pu définir `insert_in_all` via la primitive `map` :

```
(define put (lambda (x ll) (map (lambda (l) (cons x l)) ll)))
```

```
(define insert_in_all
  (lambda (x ll)
    (if (null? ll)
        '()
        (cons (cons x (car ll))
              (insert_in_all x (cdr ll))))))
```

Par analogie avec les exemples traités au début de ce chapitre, on obtient la version suivante :

```
(define insert_in_all_a
  (lambda (x ll a)
    (if (null? ll)
        a
        (insert_in_all_a x
                         (cdr ll)
                         (cons (cons x (car ll)) a)))))
```

Voici des exemples d'exécution :

```
(insert_in_all 'x '((a b) () (c)))
  ==>  ((x a b) (x) (x c))
(insert_in_all_a 'x '((a b) () (c)) '())
  ==>  ((x c) (x) (x a b))
(insert_in_all_a 'x '((a b) () (c)) '(1 (2) 3))
  ==>  ((x c) (x) (x a b) 1 (2) 3)
```

On observe que l'ordre des éléments dans la liste est inversé. Dans le cas présent, cela n'est pas gênant; il est cependant indispensable de mentionner ce fait dans la spécification de `insert_in_all_a`, que nous laissons au lecteur comme exercice. On peut maintenant essayer d'obtenir une variante accumulante de `lpref1`, en appliquant la technique déjà utilisée pour `insert_in_all`. On observe d'abord que `lpref1` peut se récrire en utilisant `insert_in_all_a` au lieu de `insert_in_all`. La version naïve

```
(define lpref1bis
  (lambda (l)
    (if (null? l)
        (list l)
        (cons '()
              (insert_in_all_a (car l)
                               (lpref1bis (cdr l))
                               '())))))
```

n'est pas très satisfaisante parce qu'elle produit les préfixes dans un ordre peu conventionnel :

```
(lpref1bis '(a b c))  ==>  (() (a b) (a b c) (a))
```

On pourrait remédier à cet inconvénient comme suit :

```
(define lpref1ter
  (lambda (l)
    (if (null? l)
        (list l)
        (cons '()
              (insert_in_all_b (car l)
                               (lpref1ter (cdr l))
                               '())))))

(define insert_in_all_b
  (lambda (x ll a)
    (if (null? ll)
        a
        (insert_in_all_b x
                         (cdr ll)
                         (append a (list (cons x (car ll))))))))
```

La fonction `insert_in_all_b` renvoie maintenant une liste non inversée. On a :

```
(insert_in_all_b 'x '((a) () (b c)) '())
  ==>  ((x a) (x) (x b c))

(lpref1ter '(a b c))  ==>  (() (a) (a b) (a b c))
```

Cette solution n'est pas satisfaisante car la fonction `insert_in_all_b` est inefficace (à cause de l'intervention de `append`). De plus, le processus associé à l'application de `lpref1ter` à ses arguments n'est pas itératif. A ce stade, il semble préférable d'abandonner la recherche d'une version accumulante et/ou itérative de `lpref1`.

7.4.3 Deuxième solution

Le lecteur aura sans doute remarqué que la liste des suffixes est plus facile à calculer que celle des préfixes. On a immédiatement

```
(define lsuff
  (lambda (l)
    (if (null? l)
        (list l)
        (cons l (lsuff (cdr l))))))

(lsuff '(a b c))  ==>  ((a b c) (b c) (c) ())
```

Cela est dû au fait que le plus grand suffixe propre d'une liste [[l]] non vide est tout simplement [[(cdr l)]]; ceci suggère d'écrire et d'utiliser une fonction auxiliaire `butlast`, renvoyant le plus grand préfixe propre d'une liste non vide. On a alors

```
(define lpref2
  (lambda (l)
    (if (null? l)
        (list l)
        (cons l (lpref2 (butlast l))))))

(define butlast
  (lambda (l)
    (if (null? (cdr l))
        '()
        (cons (car l) (butlast (cdr l))))))
```

Ici, `lpref2` n'est pas une instance du schéma habituel, car la fonction de réduction `cdr` est remplacée par la fonction `butlast`. On notera cependant que `butlast`, comme `cdr`, renvoie comme résultat une liste plus courte que la liste-argument; c'est ce qui détermine le succès de l'approche récursive. Notons aussi que la fonction `butlast` est une instance du schéma habituel. L'usage de `butlast` est un exemple de la souplesse nécessaire lors de l'emploi d'un schéma de récursion. Un autre avantage de cette solution est la possibilité d'obtenir immédiatement une version accumulante :

```
(define lpref2_a
  (lambda (l a)
    (if (null? l)
        (cons '() a)
        (lpref2_a (butlast l) (cons l a)))))

(lpref2 '(a b c))  ==>  ((a b c) (a b) (a) ())

(lpref2_a '(a b c) '())  ==>  (() (a) (a b) (a b c))
```

7.4.4 Troisième solution

On peut directement mettre à profit le fait que la liste des suffixes est facile à calculer. Il suffit d'observer que la liste des préfixes d'une liste est la liste des inverses des suffixes de la liste inverse. Cela donne lieu au code suivant :

```
(define lpref3
  (lambda (l)
    (lreverse (lsuff (reverse l)))))

(define lreverse
```

```
  (lambda (ll)
    (if (null? ll)
        '()
        (cons (reverse (car ll)) (lreverse (cdr ll))))))

(lpref3 '(a b c))  ==>  ((a b c) (a b) (a) ())
```

On laisse au lecteur le soin de spécifier la fonction `lreverse`; la fonction `reverse` est déjà connue.[89]

Remarque. Les trois solutions que l'on vient d'exposer sont des exemples typiques du style de programmation fonctionnel. Dans chaque cas, le problème est résolu par deux ou trois fonctions très simples, le plus souvent définies récursivement, selon un schéma élémentaire. L'utilisation de `(butlast l)` au lieu de l'habituel `(cdr l)` est une variante dictée par les particularités du problème posé, mais on reste dans un cadre strictement fonctionnel, qui se caractérise par trois points :

- Approche abstraite : le raisonnement n'évoque pas les étapes intermédiaires du calcul.
- Approche procédurale, de type "top-down" : toute fonction dont la programmation n'est pas immédiate suscite l'introduction (puis la programmation) d'une ou plusieurs fonctions auxiliaires.
- Usage systématique de la récursivité, inséparable de l'approche fonctionnelle.

