Passeports pour Scheme

et l'Algorithmique

Pierre Castéran, Robert Cori
Département Informatique
Université Bordeaux 1

Partie I

Eléments de base de la programmation.

Chapitre 1

Expressions, Variables, Fonctions

Ce chapitre est destiné à la familiarisation avec quelques constructions du langage Scheme. D'abord nous présentons la syntaxe des expressions, qui est une syntaxe préfixe; cette particularité peut dérouter certains utilisateurs du langage car elle remet en question des habitudes acquises depuis leur plus jeune âge dans les notations du calcul arithmétique. Il ne faut pas lui accorder trop d'importance. On ferait une lourde erreur en donnant une place trop grande à cet aspect du langage, il ne s'agit en fait que d'une notation. Bien plus importantes sont les notions de variable et de fonction qui sont introduites dans les paragraphes suivants. Le concept de variable, qui n'est pas le même en mathématiques et en informatique, se retrouve dans tous les langages de programmation sous une forme voisine. La notion de fonction joue un rôle primordial en Scheme, c'est ce qui fait l'originalité et l'intérêt de ce langage car il permet d'exprimer des compositions de fonctions et des opérations sur celles ci de façon très élégante qui le rapproche du langage mathématique.

1.1 Expressions Préfixées

Revenons sur les notations couramment utilisées en mathématiques pour écrire des formules, elles ne sont guère homogènes. Ainsi dans,

$$\ln(x)$$

la fonction logarithme (ln) précède l'opérande sur lequel elle agit, même chose pour sin, cos, par contre la fonction valeur absolue s'écrit

$$|x|$$

et le symbole de l'opération est situé à la fois à droite et à gauche de l'opérande. Pour des opérations qui s'effectuent sur deux opérandes, les symboles sont souvent situées entre les deux nombres ou variables, mais le signe de la division est lui placé de façon horizontale curieuse

$$a+b \qquad a-b \qquad \frac{a}{b}$$

enfin pour une fonction f non précisée de deux variables on reprend la notation consistant à écrire le symbole de fonction avant les variables :

$$f(x,y)$$

.

1.1.1 Notation des expressions en Scheme

En Scheme on a choisi de normaliser les notations en prenant comme convention que le symbole d'une opération s'écrit toujours avant ses opérandes; ce symbole et les variables sont séparés par des espaces, de plus une expression est entourée de parenthèses, ainsi

```
(+ 3 4)
```

est la façon de demander à l'interprète Scheme d'évaluer $3+4$. On a de même les expressions :

```
(* 5 12)
(/ 18 3)
(sin 3.14159)
(log 1)
(sqrt 64)
```

qui expriment respectivement $5.12, 18/3, sin(\pi), ln 1, \sqrt{64}$.
Pour exprimer des calculs plus compliqués il faut utiliser la règle suivant laquelle on peut remplacer, dans une expression, un nombre par une expression bien formée. Ainsi une expression composée en Scheme est toujours de l'une des formes :

$$\boxed{\begin{array}{l}(\mathbf{op}\ e)\\ (\mathbf{op}\ e_1\quad e_2)\\ (\mathbf{op}\ e_1\quad e_2\ldots\quad e_k)\end{array}}$$

Où **op** est un opérateur agissant sur un nombre dans le premier cas (on dit que c'est un opérateur *unaire*), sur deux nombres dans le second (opérateur *binaire*), sur k nombres ($k \geq 0$) dans le troisième et où les e_i sont des expressions composées ou des nombres.
Ainsi, on peut construire des expressions faisant intervenir plusieurs opérations par imbrication, on utilise pour cela des opérateurs et des nombres. Un opérateur peut agir sur plusieurs nombres, on dit qu'il est d'*arité* k s'il agit sur k nombres. Les fonctions usuelles d'une variable sont d'arité 1, par exemple $\sin, \cos, \log, \exp$. Les opérations arithmétiques comme l'addition, la soustraction, la multiplication, la division sont en général considérées comme d'arité 2, toutefois il faut remarquer que le symbole `-` désigne aussi bien une opération d'arité 1 (l'opposé d'un nombre) et une opération d'arité 2 (la soustraction). En Scheme on considère les opérations arithmétiques usuelles `(+ * - /)` comme d'arité quelconque, par exemple :

```
(+ 1 3 5 7 9 11)
```

signifie : $1 + 3 + 5 + 7 + 9 + 11$.

Pour construire des expressions on utilise la définition récursive suivante :

Définition. Une *expression* est :

- ou bien un nombre
- ou bien de la forme `(op e1 e2 ... ek)` où `e1, e2, ... ek` sont des expressions et où op est un opérateur d'arité $k > 0$.

Exemple. Ceci permet de définir des expressions quelconques en Scheme comme par exemple :

```
(+ 3 (* 2 4) (/ 12 3)  (- 15 4))
```

qui exprime :

$$3 + 2 * 4 + 12/3 + (15 - 4)$$

1.1.2 Représentation des nombres

Les nombres peuvent être de plusieurs sortes :

- Des nombres entiers écrits en notation décimale et pouvant être précédés d'un signe qui peut être le signe `+` ou le signe `-`on a par exemple :

  `3 56 +78 -456 112421342152345624356`

- Des nombres réels dits en *virgule fixe* écrits suivant la notation habituelle et dans laquelle la virgule est remplacée par un point. Exemples :

  `3.14159 -1.414 -123.5`

- Des nombres dits *flottants* représentés par une mantisse m et un exposant u, dans ce cas la notation :

  $$m \texttt{ e } u$$

  où `m` est un décimal et `u` est un entier, représente le nombre décimal $m.10^u$. Ainsi `12e3` représente 12000, `1.345e2` représente $134,5$ et `25e-3` signifie $0,025$.

Cette distinction entre nombres entiers, nombres décimaux en virgule fixe et nombre décimaux flottants existe dans tous les langages de programmation. On peut noter que l'interpréteur Scheme permet d'utiliser des entiers aussi grands que l'on veut, mais que les nombres décimaux ont une précision limitée.

1.1.3 Opérateurs

Nous donnons ici une liste des opérateurs principaux sur les entiers et les nombres décimaux que l'on peut utiliser en Scheme.

fonction	symbole en Scheme	arité
Addition	+	quelconque
Multiplication	*	quelconque
Soustraction	-	quelconque
Division	/	quelconque
Logarithme	`log`	1
Exponentielle	`exp`	1
Sinus	`sin`	1
Cosinus	`cos`	1
Tangente	`tan`	1
Arc sinus	`asin`	1
Arc cosinus	`acos`	1
Arc tangente	`atan`	1
Racine carrée	`sqrt`	1
Puissance	`expt`	2
Valeur absolue	`abs`	1
Division entière	`quotient`	2
Reste de la division	`remainder`	2
Modulo	`modulo`	2
Partie entière par excès	`ceiling`	1
Partie entière par défaut	`floor`	1
Entier le plus voisin	`round`	1

Remarques.
1. Les opérateurs `modulo` et `remainder` donnent le même résultat sur les entiers positifs, la différence se manifeste uniquement lorsque l'on considère les nombres négatifs.

```
> (modulo 13 4)
1
> (remainder 13 4)
1
> (modulo 13 -4)
-3
> (remainder 13 -4)
1
> (modulo -13 4)
3
> (remainder -13 4)
-1
> (modulo -13 -4)
```

```
-1
> (remainder -13 -4)
-1
```

2. La division de deux nombres entiers, qui n'est pas toujours un nombre entier, est souvent présentée par l'interpréteur Scheme sous la forme d'une fraction irréductible :

```
> (/ 42 28)
3/2
>
```

3. Les opérateurs d'arité quelconque sont évalués sans ambiguïté lorsqu'ils sont associatifs (par exemple `+` et `*`) par contre il faut manipuler avec précaution les opérateurs non associatifs (`-` et `/`).

1.1.4 Règle d'évaluation

Lorsqu'on fournit une expression à l'interpréteur Scheme celui ci effectue une évaluation. Il obéit à une règle d'évaluation, appelée *Evaluation par valeur*, elle précise que tous les arguments d'un appel de fonction sont évalués **avant** l'application de l'opération de nom $\langle pr \rangle$.
Les étapes de l'évaluation de ($\langle pr \rangle$ $\langle e_1 \rangle$ $\ldots \langle e_n \rangle$) sont les suivantes :

1. Evaluation des arguments de $\langle pr \rangle$
   - calcul de $\epsilon_1 =$ Evaluation de $\langle e_1 \rangle$
   - calcul de $\epsilon_2 =$ Evaluation de $\langle e_2 \rangle$
   - ...
   - calcul de $\epsilon_n =$ Evaluation de $\langle e_n \rangle$
   - evaluation de la fonction représentée par le symbole $\langle pr \rangle$
2. Application de celle-ci aux arguments formés par le n-uplet $\langle \epsilon_1, \epsilon_2, \ldots \epsilon_n \rangle$

Notation.
La valeur obtenue par l'évaluation de l'expression e est notée dans la suite par : $\mathrm{Eval}[\![\langle e \rangle]\!]$.
Remarque.
L'ordre dans lequel les étapes marquées "$-$" sont effectuées n'est pas important; par contre il est clair que l'étape 1 doit précéder l'étape 2. On distingue, comme pour les nombres le *nom* $\langle pr \rangle$ de la fonction pr associée.

Exemple
Considérons l'expression `(+ 3 (sqrt 2) (* 2 3 4))`. Son évaluation se décompose de la façon suivante :

Evaluation de `(+ 3 (sqrt 2) (* 2 3 4))`

1. – $\boxed{\texttt{3} \stackrel{0}{\Longrightarrow} 3}$
   – Evaluation de `(sqrt 2)`
     (a) $\boxed{\texttt{2} \stackrel{0}{\Longrightarrow} 2}$
     (b) $\sqrt{2} = 1.414$
   – Evaluation de `(* 2 3 4)`
     (a) – $\boxed{\texttt{2} \stackrel{0}{\Longrightarrow} 2}$
         – $\boxed{\texttt{3} \stackrel{0}{\Longrightarrow} 3}$
         – $\boxed{\texttt{4} \stackrel{0}{\Longrightarrow} 4}$
     (b) $2 \times 3 \times 4 = 24$
2. $3 + 1.414 + 24 = 28.414$

En bref, nous venons de décomposer la relation

$$\texttt{(+ 3 (sqrt 2) (* 2 3 4))} \stackrel{0}{\Longrightarrow} 28.414$$

1.1.5 Erreurs

Certaines évaluations peuvent conduire à un arrêt du calcul sur une *erreur*, les principales causes possibles d'erreur sont les suivantes.

Nombre d'arguments

Une fonction a une *arité*, ne pas respecter cette arité conduit à une erreur. Les expressions suivantes conduisent à une erreur de ce type :

– `(sqrt 3 4)` (la fonction *racine carrée* admet un seul argument)

– `(log)` (la fonction *logarithme* admet un argument)

– `(modulo 8)` (la fonction *modulo* admet deux arguments)

Types des arguments

Outre l'arité, chaque fonction prend ses arguments dans un domaine précis, il est impossible de calculer par exemple la somme d'un nombre et d'un booléen (voir chapitre 2, pour la définition des booléens).

Restriction des domaines

Ceci est une notion classique; en effet une fonction numérique peut très bien ne pas être définie pour tout nombre, voici quelques expressions dont l'évaluation conduit à une erreur de ce type :

- `(sqrt -8)`
- `(log 0.0)`
- `(/ 6 (- 3 (+ 2 1)))`

Portée des erreurs

Si l'évaluation d'une sous-expression provoque une erreur, aucune valeur n'est retournée, et l'évaluation de l'expression entière est abandonnée. Par exemple, dans l'expression `(* 5 (log 0.0))`, la multiplication par 5 n'est jamais effectuée.

1.2 Exemple de session Scheme

Nous pouvons désormais utiliser une machine pour tester l'écriture et l'évaluation des expressions. Concrètement, dès le début d'une *session Scheme*, vous interagissez avec l'interpréteur Scheme sous la forme d'une "boucle Top-Level" ou "boucle Read Eval Print" (R-E-P), répétition *ad libitum* des étapes suivantes :

1. Scheme envoie au terminal une chaîne de caractères : l' *invite* (ou le *prompt*); dans ces notes le prompt sera la suite de caractères : `>` .
2. Vous tapez une expression $\langle exp \rangle$.
3. Si l'expression $\langle exp \rangle$ est *syntaxiquement* correcte, Scheme tente de l'évaluer .
4. Si l'évaluation se passe bien, soit ϵ la valeur obtenue : Scheme associe à cette valeur une *forme imprimée* qui est affichée sur votre console.

1.2.1 Réalisation

Nous donnons ci-dessous un exemple de session Scheme en reprenant les exemples ci-dessus, à vous de pratiquer un peu ce langage en essayant d'autres exemples :

```
> 12
```
12

```
> -5656519987987
```
-5656519987987

```
> 6/8
```
3/4

```
> 1234.89
```
1234.89

```
> (+ 3 (sqrt 2) (* 2 3 4))
```
28.414

> (<= 2 (sqrt 2)) ;$2 \leq \sqrt{2}$

#f

```
> (sqrt 3 4)
```
Error : Incorrect number of arguments to sqrt : (3 4)

```
> (log -2)
```
Error : Argument of Log must be positive : -2

1.2.2 Remarques sur l'écriture des expressions

Espaces

La forme `(` $\langle pr \rangle$ $\langle e_1 \rangle$ $\ldots \langle e_n \rangle$ `)` s'appelle une *liste*; les caractères `(` et `)` sont des *délimiteurs*; les constituants de l'expression : $\langle pr \rangle$, $\langle e_1 \rangle$ … sont séparés par des *espaces*, c'est à dire des suites de caractères formées l'espace proprement dit, le caractère de tabulation ou le changement de ligne.

Disposition sur plusieurs lignes

La disposition d'une expression peut être améliorée si l'on adopte la règle suivante :

– Soit une expression tient sur une seule ligne

– Soit tous les arguments d'une fonction commencent sur la même colonne..

un exemple correct est :

```
(+ (* 3 4 5)
   (- (* 3 4)
      (sqrt (log 78))))
```

La même expression correctement écrite, mais peu lisible :

```
(+ (* 3 4 5) (- (* 3 4)
(sqrt (log 78))))
```

Remarquons que les éditeurs plein écran de type Emacs ont souvent un mode Scheme (ou Lisp) contenant un système pour *présenter* correctement les expressions, c'est à dire organiser les positions en début de ligne pour rendre l'expression lisible.

Commentaires

Ils sont destinés à faciliter à un lecteur *humain* la compréhension des programmes et expressions Scheme, un commentaire est une suite de caractères compris entre un point-virgule (caractère ;) et une fin de ligne. Les commentaires sont ignorés lors de l'évaluation des expressions Scheme, par exemple :

```
> (/ (* 1 2 3 4 5 6 7 8 9) ; 9!
     (* (* 1 2 3 4) ; 4!
        (* 1 2 3 4 5))) ;  5!
```

126

1.2.3 Remarques sur la notation préfixée

Certains dénigrent cette notation qui vient du langage *LISP* qu'ils qualifient de *Lots of Insipid and Stupid Parentheses* au lieu de *List Programming.*

Si la notation Lisp provoque une inflation de parenthèses, remarquons qu'elle possède certains avantages certains.

non ambiguïté :

Pour toute expression, il existe une et une seule décomposition en un couple :

$$< \text{opérateur, suite d'opérandes} >$$

Remarquons qu' en notation mathématique, une expression comme "`2+3*4`" doit sa décomposition unique à la règle de priorité suivant laquelle la multiplication s'effectue avant l'addition.

Comparons les expressions :

```
3+5*8 mod 5
```

et la version Lisp :

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

Dans un cas il faut connaître les priorités respectives de `+`, `*` et `mod`, alors que ce n'est pas nécessaire dans une notation *complètement parenthésée.*

lisibilité :

Du point de vue d'un lecteur humain, cette notion est parfois subjective; tout dépend des habitudes de l'utilisateur. Il est à noter que la notation parenthésée que nous utilisons permet la construction d' éditeurs de programmes qui pallient ces problèmes (EDWIN sur IBM-PC, mode Scheme sur GnuEmacs).

simplicité :
Il est souhaitable d'avoir *peu* de règles pour définir la syntaxe des expressions. De ce point de vue, la notation mathématique classique est désastreuse (règles de priorité, syntaxe spécifique pour chaque opérateur).

cohérence :
Tous les opérateurs et constructions du langage ont exactement la même forme appelée *liste*, ceci autorise une puissance de calcul permettant de traiter un programme comme une donnée d'un autre programme, ce qui est très difficile à réaliser dans les langages de programmation comme Pascal, Ada, C, Fortran ou encore moins Basic.

1.3 Variables

Les mathématiques utilisent des noms symboliques pour désigner soit des paramètres (dont la valeur est déjà connue ou qui sera spécifiée en cours du calcul), des inconnues (dont la valeur doit être déterminée à partir d'équations), des variables à l'occasion de définition de fonctions (dont la valeur est quelconque à l'intérieur du domaine de définition de la fonction). Dans ce paragraphe nous considérons des variables informatiques dont la signification se rapproche plutôt du concept mathématique de paramètre, cette notion existe dans presque tous les langages de programmation. Nous verrons dans un autre chapitre la notion de variable intervenant dans une définition de fonction, celle ci est assez particulière au langage Lisp et ses dérivés (par exemple Scheme). Par contre la notion d'inconnue courante en mathématique n'existe pas dans les langages de programmation usuels (sauf Prolog).

1.3.1 Définition

Nous considérons ici qu' à toute variable est attribuée une *valeur* de préférence au moment de sa *définition* ou au plus tard au moment d'un calcul où elle intervient. L'attribution d'une valeur à une variable se fait à l'aide de `define` en Scheme, l'instruction :

$$\boxed{\texttt{(define a } \langle\ expr\ \rangle\texttt{)}}$$

consiste à donner la valeur de l'expression $< expr >$ à la variable `a`, celle ci ne changera pas de valeur tant qu'un nouveau `define` ne lui aura pas attribué une autre valeur. Une fois qu'une valeur a été donnée à une variable celle ci peut être utilisée pour attribuer une valeur à une autre variable. Par exemple :

```
> (define a 23)
```

```
a
> (define b (* 2 a))
b
> b
46
```

Mais attention, une nouvelle définition de la variable `a` ne changera pas la valeur attribuée aux variables définies à partir de `a`, ceci est un peu déroutant au début car cela contredit l'intuition acquise en mathématiques. Ainsi après la suite de définitions précédentes si on donne une nouvelle valeur à la variable `a` celle de la variable `b` ne changera pas :

```
> (define a 10)
a
> b
46
```

Pour bien comprendre ce qui se passe en machine lors de la définition d'une variable il faut se référer à la notion de mémoire et d'environnement.

1.3.2 Un modèle de la mémoire

Nous présentons ci-dessous un modèle simple et intuitif de l'organisation de la mémoire dans un modèle de machine compatible avec l'utilisation de Scheme. C'est ce modèle qui nous permettra de définir l'utilisation des variables en Scheme.

Les places

On considère que les valeurs manipulées au cours des évaluations sont stockables dans des *cellules*, ou *places*. Formellement, considérons un ensemble **L** (pour l'anglais "locations") dont les éléments sont appelés *places*. Les noms que nous choisirons pour les places seront des lettres grecques : α, α', α_1 ... Un *rangement* est une application partielle de **L** dans un ensemble de valeurs E.

Intuitivement un rangement consiste à ranger des valeurs dans certaines places, nous ne voulons pas formaliser à l'excès cette notion de rangement, mais considérons qu'une place peut ou non *contenir* une valeur. Nous considérons aussi une notion intuitive de *place libre* ou *non encore utilisée*

Exemple

Voici un exemple de rangement :

place	contenu
α_0	12
α_1	782.67
α_2	454654654654
α_3	3.5e12
α_4	

Notations

Dans le rangement ci-dessus, à la place α_4 n'est associée aucune valeur, on dira que α_4 est *non affectée*.

Une notation graphique pour les cellules peut être la suivante :

$$\boxed{\alpha_0 \mid 12}\ ,\ \boxed{\alpha_1 \mid 782.17}\ ,\ \boxed{\alpha_3 \mid 3.5e12}\ ,\ \boxed{\alpha_4 \mid \quad}$$

Si α est une place, alors $\lfloor\alpha\rfloor$ dénote le contenu de α, par exemple $\lfloor\alpha_0\rfloor = 12$ et $\lfloor\alpha_4\rfloor$ est indéfinie.

1.3.3 Environnements

Les *environnements* permettent de faire la liaison entre les variables des expressions Scheme et le modèle de la mémoire, ils associent à toute variable d'une session Scheme une *place* ou sa *valeur* est *rangée*. On dit que l'on a défini une fonction ρ de *rangement*.

Exemple

Un environnement ρ peut être représenté graphiquement par une table, indiquant pour chaque variable la place où elle est rangée :

$\langle var\rangle$	$\rho[\![\langle var\rangle]\!]$
`x`	α_0
`y`	α_1
`z`	α_2
`u`	α_3
`w`	α_4

Le couple formé d'un environnement et d'un rangement pourra être directement représenté par :

$\langle var\rangle$	$\rho[\![\langle var\rangle]\!]$
`x`	$\boxed{\alpha_0 \mid 12}$
`y`	$\boxed{\alpha_1 \mid 782.17}$
`z`	$\boxed{\alpha_2 \mid 454654654654}$
`u`	$\boxed{\alpha_3 \mid \#t}$
`w`	$\boxed{\alpha_4 \mid \quad}$

1.3.4 (Re)définition de l'évaluation

La définition de Eval présentée dans un paragraphe précédent est restreinte aux expressions sans variables autres que les symboles de fonctions primitives. Nous devons étendre le mécanisme d'évaluation de façon à autoriser les expressions avec variables. Ceci s'effectue simplement de la façon suivante :

> toute évaluation d'une variable dans un environnement ρ donne pour résultat la valeur contenue dans la place qui lui est associée par ρ. Il y a erreur si aucune place ne lui est associée ou si la place qui lui est associée est *vide* (ne contient aucune valeur).

L'évaluation de l'expression e dans un environnement ρ est notée $\mathrm{Eval}[\![\langle e \rangle]\!]_\rho$ par la suite.

1.3.5 Exemples d'évaluation

Considérons l'expression $\langle e_0 \rangle$ = `(* y (+ x y))` et un environnement ρ_1 défini par la table :

$\langle var \rangle$	$\rho_1[\![\langle var \rangle]\!]$
`x`	$\alpha_{89} \mid 6$
`y`	$\alpha_{789} \mid 5$
`+`	$\alpha_{+} \mid +$
`*`	$\alpha_{*} \mid *$

Détaillons les étapes du calcul de

$$\mathrm{Eval}[\![\langle e_0 \rangle]\!]_{\rho_1} = 55$$

Evaluation de `(* y (+ x y))` dans ρ_1

1. – $\boxed{\mathrm{Eval}[\![*]\!]_{\rho_1} = *}$
   – $\boxed{\mathrm{Eval}[\![\mathtt{y}]\!]_{\rho_1} = \lfloor \alpha_{789} \rfloor = 5}$
   – $\boxed{\mathrm{Eval}[\![\mathtt{(+\ x\ y)}]\!]_{\rho_1} = \ldots = 11}$
2. $\mathrm{Eval}[\![\mathtt{(*\ y\ (+\ x\ y)}]\!]_{\rho_1} = \ldots = 55$

L'évaluation de l'expression

```
(+ (* u 10) 5)
```

dans ρ_1 produit une erreur de type *variable indéfinie* car la place associée à la variable `u` est vide.

1.3.6 Retour à Scheme

Les concepts vus ci-dessus permettent d'aborder la définition de variables en Scheme.

Initialisation

Au début d'une session Scheme, un environnement global ρ_g est créé, et donc tous les symboles de primitives sont liés à la fonction qui leur est naturellement associée. Nous faisons l'hypothèse suivante, qui ne sera contredite qu'au moment de l'étude précise des implantations de Scheme, mais qui n'entre pas en conflit avec son utilisation :

> L'environnement global ρ_g associe une place vide à toute variable possible.

Cela implique que presque toutes les cellules $\rho_g[\![\langle v \rangle]\!]$ sont vides.