7.5 Simplification syntaxique d'une récursion

La technique des accumulateurs permet parfois de transformer un schéma récursif difficile et/ou inhabituel en un schéma plus simple, voire dégénéré. Nous donnons ici quelques exemples typiques.

7.5.1 La fonction 91

Considérons d'abord l'exemple classique de la "fonction 91". Le code est

```
(define M
  (lambda (x)
    (if (> x 100) (- x 10) (M (M (+ x 11))))))
```

On pourrait croire que l'étude de la fonction `M` serait facilitée si on pouvait éliminer la récursivité imbriquée. Syntaxiquement, c'est très simple. Il suffit d'imaginer une fonction auxiliaire `M_c`, comportant un argument supplémentaire destiné à "contrôler" l'itération et employé comme suit :

`(M_c x 0)` équivaut à `x`;

[89] Notons aussi que `lreverse` peut se définir au moyen de la primitive `map` :
`(define lreverse (lambda (ll) (map reverse ll)))`

`(M_c x 1)` équivaut à `(M x)`;
`(M_c x 2)` équivaut à `(M (M x))`;
`(M_c x 3)` équivaut à `(M (M (M x)))`;
... ...
Il est alors immédiat d'écrire le code suivant :

```
(define M_c
  (lambda (x c)
    (cond ((= c 0) x)
          ((> x 100) (M_c (- x 10) (- c 1)))
          (else (M_c (+ x 11) (+ c 1))))))
```

Syntaxiquement, la récursivité est dégénérée ... mais sémantiquement, la fonction `M_c` est aussi "bizarre" que la fonction `M`.

Remarque. On peut démontrer que la valeur de `(M x)` existe quelle que soit la valeur de l'entier (relatif) `x` et, en outre, que cette valeur coïncide avec celle de `(M91 x)`, si on pose

```
(define M91
  (lambda (x)
    (if (> x 100) (- x 10) 91)))
```

Cela explique le nom donné à la fonction `M` et montre aussi que cette fonction n'a pas d'intérêt intrinsèque.

7.5.2 La fonction d'Ackermann

Un exemple du même genre mais plus significatif est celui de la fonction d'Ackermann. Il s'agit d'une fonction de $\mathbb{N} \times \mathbb{N}$ dans $\mathbb{N}$ définie récursivement comme suit :

```
(define ack
  (lambda (m n)
    (cond
      ((zero? m) (+ n 1))
      ((zero? n) (ack (- m 1) 1))
      (else (ack (- m 1) (ack m (- n 1)))))))
```

Ici aussi, on peut éliminer les appels récursifs imbriqués, mais compter le nombre d'imbrications ne suffit plus; il faut en outre, pour chacune d'elle, noter le premier argument de l'appel. Vu la sémantique opérationnelle de SCHEME, les appels les plus internes sont réduits avant les plus externes. On gérera donc une liste d'arguments telle que ceux des appels internes soient en tête. Concrètement, on prévoit une fonction `acl` prenant comme argument une liste non vide de naturels et telle que

`(acl '(n))` équivaut à `n`;
`(acl '(n m))` équivaut à `(ack m n)`;
`(acl '(n m p))` équivaut à `(ack p (ack m n))`;

... ...

Le code s'écrit alors facilement :

```
(define acl
  (lambda (l)
    (cond
      ((null? (cdr l))
       (car l))
      ((zero? (cadr l))
       (acl (cons (+ (car l) 1)
                  (cddr l))))
      ((zero? (car l))
       (acl (cons 1
                  (cons (- (cadr l) 1)
                        (cddr l)))))
      (else
       (acl (cons (- (car l) 1)
                  (cons (cadr l)
                        (cons (- (cadr l) 1)
                              (cddr l)))))))))
```

Remarque. On peut démontrer que la fonction `ack` est totale et croît plus rapidement que toute fonction primitive récursive.

Cet exemple illustre à la fois la puissance de la technique des accumulateurs et ses limites. Dans les deux cas on a obtenu des programmes à récursivité terminale mais cela n'a pas significativement simplifié l'étude des procédures concernées ni réellement amélioré leur efficacité.

7.6 Base fonctionnelle de l'accumulateur, style CPS

Le concept même d'accumulateur semble de nature opérationnelle. Sa motivation réside dans la structure d'un processus de calcul. Pour le voir, nous reconsidérons deux "états homologues" des processus de calcul associés aux évaluations des formes `(fact 4)` et `(fact_a 4 1)`, par exemple

```
(* 4 (* 3 (fact 2)))        (fact_a 2 12)
```

On a trois paires de constituants homologues :

```
fact                        fact_a
2                           2
(* 4 (* 3 ...))            12
```

On passe de la version de gauche à celle de droite en créant un argument supplémentaire, dont la valeur "représente" celle de l'expression `(* 4 (* 3 ...))`, c'est-à-dire une

fonction à un argument (qui remplace le "trou" noté par les points de suspension). On peut donc récrire l'expression en `(lambda (k) (* 4 (* 3 k)))`, dont la valeur est bien une fonction à un argument. Vu l'associativité de la multiplication, on peut simplifier cette λ-forme en `(lambda (k) (* 12 k))`. Enfin, comme toute fonction linéaire $x \mapsto ax$ est connue dès que a est connu, on peut représenter cette dernière λ-forme par son coefficient, ici `12`. Si la multiplication n'était pas associative, on aurait dû conserver la λ-forme `(lambda (k) (* 4 (* 3 k)))`, en la représentant éventuellement par la liste `(4 3)`.