La commande `define`

L'évaluation de l'expression `(define` $\langle var \rangle$ $\langle exp \rangle$ `)` provoque :

- **si** le résultat de l'évaluation de $\langle exp \rangle$ est ϵ, **alors** la cellule $\rho_g[\![\langle var \rangle]\!]$ a pour nouveau contenu la valeur ϵ
- **si** l'évaluation de $\langle exp \rangle$ ne donne pas de résultat, une erreur est provoquée, et la cellule $\rho_g[\![\langle var \rangle]\!]$ conserve son ancien contenu.

Remarques

Soit $\langle var \rangle$ une variable, une commande
`(define` $\langle var \rangle$ $\langle exp \rangle$ `)`
fait perdre les anciennes liaisons de $\langle var \rangle$ vers les variables qui ont été définies à partir d'elle. En conséquence, nous donnons les conseils suivants :

> Ne **re-**définir une variable que pour corriger ou améliorer une définition précédente erronée.
>
> Ne pas redéfinir une variable liée à une primitive, comme par exemple les opérateurs donnés dans le tableau du paragraphe 1.

En ce qui concerne le dernier point, certaines réalisations de Scheme refusent ou ne prennent pas en compte des re-définitions du genre suivant qui sont formellement à proscrire en raison des confusions auxquelles elles peuvent aboutir.

```
(define + *)
```

Un exemple d'utilisation

On veut calculer $1+\sqrt{2+\sqrt{3+\sqrt{5}}}$

```
> (define x1 5.0)
x1
> (define x2 (+ 3 (sqrt x1)))
x2
> (define x3 (+ 2 (sqrt x2)))
x3
> (define x4 (+ 1 (sqrt x3)))
x4
> x4
3.07080796098304
```

Remarquons dans cet exemple que les variables `x1`, `x2`, `x3` sont utilisées comme *variables auxiliaires* pour le calcul de `x4`. Cela ne pose aucun problème **à condition que l'utilisateur n'ait pas déjà défini ces variables pour d'autres usages**. C'est là un problème de sécurité que nous pallierons par l'usage de variables locales; ce concept sera vu dans un chapitre suivant.

1.4 Fonctions

Dans ce paragraphe on étudie la possibilité offerte en Scheme de définir de nouvelles fonctions à l'aide de fonctions prédéfinies. La syntaxe s'apparente à celle de la définition de variables c'est à dire que l'on utilise la forme `define`. Plus précisément une définition de fonction se présente sous la forme :

```
(define (f x)
⟨exp⟩)
```

Cette forme exprime le moyen de calcul de $f(x)$, par l'expression $\langle$ exp $\rangle$. Une représentation de ce qui se passe ne machine lors de la définition d'une fonction consiste à considérer qu'une place est associée à la variable `f` représentant une fonction et que le contenu de cette place est l'expression permettant de calculer $f(x)$. L'évaluation de l'expression n'est effectuée que lorsque qu'un calcul effectif est demandé. Par exemple, si on souhaite connaître la valeur de la fonction f au point a, on écrit `(f a)` alors `a` est évaluée, puis l'expression décrivant f au point a l'est aussi.

1.4.1 Utilisation d'une fonction

Une fois une fonction définie on peut l'utiliser soit pour calculer la valeur de cette fonction en certains points soit pour définir d'autres variables ou d'autres fonctions. Par exemple on a la session suivante en Scheme, où on définit la fonction $f(x)$ par

$$f(x) = \log\sqrt{x^2+1}$$

on calcule $f(0), f(100000), f(2)$, puis on pose

$$a = f(-1) \qquad g(x) = f(x-1) \qquad h(x) = f(x) - x$$

Enfin on demande à l'interprète Scheme les valeurs de :

$$g(1), g(0), h(0), h(1000), a$$

.

```
> (define (f x) (log (sqrt (+ (* x x) 1))))
f
> (f 0)
0.0
> (f 10000)
9.210340376976182
> (f 2)
0.8047189562170503
> (define a (f -1))
a
> (define (g x) (f (- x 1)))
g
> (define (h x) (- (f x) x))
h
> (g 1)
0.0
> (g 0)
0.3465735902799727
> (h 0)
0.0
> (h 1000)
-993.0922442210181
> a
0.3465735902799727
```

1.4.2 Fonctions à plusieurs variables

On peut aussi définir des fonctions à plusieurs variables en modifiant légèrement la notation précédente.

$$\boxed{\texttt{(define (f x1 x2 ... xk)} \langle\texttt{exp}\rangle\texttt{)}}$$

Dans cette définition $\langle$`exp`$\rangle$ représente une expression faisant intervenir les variables `x1, x2, ... xk`. On peut alors à l'aide d'une fonction à plusieurs variables définir de nouvelles fonctions en fixant la valeur d'une ou plusieurs variables. On a par exemple la session suivante au cours de laquelle on définit la fonction :

$$f(x, y, z) = xy + yz + zx + xyz$$

puis

$$g(x) = f(x, x, x) \qquad\qquad h(x, y) = f(x, y, 1)$$

soit : $g(x) = 3x^2 + x^3$ et $h(x, y) = 2xy + x + y$

```
> (define (f x y z)
          (+ (* x y ) (* y z) (* x z) (* x y z))
   )
f
> (f 1 1 1)
4
> (f 1 0 0 )
0
> (f 2 2 2)
20
> (define (g x) (f x x x))
g
> (g 1)
4
> (g 2)
20
> (define (h x y) (f x y 1))
h
> (h 1 1)
4
> (h 2 3)
17
>
```

1.4.3 Conclusion

Nous avons vu dans ce paragraphe une méthode particulièrement simple et concise pour définir des fonctions, de les composer, de calculer la valeur d'une fonction en un point, de définir des fonctions de plusieurs variables. Nous verrons dans un chapitre suivant la notion de fonctionnelle qui généralise la notion de fonction. Une fonctionnelle est une opération dont la donnée ou le résultat est une fonction, cette possibilité de définir des fonctionnelles augmente encore le degré d'abstraction du langage et permet des développements puissants. Ceci est d'un intérêt majeur dans un langage de programmation et permet de classer Scheme dans la famille des langages fonctionnels, famille très utile dans la programmation d'applications modernes de l'Informatique : les systèmes experts, les systèmes de calcul formels, par exemple.

Chapitre 2

Fonctions numériques récursives

Dans ce chapitre on introduit le principe fondamental de la programmation en langage Scheme: *la récursivité*. Cette façon de programmer consiste à exprimer la solution d'un problème qui dépend de certaines variables à l'aide de la solution du même problème pour une valeur plus petite des variables. Il faut de plus, pour que le calcul récursif se termine, indiquer la solution pour la plus petite valeur prise par celles ci. La récursivité s'apparente ainsi à la notion mathématique de récurrence et les suites numériques récurrentes constituent l'exemple de base de programmation récursive.

2.1 Récursivité et itération

2.1.1 Factorielle

Un exemple instructif consiste à écrire une fonction calculant factorielle n. Il y a plusieurs façons de décrire une méthode pour effectuer ce calcul. Une première consiste à dire:

Pour calculer factorielle n on multiplie entre eux tous les nombres $1, 2, \ldots n$.

Cette première façon n'est pas récursive, on dit qu'elle est itérative. Cela revient à écrire:

$$n! = \prod_{i=1}^{n} i$$

En Scheme on ne peut pas exprimer simplement l'itération comme dans d'autres langages, comme par exemple C ou Pascal. Toutefois on peut utiliser la méthode récursive permettant d'exprimer le calcul, cette méthode peut s'exprimer par :

Si $n = 0$ le résultat est 1, sinon $(n \geq 1)$ on calcule factorielle $n - 1$ et on multiplie ce résultat par n.

Cette façon revient d'un point de vue mathématique à dire que factorielle n est donnée par la suite récurrente u_n suivante:

$$u_0 = 1, \quad \text{et pour } n > 1, \; u_n = n.u_{n-1}$$

On remarque que cette forme récursive est très répandue dans le langage mathématique et dans beaucoup de cas elle est bien plus claire que la forme itérative. On l'écrit en Scheme:

```
(define (fact n)
        (if (= n 0) 1
            (*  (fact (- n 1)) n)
        )
)
```

Cette forme est de fait une traduction quasi immédiate en Scheme de la définition mathématique.

2.1.2 Suites récurrentes à un terme

Le calcul de toutes les suites récurrentes à un terme, c'est à dire des suites dans lesquelles on définit le terme général par:

$$\boxed{u_1 = a, \quad \text{et pour } n > 1, \ \ u_n = \Phi(u_{n-1}, n)}$$

s'exprime très simplement en Scheme sur le même modèle que le calcul de factorielle n. Il suffit en effet d'écrire:

```
(define a ...)

(define (PHI x y) ...)


(define (u n)
        (if (= n 1) a
                   (PHI  (u (- n 1)) n)
       )
)
```

Les exemples de suites récurrentes à un terme sont très nombreux, citons l'exponentiation x^n qui est donnée par la suite

$$\boxed{u_1 = x, \quad \text{et pour } n > 1, \ \ u_n = x.u_{n-1}}$$

Ce qui se traduit en Scheme par:

```
(define (puissance x n)
        (if (= n 1) x
                    (* x (puissance  x (- n 1)) )
       )
)
```

2.1.3 Sommes de termes

On peut calculer aussi sur ce même principe des sommes de la forme:

$$S_n = \sum_{i=1}^{n} u_i$$

Il suffit pour cela de remarquer que S_n est donnée par:

$$S_1 = u_1 \qquad \text{et} \quad S_n = u_n + S_{n-1}$$

ce qui donne en Scheme:

```
(define (S n)
        (if (= n 1) (u 1)
            (+  (S (- n 1)) (u n))
       )
)
```

Le calcul de cette somme peut servir à trouver des valeurs moyennes, calcul très fréquent dans les applications. Il suffit en effet de diviser la somme par le nombre de termes.
Un autre exemple est celui où on recherche le maximum des valeurs prises par une suite u_i pour i variant entre 1 et n. On note cette fonction $Max(u, n)$ ou `(Maximum u n)` en Scheme, sa valeur est u_1 si $n = 1$ et le plus grand des deux nombres $Max(u, n-1)$ et u_n pour $n > 1$. On a ainsi:

```
(define (Maximum u n)
        (cond ((= n 1) (u 1))
              ((< (Maximum  u (- n 1)) (u n)) (u n))
              (else (Maximum u (- n 1)))
       )
)
```

En introduisant la fonction `Plusgrand`, on obtient une écriture plus élégante et plus efficace:

```
(define (Plusgrand a b)
        (if (< a b) b a ))

(define (Maximum u n)
        (if (= n 1) (u 1)
            (Plusgrand  (Maximum u (- n 1)) (u n))
       )
)
```

Le calcul de la somme des éléments et du maximum ont des points communs car il s'agit dans les deux cas d'appliquer une opération associative, l'addition et le maximum respectivement, à tous les termes d'une suite numérique.

2.2 Evaluation d'une fonction récursive

Dans un langage de programmation, une *fonction récursive* est une fonction F dont la définition conduit à un calcul qui effectue un ou des appels à la fonction F elle même. Cette façon peut surprendre, il faut pour comprendre son caractère licite se reporter aux exemples donnés par les suites récurrentes.
La question se pose alors de savoir ce qui se passe en machine lors du calcul d'une fonction récursive. L'évaluation de F nécessite dans ce cas un appel de F avec une autre valeur des variables et les évaluations successives de F s'emboîtent ainsi l'une dans l'autre.
Pour comprendre ce qui se passe nous détaillons l'évaluation de l'expression `(fact 4)` dans l'exemple de défintion de factorielle n donné ci dessus. Cette évaluation va demander celle de `(fact 3)`, laquelle va demander celle de `(fact 2)` puis celle de `(fact 1)` en enfin `(fact 0)` qui n'appelle aucune demande. Ensuite le résultat va être rendu de `(fact 1)` à `(fact 2)` puis de `(fact 2)` à `(fact 3)` et enfin de `(fact 3)` à `(fact 4)`. Cet enchaînement peut se représenter par le schéma suivant:

> Evaluation de `(fact n)` avec $n = 4$:
>
> > Evaluation de `(fact n)` avec $n = 3$
> >
> > > Evaluation de `(fact n)` avec $n = 2$
> > >
> > > > Evaluation de `(fact n)` avec $n = 1$
> > > >
> > > > > Evaluation de `(fact n)` avec $n =$
> > > > > 0
> > > > >
> > > > > Résultat: $= 1$
> > > >
> > > > Résultat: $1 \times 1 = 1$
> > >
> > > Résultat: $2 \times 1 = 2$
> >
> > Résultat: $3 \times 2 = 6$
>
> Résultat: $4 \times 6 = 24$

Le fait de bien comprendre ce qui se passe lors de l'évaluation d'une fonction récursive n'aide pas particulièrement pour la conception d'une telle fonction. Lorsque l'on est amené à écrire une fonction récursive il vaut mieux penser à la notion mathématique qui consiste à exprimer la solution d'un problème à partir de sa solution pour une valeur plus petite des variables. Toutefois une vision correcte de l'évaluation d'une telle fonction est indispensable pour trouver les raisons du non-fonctionnement de certains programmes.
Ainsi par exemple une fonction récursive $f(n)$ qui est définie en fonction de $f(n+1)$ ne peut que conduire à un calcul qui ne termine pas car des appels successifs vont s'emboîter et à aucune étape le calcul n'atteindra une situation où la valeur de $f(n)$ peut être calculé sans un nouvel appel. Cette situation pathologique est appelée bouclage. L'arrêt de l'évaluation d'un calcul qui boucle peut poser parfois des problèmes et sur certains micro ordinateurs on peut être amené à se résoudre à éteindre l'appareil puis à redémarrer le système.

2.3 Calculs récursifs non élémentaires

2.3.1 Récurrence à deux termes

Les suites récurrentes à deux termes sont de même nature que celles à un terme, pour en programmer le calcul on peut procéder de la même façon, il faut toutefois indiquer les valeurs prises par les deux premiers termes de la suite. Un exemple célèbre est constitué par la suite dite de Fibonacci qui est donnée par:

$$f_n = f_{n-1} + f_{n-2} \qquad f_0 = 1, f_1 = 1$$

Cela s'écrit en Scheme par:

```
(define (f n)
        (if (or (= n 0) (= n 1)) 1
            (+  (f (- n 1)) (f (- n 2)) )
       )
)
```

Plus généralement le calcul d'une suite de la forme:

$$\boxed{u_0 = a, \quad u_1 = b \quad \text{et pour } n > 1, \quad u_n = \Phi(u_{n-1}, u_{n-2}, n)}$$

Se traduit en Scheme par:

```
(define a ...)
(define b ...)


(define (PHI x y z) ...)


(define (u n)
        (cond ((= n 0) a)
              ((= n 1) b)
              (else  (PHI  (u (- n 1)) (u (- n 2)) n)
       )
)
```

La définition de ces suites à deux termes pose le problème de l'efficacité de la programmation. L' efficacité de la forme donnée ci-dessus est très mauvaise car le nombre d'appels récursifs lors de l'évaluation d'une telle définition est très grand ce qui donne lieu à des temps de calcul prohibitifs. On peut tenter de calculer la valeur de la suite de Fibonnacci pour $n = 30$ afin de se convaincre de l'ampleur des temps de réponse. On verra comment améliorer cette efficacité dans un prochain chapitre.

2.3.2 Exemple non élémentaire

Dans beaucoup d'exemples le calcul d'une fonction ne s'exprime pas directement sous une forme récursive simple. Il faut alors trouver une formulation récursive du problème qui permette d'écrire une fonction Scheme sans trop de mal.
Considérons par exemple la fonction:

$$f(0) = 0 \text{ et } f(n) = k \text{ si } n = 2^k.m \text{ où } m \text{ n'est pas divisible par } 2$$

Il est alors clair que $f(n)$ ne peut pas s'exprimer à l'aide de $f(n-1)$ car l'une des deux quantités est toujours égale à 0 et l'autre prend une valeur quelconque. Par contre on peut remarquer que si n est un nombre pair ($n = 2m$), alors on a $f(2m) = f(m) + 1$ et si n est impair alors $f(n) = 0$. Cette fonction s'écrit donc en Scheme:

```
(define (f n)
        (cond ((= n 0) 0 )
              ((= (modulo n 2) 0) (+ 1 (f (/ n 2))) )
              (else  0)
       )
)
```

2.3.3 Plus grand diviseur commun

Un autre exemple de calcul récursif est le classique algorithme du plus grand commun diviseur attribué à Euclide. Le principe repose sur l'observation suivante:
Si b divise a, le pgcd de a et b est b, sinon le pgcd de a et b est aussi celui de b et du reste de la division de a par b.

La terminaison pour tout couple $(a, b), a \geq b > 0$ de nombres positifs du processus de calcul découle du fait que le reste de la division de a par b est toujours strictement plus petit que b. Cette remarque prouve aussi que le nombre d'appels de la fonction est toujours plus petit que b. On écrit cet algorithme en Scheme:

```
(define (pgcd a b)
        (if (= (modulo a b) 0 )
            b
            (pgcd b (modulo a b))
        )

)
```

2.3.4 Nombres premiers

Dans certains cas, la récursivité ne permet pas d'exprimer directement le calcul d'une fonction. Il faut alors savoir introduire une nouvelle fonction. Cette fonction peut contenir un nouveau paramètre, elle s'exprimera simplement sous forme récursive et elle permettra de calculer la première en donnant une valeur particulière à ce paramètre.

Un exemple classique est le prédicat `(premier n)` qui est égal à `#t` si le nombre n est premier et qui est égal à `#f` sinon. Il est totalement exclus de définir ce prédicat récursivement car rien ne relie la primalité de n avec celle de $n-1$, ou d'un nombre plus petit que n. On est ainsi amené à introduire une autre fonction $divise - plus - petit(nx)$ qui est égale à `#t` si le nombre n est divisible par un entier compris au sens large entre 2 et x et qui est égale à `#f` sinon. On a alors

$$premier(n) = not(divise - plus - petit(n, \sqrt{n}))$$

$divise - plus - petit$ est un prédicat qui s'exprime bien de façon récursive ainsi en Scheme:

```
(define (divise-plus-petit a b)
        (cond ((= b 1) #f )
              ((= (modulo a b) 0 ) #t)
              (else (divise-plus-petit a (- b 1)))
        )

)

(define (premier a)
    (if (= a 1) #f
        (not (divise-plus-petit a (floor (sqrt a))))
    )
)
```

2.3.5 Dichotomie

La technique de résolution consistant à remplacer un intervalle de recherche par un sous intervalle de taille deux fois plus petite est une situation où la récursivité s'impose. C'est le cas, par exemple, pour la recherche d'une solution approchée de l'équation

$$f(x) = 0$$

où f est une fonction continue, lorsque l'on a déjà trouvé deux valeurs a et b, $(a < b)$, telles que:

$$f(a)f(b) < 0$$

l'algorithme consiste alors à partir de l'intervalle $[a, b]$, à chercher le milieu c de cet intervalle, tester si $f(a)f(c) < 0$, dans ce cas remplacer $[a, b]$ par $[a, c]$, sinon remplacer $[a, b]$ par $[b, c]$, et répéter cette opération jusqu'à ce que $b - a$ soit plus petit que ϵ, la valeur de la précision cherchée. Cette technique se traduit simplement en Scheme par:

```
(define (resoudre f a b epsilon)
    (define c (/ (+ a b) 2))
        (cond ((< (-  b a) epsilon)   c)
              ((< (* (f a) (f c)) 0) (resoudre f a c epsilon))
              (else                  (resoudre f c b epsilon))
        )

)
```

2.4 Problèmes posés par la récursivité

Plusieurs questions se trouvent posées lorsque l'on écrit une fonction récursive la première est la question de la terminaison, la seconde celle de l'efficacité. Nous examinerons ces questions plus en détail dans un prochain chapitre. Signalons toutefois qu' il n'est pas du tout immédiat de vérifier pour une fonction récursive donnée si le calcul termine quelques soient les valeurs de la variable. Voici quelques exemples qui doivent faire réfléchir.
Une fonction qui de manière évidente boucle indéfiniment si $n > 0$, elle utilise un appel à $f(n+1)$:

```
(define (f n)
        (if (= n 0) 1
            ( + 1 (f (+ n 1)))
        )
)
```

Une fonction qui donne un résultat pour tout entier n en utilisant pourtant un appel à $f(n+1)$:

```
(define (f n)
        (if (= (modulo n 2) 0) n
            ( + 1 (f (+ n 1)))
        )
)
```

Une fonction qui ne se termine pas lorsque n est divisible par 3 et termine sinon:

```
(define (f n)
        (cond ((= (modulo n 3) 0) (+ 1 (f (- n 3))))
              ((= (modulo n 3) 2) (+ 1 n))
              (else  ( + 1 (f (+ n 1))))
        )
)
```

Une fois que l'on a pu démontrer qu'une fonction récursive donnée termine son calcul pour toutes les valeurs possibles des variables, le travail de conception d'un algorithme n'est pas terminé car se pose alors la question du temps mis par le calcul. On parle alors de *complexité* de l'algorithme; on verra des exemples de fonctions pour lesquelles une solution trop hâtive donne lieu à un algorithme très long alors qu'il est possible de trouver des solutions rapides en se donnant un peu de mal. Tel est le cas pour l'exemple du calcul des suites de Fibonnacci donné plus haut car le calcul de $f(n)$ va systématiquement refaire le calcul de $f(n-2)$ même si cette dernière valeur avait déjà été obtenue pour le calcul de $f(n-1)$ ce qui se traduira par des temps de calcul prohibitifs.

Chapitre 3

Les listes

Dans ce chapitre nous introduisons la structure de données de base du langage Scheme. Il s'agit de la structure de *liste*. Une liste permet de travailler sur un nombre quelconque d'objets : plusieurs nombres, plusieurs variables, plusieurs chaînes de caractères. Jusqu'ici nous n'avons pu définir de fonctions que pour un nombre limité de variables donné à l'avance. Grâce aux listes ce nombre pourra être quelconque, à priori aussi grand que l'on veut, la seule limitation est dans la taille de la mémoire de l'ordinateur dont nous disposons. Sur les listes trois opérations fondamentales sont définies à partir desquelles toutes les opérations peuvent être construites. On peut bien sur définir des fonctions récursives sur les listes, c'est l'outil indispensable. De la même façon que l'on a utilisé le nombre 0 comme base de l'induction sur les nombres, on utilise la liste vide pour le même usage sur les listes. Les trois opérations de base vont servir à raccourcir et à construire des listes, elles constituent des équivalents aux opérations de soustraction et d'addition sur les entiers.

3.1 Eléments de base

3.1.1 Formalisation

Une *liste plate* est composée d'une parenthèse ouvrante, d'une suite d' *atomes* séparés par des espaces et d'une parenthèse fermante. Dans ce chapitre, le mot liste signifiera liste plate, la notion de liste généralisée sera introduite et étudiée dans un prochain chapitre. Nous ne pouvons donner la définition précise de ce qu'est un atome car cela nous conduirait trop loin dans les considérations abstraites. Pour les besoins de ce cours nous pouvons nous limiter à des notions simples et considérer que les objets suivants sont des atomes :

- Les nombres entiers ou décimaux.
- Les valeurs booléennes `#t, #f`
- Les chaînes de caractères (entourées de guillemets "et ")

– Les symboles, dont nous verrons plus loin l'utilisation.

– Les opérations, les fonctions prédéfinies et les fonctions définies par le programmeur.

Remarque sur les symboles: Les symboles sont des objets constitués d'une suite de caractères, par exemple `x rouge var` sont des symboles. La manipulation de ceux-ci nécessite une certaine attention. Le prédicat `symbol?` teste si une valeur est un symbole. La règle suivante est importante :

> *lorsque l'on utilise un symbole comme une constante il faut faire précéder la suite de caractères qui le représente d'une apostrophe,* (`quote` en anglais).

Noter qu'un symbole non précédé d'une apostrophe est pris comme une variable et donc est évalué. Si aucune valeur ne lui a été attribuée cela se conclut en général par un message d'erreur. La session suivante illustre ces notions :

```
> (define x 4)
x
> (define y x)
y
> y
4
> (define var 'x)
var
> var
x
> (symbol? x)
 \false
> (symbol? 'x)
#t
> (symbol? var)
#t
```

Le nombre d'atomes composant une liste est appelé sa *longueur*. La seule liste de longueur 0 est la *liste vide* elle est notée **#f** en Scheme. La notion mathématique qui se rapproche le plus des listes plates est celle de suite finie. Dans une suite, comme dans une liste, on peut répéter plusieurs fois le même élément et l'ordre dans lequel les éléments interviennent est pris en compte.