Il ressort de ceci que l'accumulateur est une représentation non fonctionnelle d'un "argument implicite" fonctionnel. Il est d'ailleurs intéressant d'étudier ce qui se passe quand on se contente d'expliciter l'argument fonctionnel, sans chercher à le représenter de manière non fonctionnelle. Dans le cas de la factorielle, la version accumulante `fact_a` devient la version `fact_c` ci-dessous :

```
(define fact_c
  (lambda (n c)
    (if (zero? n)
        (c 1)
        (fact_c (- n 1)
                (lambda (k) (c (* n k)))))))
```

La factorielle se définit alors aisément :

```
(define fact
  (lambda (n) (fact_c n (lambda (k) k))))
```

Avant de spécifier `fact_c`, étudions le processus de calcul associé ou, plus précisément, un processus voisin. On a par exemple

```
(fact_c 4 (lambda (k) k))
(fact_c 3 (lambda (k) ((lambda (k) k) (* 4 k))))
(fact_c 3 (lambda (k) (* 4 k)))
(fact_c 2 (lambda (k) ((lambda (k) (* 4 k)) (* 3 k))))
(fact_c 2 (lambda (k) (* 4 (* 3 k))))
...
(fact_c 0 (lambda (k) (* 4 (* 3 (* 2 (* 1 k))))))
((lambda (k) (* 4 (* 3 (* 2 (* 1 k))))) 1)
(* 4 (* 3 (* 2 (* 1 1))))
==>  24
```

Remarque. SCHEME n'effectue les réductions qu'à la fin du processus. La dernière étape n'est donc pas l'évaluation de

```
((lambda (k) (* 4 (* 3 (* 2 (* 1 k))))) 1)
```

mais celle de

```
((lambda (k)
   ((lambda (k)
      ((lambda (k)
         ((lambda (k)
            ((lambda (k) k) (* 4 k)))
          (* 3 k)))
       (* 2 k)))
    (* 1 k)))
 1)
```

c'est pourquoi nous avons évoqué un processus voisin. Sur le plan conceptuel, cette différence n'est pas importante ici. Le point crucial est d'observer que `fact_c` admet la spécification suivante.

> *Si n est un naturel et si c est une fonction de $\mathbb{N}$ dans $\mathbb{N}$,*
> *alors* `(fact_c` n c `)` *vaut* $c(n!)$.

L'argument fonctionnel auxiliaire `c` est une *continuation*. Cette fonction, appliquée à un résultat intermédiaire du calcul en cours, fournit le résultat final.

Dans le cas de `fact_c`, l'exécution construit progressivement une fonction de plus en plus complexe, représentant l'enchaînement des multiplications à faire. A chaque étape, l'argument continuation est transformé en une fonction impliquant une multiplication supplémentaire; cette fonction est l'argument de l'appel suivant. Les calculs numériques n'ont lieu que quand l'enchaînement complet des multiplications a été formé. Cette technique est le *Continuation-Passing Style* (CPS). Elle ressemble à la technique de séparation fonctionnelle vue précédemment mais ne s'identifie pas à elle.

Le style CPS et la notion de continuation permettent de *réifier*, c'est-à-dire de transformer en un objet (en fait, une procédure) le concept opérationnel de processus de calcul. Cette transformation est utile parce qu'elle permet de séparer la "fabrication" d'un processus de calcul et la mise en œuvre de ce processus. Les exemples élémentaires vus au paragraphe 5.7 consacré à la séparation fonctionnelle ont déjà montré l'intérêt potentiel de ce type de transformation. Nous revoyons maintenant ces exemples pour illustrer le style CPS.

7.6.1 Le produit d'une liste de nombres

Rappelons ici les versions classiques de la fonction calculant le produit d'une liste de nombres (cf. § 5.7.2) :

```
(define prodlist0
  (lambda (l)
    (if (null? l) 1 (* (car l) (prodlist0 (cdr l))))))
```

```
(define prodlist1
  (lambda (l)
    (cond ((null? l) 1)
          ((zero? (car l)) 0)
          (else (* (car l) (prodlist1 (cdr l)))))))
```

On a, pour la liste `(3 5 0 2 4)`, le processus suivant :

```
(prodlist1 '(3 5 0 2 4))
(* 3 (pl '(5 0 2 4)))
(* 3 (* 5 (pl '(0 2 4))))
(* 3 (* 5 0))
(* 3 0)
==>  0
```

Il y a autant de multiplications (par 0) que de facteurs précédant le premier 0. Utiliser une version accumulante ne résout pas le problème. On peut écrire

```
(define prodlist1_a
  (lambda (l a)
    (cond ((null? l) a)
          ((zero? (car l)) 0)
          (else (prodlist1_a (cdr l) (* a (car l)))))))
```

On a $[\![\texttt{(prodlist1\_a l a)}]\!] = [\![\texttt{(prodlist1 l)}]\!] * [\![\texttt{a}]\!]$. Voici un exemple d'exécution :

```
(prodlist1_a '(3 5 0 2 4) 1)
(prodlist1_a '(5 0 2 4) (* 1 3))
(prodlist1_a '(5 0 2 4) 3)
(prodlist1_a '(0 2 4) (* 3 5))
(prodlist1_a '(0 2 4) 15)
==>  0
```

Il y a toujours le même nombre de multiplications; en outre, les facteurs impliqués ne sont pas nuls. Pour éviter à coup sûr *toutes* les multiplications au cas où l'un des facteurs est nul, il existe une solution élémentaire : on parcourt d'abord la liste pour rechercher un zéro éventuel. L'inconvénient est que, si aucun zéro n'est présent, on devra parcourir la liste une deuxième fois. La technique de séparation fonctionnelle a donné lieu à une solution, la fonction `prodlist2` (cf. § 5.7.2). Voilà maintenant une solution voisine, basée sur le CPS :

```
(define pl3_c
  (lambda (l c)
    (cond ((null? l) (c 1))
          ((zero? (car l)) 0)
          (else (pl3_c (cdr l)
                       (lambda (k) (c (* (car l) k))))))))
```

```
(define prodlist3
  (lambda (l) (pl3_c l (lambda (k) k))))
```

La fonction auxiliaire [[pl3_c]] peut être spécifiée comme suit :

> *Si* [[l]] *est une liste dont les éléments sont les nombres entiers (ou réels)* $x_1, \ldots, x_n$ *et si* [[f]] *est une fonction* f *de domaine* $\mathbb{N}$ *(ou* $\mathbb{R}$*), alors* [[(pl3_c l f)]] *est* $f(\Pi_{i=1}^{n} x_i)$.