Remarque On peut définir une notion de liste généralisée pour laquelle on s'autorise à emboîter des listes les unes à l'intérieur des autres. Cette notion sera définie et étudié dans un prochain chapitre, celui-ci est consacré à une étude en détail des listes plates qui permettent déjà de réaliser un grand nombre de fonctions.

3.1.2 Utilisation en Scheme

Pour pouvoir utiliser une liste en Scheme on utilise l'ordre :

`(define x '( a b c d .....))`

où `a b c d ....` sont des atomes. Avec ce que nous avons vu jusqu'ici sur les listes nous pouvons simplement définir une liste, faire afficher son contenu (il suffit pour cela de taper au clavier le nom de cette liste), tester si une liste est vide (ceci se fait à l'aide du prédicat prédéfini `null?`) et demander sa longueur à l'aide de la fonction `length`. Un exemple très simple de session est alors le suivant :

```
> (define x '(a b c d))
X
> (define y '(1 5 67 23 456 8))
Y
> (define z ' \false )
Z
> (null? x)
#f
> (null? z)
#t
> (length y)
6
> x
(a b c d)
> (< (length x) (length y))
#t
```

La suite du chapitre montrera des notions moins rudimentaires sur les listes, puis au cours des derniers chapitres du cours on verra tout le parti que l'on peut tirer de ces notions.

3.1.3 Les trois opérations de base sur les listes : car, cdr, cons

Ces trois opérations sont appelées ainsi pour des raisons historiques. Les deux premières reprennent les codes des instructions du langage de la première machine sur laquelle le langage Lisp a été réalisé, la dernière est simplement l'abréviation de *construire*. La puissance de ces trois opérations est parfois surprenante puisque l'on peut réaliser toutes les autres par combinaison de celles-ci. Toutefois en se rappelant que l'on peut exprimer toutes les opérations sur les nombres entiers simplement à l'aide des fonctions successeur (addition de l'entier 1) et prédécesseur (soustraction de l'entier 1) cette puissance devient moins étonnante.

Les opérations `car, cdr` d'arité 1 ne sont définies que pour des listes non vides. La fonction `car` associe à la liste `x` son premier élément `(car x)`, `(cdr x)` est la liste obtenue à partir de `x` en supprimant ce premier élément. Ainsi `(cdr x)` est la liste vide si `x` est une liste ne comportant qu'un seul élément. On fait souvent l'erreur de croire que si la liste `x` ne comporte que deux éléments alors `(cdr x)` est le dernier élément de cette liste, de fait

`(cdr x)` est dans ce cas une liste comportant un seul élément. On a l'habitude de composer les fonctions `car, cdr` pour retrouver des sous listes ou des éléments qui se trouvent au milieu d'une liste. Pour cela une abréviation pratique consiste a écrire `(cadr x)` au lieu de `(car (cdr x))` pour désigner le deuxième élément de la liste `x`, plus généralement si `y,z,t` désignent l'une des lettres `a` ou `d` alors `(cyztr x)` est une forme abrégée de l'imbrication des fonctions : `(cyr(czr(ctr x)))`.
La fonction `cons` est une fonction d'arité 2 qui construit une liste à partir d'un élément `a` et d'une liste `x`, en insérant `a` en première position dans la liste `x`, c'est en quelque sorte l'opération inverse de `car` et `cdr` puisque l'on a les relations dans lesquelles `a` est un élément et `x` est une liste non vide :

`(cons (car x) (cdr x))` $=$ `x`

`(car (cons a y))` $=$ `a` `(cdr (cons a y))` $=$ `y`

Comme exemple d'utilisation de ces fonctions on peut considérer la session Scheme suivante :

```
> (define x '(paris bordeaux pau))
X
> (define y '(amsterdam bruxelles lille))
Y
> (define z '(bayonne burgos madrid seville tanger))
Z
> (car x)
paris
> (cons (car y) x)
(amsterdam paris bordeaux pau)
> (cdr z)
(burgos madrid seville tanger)
```

3.1.4 Remarques

1. Les combinaisons des fonctions `car, cdr` permet de retrouver n'importe quel élément à l'intérieur d'une liste, mais la notation est souvent un peu lourde même si l'on utilise la forme abrégée donnée plus haut.
2. La fonction `cons` permet aussi de construire des listes de taille très grande avec la même lourdeur que celle indiquée précédemment.
3. Les relations suivantes relient les longueurs de `x`, `(cdr x)`, et `(cons a x)` :

`(length (cdr x))` $=$ `(length x) - 1`
`(length (cons a x ))` $= 1 +$ `(length x)`

4. Lors de l'utilisation de `(cons a x)`, `a` doit nécessairement un atome et `x` une liste. Si par mégarde on utilise une variable `a` qui est une liste alors il n'y aura pas d'erreur détectée par l'interprète mais on construira une liste généralisée, ce que l'on ne souhaite pas nécessairement faire, rappelons que cette notion sera étudiée dans un chapitre suivant. Par contre, utiliser pour `x` une variable qui n'est pas une liste conduit à une erreur même dans le contexte des listes généralisées.

3.2 Construire des listes

La fonction `cons` permet de construire des listes mais elle n'est pas toujours très pratique, en particulier lorsque l'on veut concaténer deux listes, c'est à dire fabriquer une liste à partir de deux autres en les mettant " bout à bout " . Dans ce cas c'est la fonction `append` qu'il faut utiliser.

3.2.1 Les fonctions append et list

La fonction `append` d'arité 2 concatène deux listes; elle construit à partir des listes `x,y` la liste `(append x y)` constituée de la suite des éléments de `x` suivie de ceux de `y`. Ainsi, la longueur de `(append x y)` est égale a la somme des longueurs des deux listes `x` et `y`. La liste vide constitue un élément neutre pour l'opération `append`, ainsi on obtient la liste `x` lorsque l'on effectue `(append x y)` ou bien `(append y x)`, si `y` est la liste vide. La fonction `list` est surtout utile pour construire des listes généralisées, on peut aussi l'utiliser pour des listes plates pour lesquelles elle constitue une opération similaire à `'` ainsi on peut remplacer :

```
> (define x '(a b c d))
```

par :

```
> (define x (list 'a 'b 'c 'd))
```

avec un résultat tout à fait identique.

3.2.2 Exemples.

Voici une session Scheme qui complète la session précédente

```
> (define x '(paris bordeaux pau))
X
> (define y '(amsterdam bruxelles lille))
Y
> (define z '(bayonne burgos madrid seville tanger))
Z
> (append y x)
(amsterdam bruxelles lille paris bordeaux pau)
> (append x y)
(paris bordeaux pau amsterdam bruxelles lille)
> (append y x z)
(amsterdam bruxelles lille paris bordeaux pau bayonne burgos madrid seville tanger)
```

```
> (list x y)  ; erreur de manipulation
((paris bordeaux pau) (amsterdam bruxelles lille))
> (list (car x) (car y))
(paris amsterdam)
```

3.2.3 L'apostrophe : quote

L'opération consistant à mettre une apostrophe devant un symbole ou une liste déroute souvent le débutant. Sa signification est d'empêcher l'interprète d'évaluer la valeur de l'objet en question et de le considérer comme un atome ou une liste plutôt que comme une expression dont il faut évaluer la valeur. Par exemple lorsque l'on écrit en Scheme :

```
> (define a 7)
```

la variable `a` est un nombre, lorsque l'on demande la valeur de `a` à l'interprète celui-ci va répondre 7 par contre si on lui demande la valeur de `'a` la réponse sera `a` :

```
> 'a
a
> a
7
```

Le même phénomène se produit pour une liste les deux variables `b` et `c` définies ci dessous sont respectivement une liste et un nombre suivant que l'on mette ou que l'on ne mette pas une apostrophe devant l'expression qui les défini :

```
> (define b '( + 2 3))
b
> b
(+ 2 3)
> (define c (+ 2 3))
c
> c
5
> (list? c)
 \false
> (list? b)
#t
```

Remarque. On met une apostrophe devant un nom de variable ou devant une liste lorsque l'on ne souhaite pas voir évaluer ce nombre ou cette liste. Une apostrophe mise devant une expression empêche l'évaluation de tous les symboles figurant dans l'expression.
Ainsi la fonction `list`, qui construit une liste formée des valeurs de ses arguments, commence par évaluer ceux-ci sauf s'ils sont précédés d'une apostrophe. La session suivante montre un exemple illustrant cette remarque;

```
> (define a 4)
a
> (define b #t)
b
> (define c "truc")
c
> (list a b c)
```

Ici les arguments sont évalués et la réponse est :

```
(4 #t "truc")
```

Par contre pour :

```
> (list 'a 'b 'c)
```

Les symboles ne sont pas évalués et l'interprète répond :

```
(a b c)
```

3.3 Fonctions récursives usuelles sur les listes

On peut définir comme on l'a fait pour les fonctions numériques des fonctions récursives dont les données sont des listes, il faut alors commencer la base de la récurrence par la liste vide qui devient l'équivalent de 0 ou 1 pour les entiers. La fonction `cdr` est alors souvent utilisée pour diminuer la taille de la donnée et la fonction `cons` parfois utilisée pour construire le résultat.

Un exemple simple est la longueur d'une liste qui est certes disponible avec la fonction prédéfinie `length` mais que l'on peut aussi redéfinir en remarquant que la longueur de la liste vide est 0 et que la longueur de `x` est égale à la longueur de `(cdr x)` augmentée de 1. Cette remarque se traduit directement en Scheme par :

```
>(define (longueur x)
        (if (null? x) 0
            (+ 1 (longueur ( cdr x)))
        )
 )
```

Le calcul de cette fonction se termine toujours car une suite d'appels successifs de `(cdr x)` finit toujours par retrouver la liste vide.

3.3.1 Appartenance

La fonction qui indique si un élément `a` appartient à une liste `x` s'exprime de manière semblable à celle qui calcule la longueur, en effet si `x` est vide alors `a` n'appartient pas à `x`, sinon il faut comparer `a` à `(car x)`, s'ils sont égaux alors `a` appartient à `x` et s'ils sont différents il faut répéter l'opération avec `(cdr x)`

```
>(define (appartient a x)
        (if (null? x) #f
            (if (eq? (car x) a) #t
                (appartient a (cdr x)))
        )
 )
```

Dans cet exemple on utilise la fonction `eq?` qui teste si deux atomes sont égaux. La base de l'induction est la même que pour la longueur : on donne la réponse si la liste est vide sinon on peut être amené à appliquer la fonction à `(cdr x)`. Comme ceci diminue la taille de la liste sur laquelle on applique la fonction on peut en déduire que le calcul se terminera pour toute liste `x` et tout élément `a`.

3.3.2 Recherche du i-eme élément

La recherche d'un élément d'une suite connaissant son rang dans celle ci est une opération particulièrement utile, noter que cette opération est immédiate lorsque l'on utilise un vecteur plutôt qu'une liste toutefois sa programmation pour une liste est aussi assez simple. Le principe est le suivant : on suppose donné $i \geq 1$ et on cherche le i^{eme} élément de la liste `x`, si cette dernière est vide alors il y a une erreur, si $i = 1$ alors le résultat est `(car x)` sinon il faut chercher le $(i-1)^{\text{eme}}$ élément de la liste `(cdr x)`, cet algorithme s'exprime donc très directement en Scheme :

```
>(define (ieme-element i x)
        (if (null? x)  "erreur liste trop petite"
            (if (= i 1) (car x)
                (ieme-element (- i 1) ( cdr x)))
        )
 )
```

Une fonction particulière peut être écrite pour rechercher le dernier élément d'une liste, il s'agit en quelque sorte de l'opposée de l'opération `car`. Le dernier élément de `x`, c'est `(car x)` si `(cdr x)` est vide, sinon il faut chercher le dernier élément de `(cdr x)` :

```
>(define (dernier x)
         (if (null? x)  "erreur liste vide"
             (if (null? (cdr x)) (car x)
                 (dernier ( cdr x)))
         )
 )
```

On peut aussi écrire la fonction duale de `cdr` qui consiste à supprimer le dernier élément d'une liste, cette fonction construit une liste et réduit la donnée à chaque appel récursif :

```
>(define (supprimer-dernier x)
         (if (null? x) "erreur liste vide"
             (if (null? (cdr x)) ' \false
                 (cons (car x) (supprimer-dernier ( cdr x))))
         )
 )
```

3.3.3 Reverse

Un grand classique de la programmation sur les listes consiste à écrire la fonction qui calcule l'image miroir d'une liste, c'est à dire une fonction qui construit la liste obtenue à partir d'une liste `x` en en inversant l'ordre des éléments. On appelle généralement cette fonction `reverse`. Par exemple on a :

```
>(reverse '(1 2 3 4 5 6))
(6 5 4 3 2 1)
```

Plusieurs solutions sont possibles la première consiste à utiliser la fonction `cons` appliquée au dernier élément de `x` et l'image miroir de`(supprimer-dernier x)`, en débutant la récursivité sur la liste vide; on obtient ainsi la procédure suivante :

```
>(define (reverse1 x)
         (if (null? x) x
             (cons (dernier x) (reverse1 (supprimer-dernier  x)))
         )
 )
```

Une autre façon de procéder consiste à calculer l'image miroir de `(cdr x)` et d' y ajouter à la fin `(car x)`, le calcul se termine lorsque `x` est la liste vide, pour ajouter un élément à la fin d'une liste `x` il faut utiliser la fonction `append` appliquée à `x` et à la liste formée par cet élément :

```
>(define (reverse2 x)
```

```
         (if (null? x) x
             (append  (reverse2 (cdr  x)) (list (car x)))
         )
 )
```

Enfin une troisième façon de procéder consiste à utiliser une fonction auxiliaire itérative des deux variables tt x et `y` qui enlève successivement un élément à la liste `x` pour l'ajouter à la liste `y`.

```
>(define (reverse3 x)
      (define (reverse-aux x y)
          (if (null? x) y
               (reverse-aux (cdr x) (cons (car x) y))))
      (reverse-aux x ' \false )
 )
```

3.3.4 Remplacer

Le remplacement de toutes les occurrences d'un élément `a` d'une liste `x` par un autre élément `b` se définit aussi de façon récursive. On remplace `(car a)` par `b` s'il est égal au symbole `a` à remplacer et on poursuit le remplacement de ce symbole dans `(cdr x)`. Cette fonction s'écrit :

```
>(define (remplacer a b x); remplace a par b dans la liste x
         (if (null? x) x
             (if (eqv? a (car x)) (cons b (remplacer a b (cdr x)))
                 (cons (car x) (remplacer a b ( cdr x))))
         )
 )
```

3.4 Listes de nombres

Certaines fonctions sur les listes sont spécifiques à celles qui ne contiennent que des nombres. Il s'agit par exemple de construire des listes finies de nombres satisfaisant une certaine propriété, de trouver le plus grand élément d'une suite, d'insérer un élément à sa place dans une liste triée de nombres. Nous examinons ces fonction dans ce paragraphe.

3.4.1 Construire la liste des entiers de 1 à n

Pour effectuer cette construction on ajoute en dernier élément l'entier n à la liste formée des entiers de 1 à $n-1$, si $n = 1$ on construit simplement une liste composée du seul nombre 1; ceci s'écrit très directement en Scheme, (mais avec un temps de calcul particulièrement déplorable !) :

```
> (define (liste-entiers  n)
     (if (= 1 n) (list 1)
         (append (liste-entiers (- n 1)) (list n))
     )
 )
```

3.4.2 Construire la liste des nombres premiers

La liste des nombres premiers inférieurs ou égaux à n se construit en utilisant le prédicat `premier` défini au chapitre précédent. Il suffit de modifier la construction de la liste des nombres de 1 à n en testant si n est premier avant de l'ajouter en queue de liste, cet ajout n'est effectué que si n est premier :

```
> (define (liste-premiers  n)
     (if (= 1 n) ' \false
         (if (premier n) (append (liste-premiers (- n 1)) (list n))
             (liste-premiers (- n 1))
         )
     )
 )
```

3.4.3 Utilisation de listes triées

On manipule souvent des listes formées de nombres apparaissant en ordre croissant. Dans ce cas la recherche d'un élément peut être rendue plus efficace en terminant cette recherche dès que l'on a trouvé un élément de la liste plus grand que le nombre à chercher :

```
>(define (appartient-triee a x)
         (cond ((null? x) #f)
               ((= (car x) a) #t)
               ((< a (car x) ) #f)
               (else  (appartient-triee a ( cdr x)))
         )
 )
```

Dans le cas des listes triées l'ajout d'un élément ne peut se faire n'importe comment. Ainsi en utilisant par exemple `cons` pour le mettre en tête comme cela est fait dans des listes quelconques on obtient une liste qui n'est plus triée. Il faut retrouver donc l'endroit précis où l'insérer : on le place en tête s'il est plus petit que `(car x)` ou si la liste `x` est vide sinon il faut reconstruire une liste dont le premier élément est `(car x)` et dont le reste est obtenu par insertion à la bonne place de `x` dans `(cdr x)` ce qui donne en Scheme :

```
>(define (ajout-triee a x)
         (cond ((null? x) (list a))
               ((< a (car x) ) (cons a x))
               (else  (cons (car x) (ajout-triee a ( cdr x))))
         )
 )
```

3.5 Listes et fonctions

On peut construire des listes à partir de fonctions ou faire opérer des fonctions sur des listes.

3.5.1 Liste des valeurs d'une fonction

Pour une fonction f donnée il est parfois utile de connaître la liste des valeurs prises par $f(x)$ pour x prenant des valeurs données par exemple $1, 2 \ldots n$. On construit cette liste en modifiant très légèrement la construction de la liste des nombres entiers $1 \ldots n$, il suffit en effet dans cette fonction de remplacer `n` par `(f n)` :

```
> (define (liste-valeurs f  n)
     (if (= 1 n) (list (f 1))
         (append (liste-valeurs f (- n 1)) (list (f n)))
     )
 )
```

3.5.2 Appliquer une fonction à chaque élément d'une liste

On peut être aussi amené à connaître les valeurs prises par une fonction f pour toutes les valeurs d'une variable contenues dans une liste x. Pour cela il faut construire une liste dont le premier élément est l'application de f sur `(car x)` et dont le reste est obtenu par application récursive de la fonction sur `(cdr x)`. Cette fonction est disponible dans les interprètes de Scheme et s'appelle `map`.

```
> (define (appliquer f  x)
     (if (null? x) ' #f
         (cons (f (car x)) (appliquer f (cdr x)))
     )
 )
```

Une dernière interaction entre listes et fonctions que nous examinerons dans ce chapitre consiste à appliquer une fonction f de deux variables à tous les éléments d'une liste. C'est à dire à partir de la liste $x_1, x_2, \ldots x_n$ calculer :

$$f(f(f(\ldots f(f(x_1, x_2), x_3) \ldots x_{n-1}), x_n)$$

C'est ce qui est fait pour la fonction $f(u, v) = u + v$ lorsque l'on écrit en Scheme :

```
(+ 1 3 5 7 9 11 13 15)
```

Pour réaliser cette opération on utilise l'algorithme suivant : si la liste ne comporte que deux éléments on applique tout simplement f à ceux ci, si la liste est plus grande on remplace les deux premiers éléments de la liste par leur image par f et on réitère cette opération. Ceci donne en Scheme :

```
> (define (appliquer-a-tous f  x)
     (cond ((null? x) (error "nombre d'arguments insuffisant"))
           ((null? (cdr x)) (error "nombre d'arguments insuffisant"))
           ((null? (cddr x)) (f (car x) (cadr x)))
           (else (appliquer-a-tous f (cons (f (car x) (cadr x)) (cddr x))))
     )
 )
```

Partie II

Compléments de Programmation en Scheme

Chapitre 1

Fonctionnelles

Dans ce chapitre nous étudions un des principaux avantages du langage Scheme sur des langages algorithmiques classiques : la possibilité de définir des fonctionnelles. Une fonctionnelle est un opérateur dont une donnée, ou dont le résultat, est une fonction. Celles-ci permettent un traitement informatique élégant de quelques problèmes mathématiques. Dans ce chapitre nous montrons comment utiliser des fonctionnelles, le premier paragraphe présente la notation qui est utilisée pour définir les fonctionnelles dans le cadre simplifié des fonctions, il s'agit de la forme `lambda`. Ce paragraphe peut ne pas apparaître très utile, dans la mesure où nous avons déjà vu comment définir des fonctions avec la forme `define`, en fait il s'agit de bien comprendre d'abord la notation `lambda` sur des exemples simples et bien connus avant de s'attaquer à des situations plus complexes, dans les deux paragraphes suivants.

1.1 Notation sur les fonctions

Un des préliminaires à l'introduction des fonctionnelles est de donner une notation pour les fonctions, c'est à dire un moyen d'exprimer que f est une fonction de la variable x ou des deux variables x et y, ou d'un nombre quelconque de variables. En mathématiques lorsque l'on définit une fonction f on dit :

Soit f la fonction qui au réel x fait correspondre $y = f(x)$

et on donne une expression permettant de calculer la fonction f. Une notation employée est aussi la suivante :

- $x \rightsquigarrow x^3 + 1 - \sqrt{x+2}$
- $(x, y) \rightsquigarrow \dfrac{x-y}{x+y}$
- $x \rightsquigarrow 33$ (fonction constante de valeur 33)

1.1.1 Notation lambda

En Scheme ces notations peuvent se traduire respectivement par

```
(lambda(x) (+ (* x x x) 1 (- (sqrt (+ x 2)))))
(lambda (x y) (/ (- x y) (+ x y)))
(lambda (x) 33)
```

On utilise donc la notation *lambda* celle-ci permet de mieux faire apparaître que la fonction f dépend d'une, deux ou plusieurs variables. Dans cette notation, le vocable `lambda` est synonyme du signe mathématique : $\rightsquigarrow$.
D'une façon générale, on définit une *lambda-expression* comme une expression de la forme :

$$\boxed{\texttt{(lambda (}\langle var_1\rangle \ldots \langle var_n\rangle\texttt{)}\ \langle exp\rangle\texttt{)}}$$

où $\langle var_1\rangle$ ... $\langle var_n\rangle$ sont des variables, et $\langle exp\rangle$ une expression.
La suite ($\langle var_1\rangle$ ... $\langle var_n\rangle$) est appelée la *liste de paramètres formels* de la lambda-expression, et $\langle exp\rangle$ le *corps* de cette lambda-expression, celui-ci décrit le moyen de calculer la valeur de la fonction pour des valeurs données des variables.

1.1.2 Remarques

Noter que dans cette notation `(lambda (x y) (/ (- x y) (+ x y)))` , forme un tout. Si on veut reparler de cette fonction, il faut écrire en totalité la description de la fonction. Par contre on peut donner un nom aux fonctions ci-dessus pour pouvoir les réutiliser facilement on doit alors écrire :

```
(define f (lambda(x) (+ (* x x x) 1 (- (sqrt (+ x 2)))) ) )
(define g (lambda (x y) (/ (- x y) (+ x y))) )
(define h (lambda (x) 33))
```

Cette écriture est strictement équivalente à celle qui a été vue dans le premier chapitre :

```
(define (f x) (+ (* x x x) 1 (- (sqrt (+ x 2))))  )
(define (g x y) (/ (- x y) (+ x y)) )
(define (h x) 33)
```

Elle fait simplement référence à la forme `lambda`.