En particulier, cette la fonction [[(pl3_c)]] calcule le produit des éléments de son premier argument (une liste de nombre) si son second argument est la fonction identique [[(lambda (k) k)]]. A titre d'exemple, on a

```
(pl3_c (enum 1 3) (lambda (q) (+ 2 q)))  ==>  8
```

puisque $2 + (1 * 2 * 3) = 8$. De même, on a

```
(pl3_c '(1 2 1.5 1)
       (lambda (k)
         (pl3_c (enum 1 (+ k 1))
                (lambda (q) (+ 22 q)))))  ==>  46
```

puisque $1 * 2 * 1.5 * 1 = 3$, $\Pi([[(\texttt{enum 1 (+ k 1)})]]) = 24$ si [[k]] $= 3$ et enfin [[(+ 22 q)]] $= 46$ si [[q]] $= 24$.

Observons aussi que l'on retombe sur la solution élémentaire si on remplace la continuation par un argument non fonctionnel :

```
(define pl4_a
  (lambda (l a)
    (cond ((null? l) (prodlist0 a))
          ((zero? (car l)) 0)
          (else (pl4_a (cdr l)
                       (cons (car l) a))))))

(define prodlist4
  (lambda (l) (pl4_a l '())))
```

On note que l'argument supplémentaire `a` n'est pas un nombre, mais une liste de nombres non nuls; pour multiplier ces nombres, on utilise donc la procédure `prodlist0` qui ne teste pas les nombres à multiplier. La spécification est

> *Si* [[l]] *et* [[a]] *sont des listes dont les éléments sont les nombres entiers (ou réels)* $x_1, \ldots, x_n$ *et* $y_1, \ldots, y_m$*, respectivement, alors* [[(pl4_c l a)]] *est* $(\Pi_{i=1}^{n} x_i)(\Pi_{i=1}^{m} y_i)$.

7.6.2 Le double comptage

La séparation fonctionnelle permettait aussi d'éviter efficacement le double parcours de la liste pour le problème du double comptage. Nous rappelons ici la solution présentée au paragraphe 5.7.1 :

```
(define c2
  (lambda (l s c)
    (if (null? l)
        c
        (if (eq? (car l) s)
            (c2 (cdr l)
                s
                (lambda (u)
                  (c (1+ (car u)) (cadr u))))
            (c2 (cdr l)
                s
                (lambda (u)
                  (c (car u) (1+ (cadr u)))))))))

(define id (lambda (v) v))

(define count2 (lambda (l s) ((c2 l s id) 0 0)))
```

La solution en CPS est semblable :

```
(define c3
  (lambda (l s c)
    (if (null? l)
        (c 0 0)
        (if (eq? (car l) s)
            (c3 (cdr l)
                s
                (lambda (u v) (c (1+ u) v)))
            (c3 (cdr l)
                s
                (lambda (u v) (c u (1+ v))))))))

(define count3
  (lambda (l s) (c3 l s cons)))
```

L'argument fonctionnel de la fonction `c3` évolue simplement; si sa valeur avant un appel récursif est celle de `c`, soit

```
(lambda (x y) (c x y))
```

sa nouvelle valeur sera celle d'une des expression

```
(lambda (x y) (c (1+ x) y))
(lambda (x y) (c x (1+ y)))
```

On peut remplacer l'argument fonctionnel par deux arguments numériques. Cela donne :

```
(define c4
  (lambda (l s a1 a2)
    (if (null? l)
        (cons a1 a2)
        (if (eq? (car l) s)
            (c4 (cdr l) s (1+ a1) a2)
            (c4 (cdr l) s a1 (1+ a2))))))

(define count4
  (lambda (l s) (c4 l s 0 0)))
```

Cette solution est simple et optimale. On voit que la séparation fonctionnelle et le CPS sont des techniques utiles, donnant lieu à des solutions efficaces. Parfois, le remplacement de l'argument fonctionnel par un ou plusieurs accumulateur(s) améliore encore le programme.

8 Expressions symboliques

Au paragraphe 2.5, nous avions déjà brièvement évoqué la "notation pointée", caractérisée par l'égalité

$$[[\texttt{(cons } \alpha \ \beta\texttt{)}]] = \texttt{(}[[\alpha]] \ \texttt{.} \ [[\beta]]\texttt{)}\,.$$

Les deux notations représentent des listes si et seulement si $[[\beta]]$ est une liste, mais l'égalité garde un sens et reste vraie quels que soient α et β. Tout objet résultant de l'application de `cons` à deux arguments est, par définition, une *paire pointée*. Toute liste non vide est donc une paire pointée. Dans ce chapitre, nous étudions les *expressions symboliques* c'est-à-dire, essentiellement, les paires pointées.

Définition. Une *expression symbolique* est un atome ou une paire formée d'expressions symboliques.
Remarque. Au sens le plus large, un atome est tout objet qui n'est pas une paire, c'est-à-dire tout objet qui ne peut être créé avec `cons`. En ce sens, tout objet SCHEME est une expression symbolique. Le plus souvent, on restreindra l'ensemble des atomes,[90] ce qui reviendra à restreindre l'ensemble des expressions symboliques. Notre but ici étant d'étudier les propriétés structurelles des expressions symboliques, il est inutile d'être plus précis à ce stade.

8.1 Arbres binaires

L'ensemble des atomes étant fixé, le domaine des expressions symboliques peut être vu comme un type abstrait de donnée. Les constructeur et accesseurs sont `cons`, `car`, `cdr`. On a aussi le reconnaisseur `pair?` qui reconnaît les paires pointées, et éventuellement un reconnaisseur spécifique pour l'ensemble des atomes admis, que nous pouvons noter `s-atom?`. Le reconnaisseur du type sera alors défini comme suit :

```
(define s-exp?
  (lambda (x)
    (if (atom? x)
        (s-atom? x)
        (and (s-exp? (car x)) (s-exp? (cdr x))))))
```

On peut aussi définir

```
(define s-pair?
  (lambda (x)
    (and (pair? x) (s-exp? (car x)) (s-exp? (cdr x)))))
```

Si on note E l'ensemble des atomes admis, l'ensemble des expressions symboliques apparaît comme la réalisation en SCHEME du domaine des E-arbres binaires introduit au paragraphe 1.3.2. Par exemple, si E est l'alphabet, on écrira :

[90] Il est courant de se limiter aux symboles atomiques; on leur ajoute parfois la liste vide, les nombres et/ou les booléens.

```
(define *alphabet*
  '(a b c d e f g h i j k l m n o p q r s t u v w x y z))

(define s-atom? (lambda (x) (member x *alphabet*)))
```

8.2 Représentation en mémoire

La zone mémoire utilisée pour la représentation des paires pointées est organisée en *cellules*, qui sont composées d'un pointeur gauche et d'un pointeur droit. La représentation en mémoire de [[`(cons` α β`)`]] est un pointeur vers une cellule dont le pointeur gauche est la représentation de α et le pointeur droit est la représentation de β. La représentation des atomes en mémoire est unique, ce qui signifie que les deux pointeurs de la cellule utilisée pour représenter `(a . a)` sont égaux. Par contre, les deux pointeurs de la cellule utilisée pour représenter `((a . b) . (a . b))` ne sont pas nécessairement égaux.

La notation pointée est un moyen simple et concis de mettre en évidence la structure de pointeurs qui représente en mémoire un objet SCHEME non atomique; une notation graphique, plus explicite, sera introduite plus loin. Dans le tableau ci-dessous, les formes de gauche ont pour valeurs des expressions symboliques, représentées à droite de la flèche en notation pointée. Comme nous le verrons plus loin, le système affiche en général cette valeur sous une forme plus concise que la notation pointée.

```
'a                      -->  a
(cons 'a 'b)            -->  (a . b)
(cons 'a '())           -->  (a . ())
(cons 'a (cons 'b 'c))) -->  (a . (b . c))
(list a)                -->  (a . ())
(list a b)              -->  (a . (b . ()))
'(a b c d)              -->  (a . (b . (c . (d . ()))))
'((a b) (c))            -->  ((a . (b . ())) . ((c . ()) . ()))
```

Le point (entouré d'espaces) et les parenthèses représentent l'appariement. Le point sépare donc les deux composants d'une paire pointée. On peut, de manière plus explicite mais moins commode, visualiser la structure de pointeurs. La représentation en mémoire de la situation induite par l'évaluation en séquence des formes `(define x (cons 'a (cons 'b 'c)))`, `(define y x)`, `(define z (cdr y))` et `(define u (car x))` est donnée à la figure 15.

Vu son encombrement, nous n'utiliserons pas dans la suite ce mode de représentation appelé "box-notation"; en fait, la notation pointée est appropriée, pour peu que l'on soit conscient de sa nature arborescente. Les expressions symboliques s'identifient en effet à des arbres binaires dont les feuilles sont des atomes. Par exemple, l'expression symbolique

`(a . ((b . (c . (d . w))) . (((e . (f . x)) . (g . y)) . z)))`,

qui est la valeur de la forme

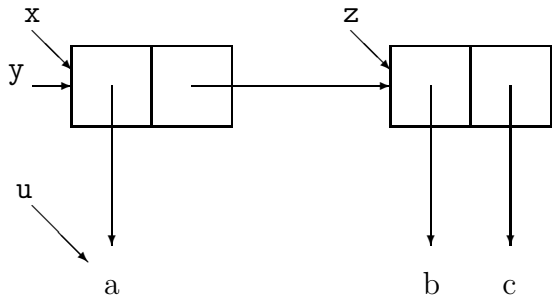


Figure 15: Paires pointées en mémoire

```
(cons a
      (cons (cons b (cons c (cons d w)))
            (cons (cons (cons e (cons f x))
                        (cons g y))
                  z)))
```

correspond à l'arbre binaire de la figure 16.

Remarque. Seules les feuilles de l'arbre de la figure 16 sont explicitement étiquetées. Les étiquettes des nœuds internes sont synthétisées à partir des étiquettes des feuilles et n'existent donc pas en mémoire. Ces étiquettes ont été écrites ici uniquement pour montrer la correspondance entre sous-arbre de gauche et `car` d'une part, entre sous-arbre de droite et `cdr` d'autre part. Il est facile de reconstituer la "box-notation" correspondante.

8.3 Notation pointée et notation usuelle

La notation pointée met en évidence la structure d'*arbre binaire décoré* des expressions symboliques : chaque nœud a zéro fils (feuille) ou deux fils (nœud interne), auxquels on accède par `car` et `cdr`. Chaque feuille est un atome.[91] Tout ceci justifie l'intérêt et l'usage de la notation pointée, mais cette notation reste néanmoins encombrante, notamment dans le cas des listes :

```
()                ()
(0)               (0 . ())
(0 1)             (0 . (1 . ()))
(0 1 2)           (0 . (1 . (2 . ())))
((0 1) 2)         ((0 . (1 . ())) . (2 . ()))
```

On passe très facilement de la notation pointée à la notation usuelle, par le procédé suivant :

[91] Plus exactement, chaque feuille est étiquetée par la représentation d'un atome.