1.1.3 Exemple de session

On peut illustrer les constructions données par quelques exemples de définition et d'utilisation de fonctions.

```
>  ((lambda (x y) (/ (- x y) (+ x y))) 3 -2)
5
>  ((lambda (x y) (/ (- x y) (+ x y))) 6 4)
0.20
>  (define  f (lambda (x y) (/ (- x y) (+ x y))))
F
>  (f 3 -2)
5
>  (f 6 4)
0.20
>  (define g (lambda(t) (f t 1))
G
>  (g 9)
0.8
```

1.1.4 Remarque

Le symbole `lambda` vient de la “ notation λ ” due au logicien Church, qui a défini un moyen de présenter de façon abstraite les fonctions, celles données plus haut s'écrivent dans la notation de Church :

- $\lambda(x) \cdot x^3 + 1 - \sqrt{x+2}$
- $\lambda(x, y) \cdot \dfrac{x-y}{x+y}$
- $\lambda(x) \cdot 33$ (fonction constante de valeur 33)

1.2 Opérateurs sur les fonctions

Nous examinons maintenant comment définir une fonctionnelle dont la donnée est une fonction et dont le résultat est un réel. Il s'agit donc d'une opération qui à une fonction associe un nombre. Plusieurs exemples de cette situation sont courants en analyse : la dérivée en un point d'une fonction dérivable, l'intégrale dans un intervalle fermé d'une fonction continue, le maximum ou le minimum d'une fonction pour certaines valeurs de la variable. L'opération qui à une fonction associe sa valeur en un point est aussi une fonctionnelle, celle-ci s'écrit très simplement et ne présente pas beaucoup d'intérêt, sinon que c'est une des fonctionnelles les plus simples à définir :

```
> (define (valeur-en-zero f) (f 0))
valeur-en-zero
>(define (valeur-en x0 f) (f x0))
 valeur-en
> (valeur-en-zero acos)
1.570796326794897
> (valeur-en 1.570796326794897 cos)
-3.828588921589438e-16
```

1.2.1 Dérivée ponctuelle

Il s'agit ici d'associer à une fonction f une valeur approchée de sa dérivée en un point x_0. Nous l'exprimons sous la forme d'une limite. Une approximation possible consiste à calculer la variation dans un intervalle de longueur 2ϵ autour de x_0 on utilise la formule mathématique :

$$f'(x_0) = \lim_{\epsilon \to 0} \frac{f(x_0+\epsilon)+f(x_0-\epsilon)}{2\epsilon}$$

Ce qui se traduit en Scheme par le calcul suivant où il faut choisir une valeur de ϵ proche de 0 :

```
(define (derivee-ponctuelle f x0 epsilon)
  (/ (- (f (+ x0 epsilon))
        (f (- x0 epsilon)))
     (* 2 epsilon)))
```

On peut alors utiliser la fonctionnelle `derivee-ponctuelle` sur toute fonction réelle de la variables réelle définie dans un intervalle $[x_0-\epsilon, x_0+\epsilon]$. Par exemple on obtient :

```
> (derivee-ponctuelle (lambda (x) (* x x x)) 2.0 0.01)
 12.00009999999985
```

Remarquons que la valeur *exacte* de la dérivée de la fonction $x \rightsquigarrow x^3$ en $x_0 = 2$ est $3 \times x_0^2 = 12$; si l'on fait diminuer la valeur de `epsilon` on peut s'attendre à une amélioration de la précision des résultats, c'est en effet le cas :

```
> (derivee-ponctuelle (lambda (x) (* x x x)) 2.0 0.0001)
12.00000001001289
```

Malheureusement il y a une limite à cet accroissement de précision et le résultat peut même devenir absurde si on prend un nombre ϵ trop petit :

```
>  (derivee-ponctuelle (lambda (x) (* x x x)) 2.0  1e-10)
```

```
12.00000099288445
>  (derivee-ponctuelle (lambda (x) (* x x x)) 2.0  1e-12)
12.00106680698809
> (derivee-ponctuelle (lambda (x) (* x x x)) 2.0  1e-20)
0.0
```

Remarque sur la précision des calculs.
Cette perte de précision est due aux problèmes d'approximation auxquels on est confronté lors du calcul de la dérivée de f en un point x_0. En analysant la formule

$$\frac{(f(x_0+\epsilon)-f(x_0-\epsilon))}{2\epsilon}$$

pour de “petites” valeurs du réel ϵ, on vérifie que le numérateur et le dénominateur prennent des valeurs à l'extérieur du domaine où l'on peut faire des calculs précis.[1]
Nous pouvons sans problème utiliser aussi `derivee-ponctuelle` pour des fonctions déjà définies (ou primitives), comme le montrent les exemples suivants :

```
> (derivee-ponctuelle sqrt 2.0 1e-4)
 0.3535533907039756
> (/ 1 (* 2.0 (sqrt 2.0)))
 0.3535533905932737
> (derivee-ponctuelle  log 1 0.001)
1.000000333333479
> (derivee-ponctuelle sin 0 0.001)
0.9999998333333416
```

1.2.2 Intégrale d'une fonction continue

On peut calculer l'intégrale approchée d'une fonction f continue dans l'intervalle $[a, b]$:

$$\int_a^b f(t)dt$$

à l'aide de la méthode dite des trapèzes, il s'agit de partager l'intervalle en n parties égales et d'évaluer l'aire du trapèze formé par l'axe des x, deux droites d'équations $x = x_{i-1}$, $x = x_i$ et le segment qui joint les points d'intersection de ces deux droites avec la courbe représentative de $y = f(x)$.

On a donc la suite de points

$$x_i = a + (b-a)i/n$$

[1] Pour approfondir ces questions, il est bon de se reporter à un ouvrage spécialisé, celles-ci sortent du cadre d'un cours d'initiation à la programmation.

courbe.eps

FIG. 1.1 - : Intégrale par la formule des trapèzes

et une valeur approchée de l'intégrale est donnée par :

$$\sum_{i=1}^{n} \frac{(f(x_i) + f(x_{i-1}))(b-a)}{2n}$$

puisque l'aire d'un trapèze est donnée par le produit de la : demi-somme des bases : $(f(x_i) + f(x_{i-1}))/2$ et de la hauteur : $(b-a)/n$.

La fonction Scheme qui effectue ce calcul consiste d'abord en des définitions auxiliaires permettant le calcul direct des x_i et du terme général intervenant dans la formule ci-dessus; un rappel de la fonction récursive qui calcule la somme des éléments d'une suite est enfin utile. Ceci donne la fonction suivante :

```
(define (integrale-approchee f a b n)
    (define (x i) (+ a (* i (/ (- b a) n))))

    (define (u i) (* (+ (f (x i)) (f (x (- i 1)) )) (/ (- b a) ( * 2 n))))

    (define (somme y n) (if (= n 1) (y 1)
                            (+ (y n) (somme y (- n 1) ))
                        )
    )

    (somme u n)
)
```

Dans cette définition on voit apparaître des fonctions auxiliaires dont la portée est locale à la partie qui est à l'intérieur de la définition et qui ne peuvent pas être utilisées à l'extérieur, cette notion sera étudiée en détail au chapitre 7. La dernière ligne `(somme u n)` indique que pour le calcul approché de l'intégrale il faut faire appel à la fonction de calcul d'une somme. Celle-ci est : elle-même une fonctionnelle puisque son argument est une fonction, ici `u`. Le calcul de `u` utilise la définition des `xi`.

1.2.3 Autres exemples

On peut donner beaucoup d'exemples de fonctionnelles dont la donnée est une fonction et dont le résultat est un nombre. Plusieurs figurent déjà dans le chapitre 3 sur la récursivité. C'est le cas du calcul du maximum des valeurs prises par une fonction d'une variable entière sur un intervalle $[1, n]$, du calcul approché de la solution d'une équation $f(x) = 0$ dans un intervalle $[a, b]$ où $f(a)f(b) < 0$. D'autres exemples peuvent être pris parmi les fonctions d'une variable entière, citons en quelques uns : recherche dans l'intervalle $[1, n]$ de la valeur de $f(n)$ la plus proche d'un réel a donné, recherche pour deux fonctions f et g de l'élément d'un intervalle où elles sont les plus proches, calcul de la somme des valeurs, de la moyenne, de l'élément médian (celui qui possède à une unité près autant de valeurs qui lui sont supérieures que de valeurs qui lui sont inférieures).

1.3 Fonctionnelles dont le résultat est une fonction

La façon de définir de telles fonctionnelles est très voisine de ce qui a été fait jusqu'à présent, la seule différence importante réside dans le fait que dans le corps de la définition doit le plus souvent figurer la forme `lambda`.

1.3.1 Fonction associée à des nombres

Un exemple très simple de fonctionnelle consiste à associer à deux nombres a, b la fonction linéaire $f : x \rightsquigarrow a.x + b$, elle est définie en Scheme par :

```
(define (fonction-lineaire a b)
     (lambda (x) (+ (* a x) b))
)
```

On peut à partir de la donnée de cette fonctionnelle définir plusieurs fonctions linéaires, effectuer des calculs sur celles-ci et les utiliser, comme le montre l'exemple de session suivante :

```
> (define f1 (fonction-lineaire 1 5))
F1
> (f1 3)
8
> (define f2 (fonction-lineaire 2 -2))
F2
> (f2 3)
4
> ((fonction-lineaire 2 -2) 3)
4

> (integrale-approchee f2 1 3 100)
4
> (integrale-approchee (fonction-lineaire 2 -2) 1 3 200)
4
```

1.3.2 Fonction associée à une liste

On peut définir une fonction à partir d'une liste de nombres, par exemple un polynôme peut être représenté par la liste formée de ses coefficients, cette représentation sera étudiée en détail dans un chapitre de la troisième partie de ce cours. Ici nous nous contenterons de quelques exemples très simples de listes de taille limitée représentant des fonctions.

Soit par exemple la fonction homographique qui est définie à partir des paramètres a, b, c, d et qui à x fait correspondre

$$f(x) = \frac{ax+b}{cx+d}$$

On peut considérer que ces paramètres sont donnés dans une liste `u` de longueur 4, alors pour retrouver a, b, c et d respectivement, on calcule `(car u)`, `(cadr u)`, `(caddr u)` et `(car (cdddr u))`. Ceci permet de donner la fonctionnelle qui à la liste `u = (a b c d)` associe la fonction :

$$x \rightsquigarrow \frac{ax+b}{cx+d}$$

```
(define (homographique u)
    (lambda(x) (/ (+ (* (car u) x) (cadr u))
                  (+ (* (caddr u) x) (car (cdddr u))))
    )
)
```

La définition d'une fonction homographique particulière est alors facilitée :

```
> (define f (homographique '(1 1 1 0)))
F
> (f 1)
2
> (f 2)
3/2
> ((homographique '(1 1 1 0)) 2)
3/2
```

Toute famille de fonctions qui dépendent d'un nombre fixé de coefficients (polynômes de degré donné, fractions rationnelles, par exemple) peut être définie par une fonctionnelle de ce type.

1.3.3 Fonction associée à une autre fonction

La notion de dérivée que nous avons vue comme une notion ponctuelle permet classiquement de définir la *fonction dérivée* d'une fonction f donnée, c'est la fonction qui à x fait correspondre la dérivée ponctuelle de f au point x, on écrit en Scheme :

```
(define (derivee f epsilon)
```

```
  (lambda (x)
    (/ (- (f (+ x epsilon))
          (f (- x epsilon)))
       (* 2 epsilon))))
```

Ou bien si on a déjà effectué la définition de la dérivée ponctuelle, on peut être plus concis :

```
(define (derivee f epsilon)
  (lambda (x) (derivee-ponctuelle f x epsilon)))
```

Un exemple de session où l'on utilise des fonctions dérivées :

```
> (define g1 (derivee (lambda (x) (* x x x)) 1e-10))
 G1
> (g1 1)
 3.000000248221112
> (g1 10)
 299.999999987
> (define cosinus (derivee sin 1e-10))
 COSINUS
> (define pi 3.1415926)
PI
> (cosinus pi)
 -1.000000082739841
```

On peut de la même façon que pour les dérivées introduire une fonctionnelle qui à une fonction f associe l'intégrale

$$\int_a^x f(t)dt$$

il suffit de considérer le paramètre b, intervenant dans le calcul d'intégrale donné plus haut, comme une variable x ce qui se fait en Scheme par

```
(define (integrale f a n)
    (lambda (x) (integrale-approchee (f a x n)))
```

Ainsi cette notation `lambda` peut apparaître comme une opération permettant de considérer des paramètres comme des variables. C'est ce que l'on appelle parfois *abstraire les données*.

1.3.4 Fonction associée à plusieurs fonctions

Certaines opérations associent une fonction à deux ou plus de deux fonctions. Parmi celles-ci la plus simple est la *composition* dénotée par le symbole $\circ$, à deux fonctions f et g on associe la fonction $f \circ g$ qui à x fait correspondre $g(f(x))$ (on effectue d'abord f puis g). Ceci s'écrit en Scheme :

```
(define (rond f g)
    (lambda(x) (g (f x))))
```

Cette fonctionnelle permet de construire de nombreuses fonctions en les décrivant simplement à l'aide d'une famille de fonctions de base. Donnons l'exemple d'une session :

```
> (define f (rond exp log))
F
> (f 1)
1.
> (f 5)
5.
> (define f1 (lambda (x) (* x x)))
F1
> (define f2 (lambda (x) (* 2 x)))
F2
> (define (h x) (- ((rond f1 log) x) ((rond log f2) x) ))
H
> (h 2)
0.
> (h 5)
0.
> (h 100)
0.
```

Les résultats obtenus dans cette session s'expliquent aisément. Dans la première partie on a calculé la fonction $f(x) = exp \circ log(x)$ qui est l'identité sur les réels. Dans la deuxième partie on a calculé la fonction $h(x) = ln(x^2) - 2ln(x)$ qui est bien entendu la fonction identiquement nulle aux erreurs d'approximation près.