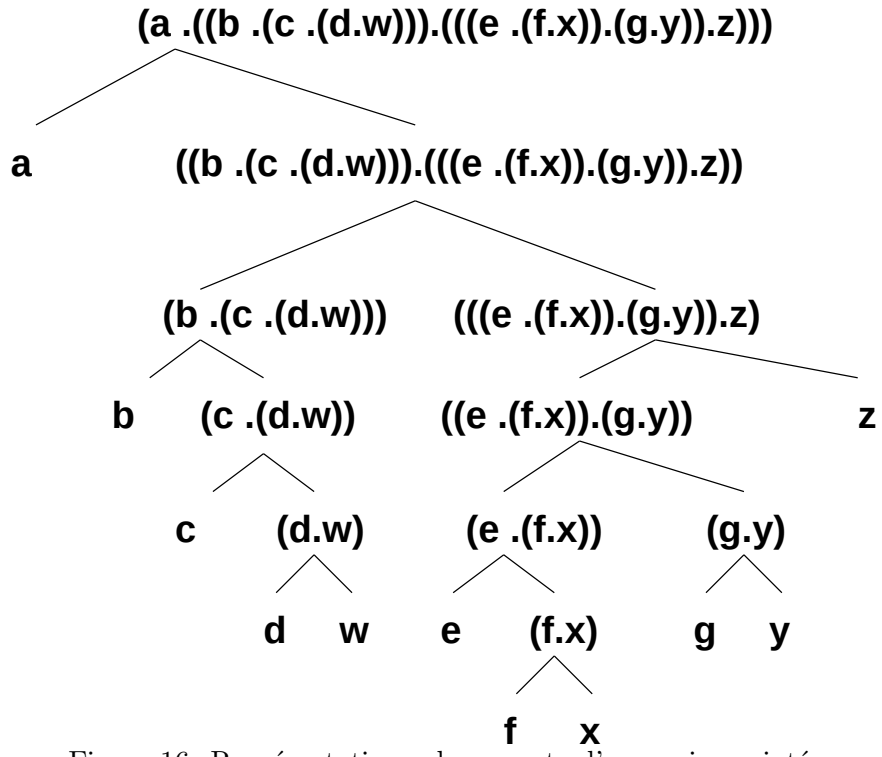


Figure 16: Représentation arborescente d'une paire pointée

Tout point suivi d'une parenthèse ouverte est supprimé,
ainsi que la parenthèse ouverte qui suit
et la parenthèse fermée correspondante.
L'ordre des suppressions est quelconque.

Un point non suivi d'une parenthèse ouverte ne peut pas être supprimé.

Les exemples qui suivent illustrent le procédé de conversion.

```
((0 . (1 . ())) . (2 . ()))
((0 . (1 . ())) . (2     ))
((0    1 . () ) . (2     ))
((0    1 . () )    2      )
((0    1      )    2      )

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

L'élimination des points peut n'être que partielle : un point non suivi d'une parenthèse ouverte n'est pas supprimable. Par exemple, `((a . b) . (c . d))` se simplifie en `((a . b) c . d)` ; c'est sous cette forme que l'évaluateur affichera cette expression symbolique.

8.4 Représentation des listes

Nous avons vu précédemment qu'une liste est conceptuellement un arbre. Nous venons de voir qu'en machine, une liste est, comme toute expression symbolique, représentée par une structure de pointeurs qui est aussi un arbre. Il convient de souligner que ces deux arbres sont bien distincts. En particulier, les nœuds de l'arbre "conceptuel" sont de degré variable tandis que tout nœud interne de l'arbre "machine" est de degré deux.[92] A titre d'exemple, les deux arbres associés à la liste `(a (b c d) ((e f) g))` sont repris à la figure 17.

Rappel. L'information attachée à un nœud interne se déduit de celle attachée à ses descendants; en conséquence, seule l'information attachée aux feuilles doit être explicite.

Remarque. En ce qui concerne la représentation des données en SCHEME, les listes sont représentées par des expressions symboliques. Ce choix, guidé par des raisons d'efficacité, justifie l'emploi de constructeur et accesseurs communs pour ces deux types concrets de données, ainsi que pour les types abstraits (arbres à degré variable et arbres binaires) qui leur correspondent.

8.5 Récursivité structurelle et expressions symboliques

Le schéma de base pour les expressions symboliques découle immédiatement de la structure de l'expression. Une expression symbolique est un atome, ou une paire dont les deux composants sont des expressions symboliques. On a :

```
(define F
  (lambda (s u)
    (if (atom? s)
        (G s u)
        (H (F (car s) (Ka s u))
           (F (cdr s) (Kd s u))
           s
           u))))
```

Dans ce code, `u` représente une liste de zéro, un ou plusieurs arguments supplémentaires.[93] Le cas où il n'y a pas d'argument supplémentaire se récrit plus simplement comme suit :

```
(define F
  (lambda (s)
```

[92] Un degré constant permet une économie de mémoire.

[93] Il y a là un léger abus de notation, d'ailleurs évitable (cf. § 5.3).

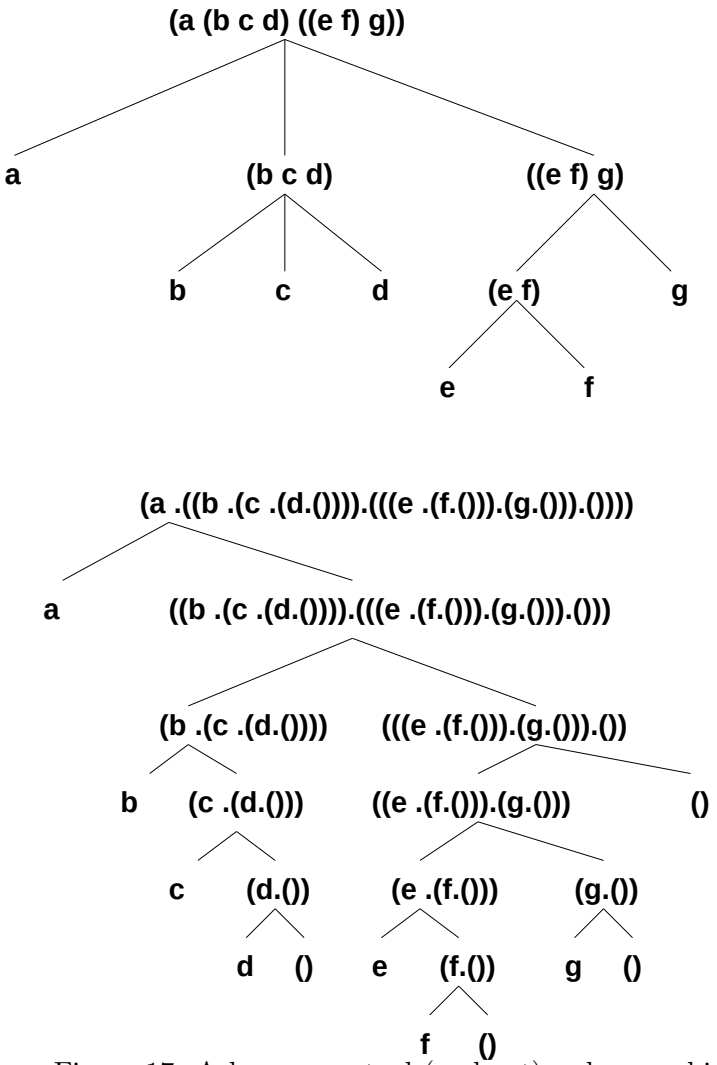


Figure 17: Arbre conceptuel (en haut), arbre machine (en bas)

```
(if (atom? s)
    (G s)
    (H (F (car s)) (F (cdr s)) s))))
```

On peut évaluer la taille d'une expression symbolique par le nombre d'atomes qu'elle contient, ou par le nombre de nœuds internes, ou encore par le nombre total de nœuds, au moyen des fonctions suivantes :

```
(define size_a
  (lambda (s)
    (if (atom? s)
        1
        (+ (size_a (car s)) (size_a (cdr s))))))
```

```
(define size_i
  (lambda (s)
    (if (atom? s)
        0
        (+ 1 (size_i (car s)) (size_i (cdr s))))))

(define size_t
  (lambda (s)
    (if (atom? s)
        1
        (+ 1 (size_t (car s)) (size_t (cdr s))))))
```

Ces programmes sont des instances du schéma structurel, de même que le programme qui renvoie la liste des atomes d'une expression symbolique :

```
(define flatten
  (lambda (s)
    (if (atom? s)
        (list s)
        (append (flatten (car s))
                (flatten (cdr s))))))
```

La technique de l'accumulateur permet d'améliorer l'efficacité de ce programme, par élimination de `append` :

```
(define flatten_a
  (lambda (s a)
    (if (atom? s)
        (cons s a)
        (flatten_a (car s)
                   (flatten_a (cdr s) a)))))
```

On a [[`(flatten_a s a)`]] = [[`(append (flatten s) a)`]] quelles que soient l'expression symbolique [[`s`]] et la liste [[`a`]]. En particulier, on a aussi [[`(flatten_a s '())`]] = [[`(flatten s)`]]. On observe que le code de `flatten_a` est basé sur la récursivité structurelle mais n'est pas une instance du schéma (qui n'autorise pas les appels récursifs imbriqués).

L'aplatissement est une opération qui s'applique à tout type d'arbre. A une liste sont associés un arbre machine que l'on peut aplatir par `flatten`, et un arbre conceptuel que l'on peut aplatir par `flat_list` (§ 5.4). Les deux frondaisons sont naturellement différentes. Soit `l` une liste telle que

`((a . (b . ())) . ((c . (d . ())) . ()))` = [[`l`]] = `((a b) (c d))`

On a

```
(flatten l)  ==>  (a b () c d () ())

(flat_list l)  ==>  (a b c d)
```

Un problème intéressant consiste à déterminer si une expression symbolique peut s'écrire sans point ou non. Le schéma structurel permet d'écrire :

```
(define point-free?
  (lambda (s)
    (if (atom? s)
        #t
        (and (point-free? (car s))
             (list? (cdr s))
             (point-free? (cdr s))))))
```

Cette solution n'est pas optimale parce que [[`(cdr s)`]] est parcouru deux fois, mais on y remédie en utilisant `pair?` plutôt que `list?`. On peut aussi utiliser l'égalité [[`(if c #t e)`]] = [[`(or c e)`]] pour obtenir

```
(define point-free
  (lambda (s)
    (or (atom? s)
        (and (point-free (car s))
             (or (null? (cdr s)) (pair? (cdr s)))
             (point-free (cdr s))))))

(point-free 'a)               ==>  #t
(point-free '((a . b)))       ==>  #f
(point-free '(a . (b . ())))  ==>  #t
(point-free '(a b . c))       ==>  #f
```

8.6 Egalité, identité

Divers prédicats permettent de tester l'égalité de deux valeurs d'un type donné. On utilise `eq?` pour les symboles atomiques, `=` pour les nombres,[94] et `eqv?` pour des valeurs atomiques (symboles, nombres, booléens). Pour les expressions symboliques, on utilise le prédicat prédéfini `equal?`, qui aurait pu être défini comme suit :

```
(define equal?
  (lambda (x y)
    (cond ((pair? x)
           (and (pair? y)
                (equal? (car x) (car y))
```

[94] Le nombre d'arguments de `=` et des autres prédicats de comparaison numérique est quelconque

```
            (equal? (cdr x) (cdr y))))
       ((pair? y) #f)
       (else (eqv? x y)))))
```

On a par exemple

```
(equal? '(a . (b . c)) '(a b . c))  ==>  #t
```

L'usage du prédicat `eq?` s'étend aux expressions symboliques mais il teste alors l'identité de deux objets : `(eq? x y)` sera vrai si `x` et `y` désignent la même structure en mémoire. Concrètement, `eq?` teste l'égalité de deux pointeurs. La session suivante montre bien la distinction entre les deux prédicats.

```
(eq? 'a 'a)          ==>  #t
(define x '(a . b))  ==>  ...
(define y '(a . b))  ==>  ...
(define z x)         ==>  ...
(equal? x y)         ==>  #t
(eq? x y)            ==>  #f
(eq? x z)            ==>  #t
```

Utiliser `eq?` plutôt que `equal?` pour des objets non atomiques peut permettre d'améliorer l'efficacité ... mais cela exige une bonne connaissance du système. Notons aussi que les symboles atomiques n'ont jamais qu'une seule représentation en mémoire; sur ce domaine, `eq`? et `equal`? sont toujours logiquement équivalents.

8.7 Déconstruction des expressions symboliques

Toute expression symbolique s'obtient en combinant des atomes au moyen du seul constructeur `cons` et ce, d'une manière unique. Décrire, ou "déconstruire", une expression symbolique est reconstituer l'unique combinaison (basée sur `cons` et des atomes) dont la valeur est cette expression. On peut aisément produire une fonction `describe` produisant cette reconstitution. On aura par exemple

```
          (cons 1 (cons (cons (cons 'a '()) (cons 2 '())) '()))
      ==> (1 ((a) 2)))

(describe '(1 ((a) 2)))
      ==> (cons 1 (cons (cons (cons 'a '()) (cons 2 '())) '()))
```

Le schéma habituel fonctionne immédiatement :

```
(define describe
  (lambda (s)
    (if (atom? s)
```

```
        (cond ((number? s) s)
              ((boolean? s) s)
              ((null? s) (quote '()))
              ((symbol? s) (list 'quote s))
              (else s))
        (list 'cons
              (describe (car s))
              (describe (cdr s))))))
```

On peut réorganiser le code en

```
(define describe
  (lambda (s)
    (cond ((null? s) (quote '()))
          ((symbol? s) (list 'quote s))
          ((pair? s) (list 'cons
                           (describe (car s))
                           (describe (cdr s))))
          (else s))))
```

Le langage SCHEME, comme tout langage, comporte des mécanismes d'entrée-sortie. Nous ne les avons guère évoqués jusqu'ici parce que la boucle "read-eval-print" du système prévoit la lecture de l'expression à évaluer et l'écriture de sa valeur (et/ou de messages éventuels). On peut cependant prévoir, au cours du processus de calcul, des lectures et des écritures supplémentaires. Nous avons vu que lors de l'évaluation d'une combinaison, l'ordre dans lequel les constituants de cette combinaison sont évalués n'est pas imposé. Un moyen d'imposer un ordre séquentiel, utilisé notamment pour les commandes explicites d'entrée-sortie, est fourni par la forme spéciale `begin`. Le processus de calcul associé à (`begin` e_1 ... e_n) consiste en l'évaluation *en séquence* de $e_1, \ldots, e_n$; la valeur de e_n (si elle existe) est la valeur de la forme. Par exemple l'évaluation de

```
(begin (display "tra")
       (newline)
       (display "la la")
       'inutile
       'utile)
```

provoquera l'affichage

```
tra
la la
==>  utile
```

Les formes spéciales `begin`, `newline` et `display` (acceptant comme argument une chaîne de caractères),[95] permettent d'écrire une variante "pretty-printing" de la fonction `describe` :

[95] Une chaîne de caractères commence et finit par le délimiteur " (guillemet). La forme spéciale `display` permet l'affichage d'une chaîne de caractères, sans les délimiteurs. Les chaînes de caractères constituent

```
(define lcons 6) ;; longueur de "(cons "

(define dis      ;; (dis n) ecrit n blancs
  (lambda (n)
    (if (zero? n)
        (display "")
        (begin (display " ") (dis (- n 1))))))

(define disp
  (lambda (n m) (dis (+ n (* m lcons)))))

(define pretty
  (lambda (d n m)
    (if (not (pair? d))
        (begin (disp n 0)
               (if (symbol? d) (display "'"))
               (display d))
        (begin (display "(cons ")
               (pretty (car d) 0 (+ m 1))
               (newline)
               (disp n (+ m 1))
               (pretty (cdr d) 0 (+ m 1))
               (display ")")))))

(define print
  (lambda (d) (begin (newline)
                     (pretty d 0 0))))
```

La fonction principale `print` écrit son argument avec une indentation correcte, au départ d'une nouvelle ligne. L'évaluation de `(pretty d n m)` provoque l'affichage correctement indenté de la valeur de `d`, sans passage initial à une nouvelle ligne et avec un décalage de 6[[`m`]]+[[`n`]] espaces. Ces fonctions ne retournent pas de valeur; seul leur effet d'affichage est intéressant. Voici quelques exemples d'utilisation.

```
(print 'a)
'a
==> ...

(print '(a . (b . 1)))
(cons 'a
      (cons 'b
            1))
==> ...
```

un type de données SCHEME, avec constructeurs, accesseurs, etc. Nous n'étudierons pas ce type plus avant, ni d'ailleurs les diverses autres formes spéciales d'entrée-sortie. Quelques autres cas seront évoqués plus loin, dans des exemples.

L'exemple suivant illustre la structure d'une liste, vue comme un cas particulier d'expression symbolique :

```
(print '(a b c d))
(cons 'a
      (cons 'b
            (cons 'c
                  (cons 'd
                        ()))))
==> ...

(print '((a . (b . 1)) . ((c . 2) . ((d . 3) . e))))
(cons (cons 'a
            (cons 'b
                  1))
      (cons (cons 'c
                  2)
            (cons (cons 'd
                        3)
                  'e)))
==> ...
```

Remarque. A titre de facilité syntaxique, le système SCHEME permet, dans certaines circonstances, l'emploi "implicite" de la forme `begin`. Nous précisons cet usage implicite dans deux cas seulement. On peut utiliser cette facilité dans les clauses de la forme spéciale `cond` (cf. § 3.5.1); par exemple,

```
(cond (pred (begin e1 e2 e3))
      (else (begin e4 e5)))
```

peut s'écrire

```
(cond (pred e1 e2 e3)
      (else e4 e5))
```

De même, on peut utiliser le `begin` implicite dans le corps d'une λ-forme; l'expression

```
(lambda (x y) (begin e1 e2 e3))
```

peut s'écrire

```
(lambda (x y) e1 e2 e3)
```