On peut définir plus généralement une algèbre de fonctions en définissant la somme, le produit de fonctions de la façon suivante :

```
> (define (plus f g)
          (lambda (x) (+ (f x) (g x)))
   )
plus
> (define (mult f g)
          (lambda(x) (* (f x) (g x)))
  )
> (define u (plus (mult sin sin) (mult cos cos)))
u
> (u 0)
1.0
> (u pi)
1.0
> (u 0.5)
1.0
```

En conclusion on voit que la notion de fonctionnelle permet de traiter les fonctions comme des objets de base de la programmation en Scheme. Ceci est très agréable pour un grand nombre d'applications en calcul numérique.

Chapitre 2

Terminaison et efficacité des fonctions récursives

Nous avons vu dans le chapitre 3 de la Partie 1, que la définition de fonctions récursives posait un certain nombre de problèmes. Le premier est de savoir, lorsqu'on définit une fonction récursive, si le calcul qui la décrit se termine. C'est un problème à la fois excessivement important et particulièrement difficile à résoudre. Un des résultats fondamentaux de l'informatique théorique affirme que ce problème est *indécidable*, c'est à dire qu'il n'y a pas d'algorithme permettant de vérifier si une fonction termine son calcul. Nous allons tout d'abord montrer comment on peut prouver ce résultat, puis nous donnerons des exemples de fonctions pour lesquelles la terminaison n'est pas évidente. Nous aborderons ensuite le problème de l'efficacité; pour une fonction qui termine son calcul, on peut en effet chercher à savoir combien d'opérations sont effectuées pour en calculer la valeur. Si ce nombre d'opérations est trop important le temps d'attente du résultat sera excessif. Une étude attentive de cette question est nécessaire dans toutes les applications informatiques de taille importante, car la mise en œuvre de certaines techniques fait souvent gagner un temps appréciable.

2.1 Indécidabilité de la terminaison

Dans cette section nous démontrons qu'il n'existe pas de fonction Scheme qui permette de tester si une fonction Scheme termine. Noter que ce fait est totalement indépendant du langage (de programmation, ou d'expression d'algorithmes) que l'on utilise. Nous présentons cette preuve sous la forme d'une petite histoire informatique imagée.

Le responsable des travaux pratiques d'Informatique en a assez des programmes qui bouclent écrits par des étudiants peu expérimentés car cela l'oblige à chaque fois à des manipulations compliquées pour stopper ces programmes. Il voit alors dans un journal spécialisé une publicité:

> Ne laissez plus boucler vos programmes Scheme! Utilisez notre fonction `(termine u)`, chaque fois que vous lui donnez la liste `u` correspondant à un programme

Scheme elle répond `#t` si le programme ne boucle pas et `#f` si le programme boucle. En n'utilisant que les programmes pour lesquels `termine` répond `#t` vous évitez tous les problèmes de non terminaison. D'ailleurs voici quelques exemples:

```
> (define (f x) ( + x 1))
F
> (termine 'f)
#t
>(define (g x) (g (+ x 1)))
G
> (termine 'g)
#f
> (define (h x) (if (<= x 0) (+ x 1)
                             (* 2 (h (- x 1))))))

H
> (termine 'h)
#t
> (define (u x) (if (<= x 0) (+ x 1)
                             (* 2 (u (+ x 1)))))
U
> (termine 'u)
#f
```

Impressionné par la publicité le responsable des travaux pratiques achète à prix d'or cette petite merveille et pense que sa vie d'enseignant va être enfin tranquille. C'était sans compter sur l'élève Gödel qui, dès qu'il apprend l'acquisition faite par le Maître, écrit la fonction suivante:

```
(define (absurde)
         (if (termine 'absurde) (absurde)
             #f)
)
```

Le raisonnement suivant montre l'impossibilité d'une cohérence logique dans la fonction ci-dessus. Si la fonction `(absurde)` boucle indéfiniment alors `(termine absurde)` sera *faux* et donc par construction de la fonction `(absurde)`, elle terminera. Si au contraire on suppose que `(absurde)` termine alors `(termine absurde)` sera vrai et par construction de la fonction `(absurde)` elle bouclera indéfiniment. Ceci est parfaitement contradictoire, et montre l'impossibilité de réaliser une telle fonction.

Pour illustrer ce raisonnement, supposons que le vendeur malhonnête ait défini `termine` comme une fonction rendant toujours la valeur `#t`, alors la session suivante montre le caractère erroné de la construction:

```
>  (define (termine u) #t)
```

```
termine
> (termine 'absurde)
#t
> (absurde)
```

Pas de résultat, bouclage infini ...
Et bien que le résultat de termine soit `#t` pour `absurde` la procédure `absurde` ne termine pas. Si à l'opposé on définit la fonction `termine` comme une fonction qui donne toujours le résultat `#f` alors la session suivante montre l'incohérence de `termine`:

```
> (define (termine u) #f)
termine
> (termine 'absurde)
#f
> (absurde)
#f
```

La c'est l'inverse qui se produit, la fonction `absurde` termine et pourtant le résultat de `(termine absurde)` est négatif. Quelque soit la procédure `termine`, même bien plus compliquée et pouvant donner certaines fois un résultat correct, on trouvera toujours une fonction `f` qui donnera l'un des deux résultats incohérent précédents: soit `f` termine bien que `(termine f)` donne pour résultat *faux*, soit `f` ne termine pas bien que `(termine f)` donne pour résultat *vrai* .

2.2 Exemples de fonctions et leur terminaison

Malgré le résultat négatif précédent, on peut pour certaines fonction récursives démontrer qu'elles terminent. Ce fait n'est pas toujours simple à prouver, on utilise souvent un raisonnement de la forme suivante: une fonction récursive où le calcul de $f(n)$ demande celui de $f(n-1)$ pour $n \neq 0$, et où pour $n = 0$ il n'y a pas d'appel récursif, termine toujours pour les valeurs *positives* de la variable n. [1].

2.2.1 Exemples simples

Il est facile de voir que la fonction suivante donne toujours un résultat pour n entier naturel $(n \geq 0)$

```
(define (f n)
      (if ( = n 0) 2
           (* (f (- n 1) ) (f (- n 1)))
       )
 )
```

[1] Noter que par contre il y a une boucle infinie pour les valeurs négatives de la variable

Un autre exemple est une fonction qui ne termine pas son calcul du fait que le calcul de $f(n)$ nécessite la connaissance de $f(n+1)$, qui demande celle de $f(n+2)$ et ainsi de suite jusqu'à l'infini.

```
(define (f n)
      (if ( = n 0) 2
           (* (f (- n 1) ) (f (+ n 1)))
       )
 )
```

Cette fonction boucle indéfiniment pour toute valeur de n, $n \neq 0$. Mais le fait qu'il y ait un appel à $f(n+1)$ ne signifie pas systématiquement que la fonction ne termine pas, ainsi la fonction g définie ci-dessous termine en raison du fait que pour tout nombre entier n l'un des trois nombres $n, n+1, n+2$ est divisible par trois.

```
 (define (g n)
    (if (= 0 (modulo n 3))
        n
        (* 2 (g (+ n 1)))
    )
)
```

2.2.2 Autres exemples

Dans certains cas la raison de la terminaison est plus difficile à trouver comme par exemple pour la fonction suivante où `premier` est le prédicat vu au chapitre 3: (le booléen `(premier n)` est vrai si et seulement si l'entier n est un nombre premier).

```
 (define (h n)
    (if (premier n)
        n
        (h (+ n 1))
    )
)
```

Dans ce cas la raison de la terminaison est le fait bien connu, mais pas complètement évident, suivant lequel la suite des nombres premiers est infinie. La convergence d'une suite peut impliquer le fait qu'une fonction termine son calcul. Ainsi tout résultat mathématique affirmant que la suite u_n tend vers une limite admet pour conséquence la terminaison de la fonction récursive suivante, lorsque le nombre ϵ est bien choisi:[2]

```
 (define (valeur-limite u epsilon)
    (define (aux-limite u epsilon n)
        (if (< (abs (- (u n) (u (+ n 1))) epsilon)
```

[2]Le traitement informatique de nombres décimaux peut parfois poser des problèmes du fait que l'on ne manipule que des valeurs approchées

```
             n
            (aux-limite u epsilon (+ n 1)) )
    )
    (aux-limite u epsilon 1)
)
```

Cette fonction termine si et seulement si pour tout réel positif ϵ il existe un entier n tel que $|f(n) - f(n-1)| < \epsilon$, ceci est impliqué par la convergence, (mais ne lui est pas équivalent!)

Suite de Syracuse.
Donnons pour terminer cette liste d'exemples une fonction, appelée parfois suite de Syracuse, dont on ne sait pas si elle termine son calcul pour tout entier n

```
 (define (syracuse n)
    (cond ((= n 1) 0)
          ((= 0 (modulo n 2) ) (+ 1 (syracuse (/ n 2))))
          (else (+ 1 (syracuse (+ 1 (* 3 n)))))
    )
  )
```

Pour $n = 7$ par exemple, les valeurs suivantes de la variable sont successivement paramètre d'appel:

$$7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$$

et la valeur de $syracuse(7)$ est 16, cette fonction, qui a été introduite pour la première fois dans l'université de Syracuse aux Etats Unis, est particulièrement mystérieuse. Malgré les très nombreux travaux accomplis à son sujet, on ne sait toujours pas pas si $syracuse(n)$ est définie pour tout entier n. C'est à dire si on finit par trouver 1 dans la suite des valeurs d'appel de la fonction.

2.3 Techniques de preuve de terminaison

Dans certains cas les méthodes ci-dessous permettent de démontrer que des fonctions terminent.

2.3.1 Ordres bien fondés

Une façon de prouver qu'une fonction termine consiste à construire *un ordre bien fondé* sur les valeurs des variables de l'appel. Donnons la définition de cette notion:
soit E un ensemble muni d'une relation d'ordre partiel c'est à dire une relation $\preceq$ telle que:

- $\forall x \in E, x \preceq x$ (réflexivité)

- $\forall x, y, z \in E, x \preceq y$ **et** $y \preceq z \Rightarrow x \preceq z$ (transitivité)
- $\forall x, y \in E, x \preceq y$ **et** $y \preceq x \Rightarrow x = y$ (antisymétrie)

On dit que $(E, \preceq)$ est *bien-fondé* s'il n'existe pas de suite infinie strictement décroissante $\ldots x_{n+1} \preceq x_n \preceq \ldots \preceq x_2 \preceq x_1$ d'éléments de E telle que pour tout n $x_n \neq x_{n+1}$. Par exemple, $(\mathbf{N}, \leq)$ est bien fondé, alors que $(\mathbf{Z}, \leq)$ ne l'est pas.
L' utilisation d'un ordre bien fondé pour prouver la terminaison d'une fonction se trouve dans la proposition suivante. Soit une fonction récursive dont on veut calculer les valeurs pour un ensemble E muni d'un ordre bien fondé $\preceq$, et dont la définition est donnée en Scheme par une suite d'opérations de la forme suivante:

```
(define (f x) (if (P x) a (f (u x))))
```

Proposition. *Si* `P` *est un prédicat et* `u` *une fonction de E dans E satisfaisant:*

- *Pour tout x_0 élément minimal de E (c'est à dire tel que $x \preceq x_0 \Rightarrow x = x_0$) $P(x_0)$ est vrai.*
- *Pour tout x de E on a: $u(x) \preceq x$*

Alors la fonction $f(x)$ donnée ci-dessus termine son calcul pour tout x dans E.

2.3.2 Exemple d'application

Le calcul des coefficients binomiaux donne une illustration de la technique précédente, lorsqu'ils sont calculés par la formule suivante:

$$\forall n \geq 0 \quad \binom{n}{0} = \binom{n}{n} = 1$$
$$\forall n > 1, \forall p, \quad 0 < p < n, \quad \binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$$

qui se traduit par la fonction Scheme:

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

Considérons l'ensemble E des couples (n, p) d'entiers et la relation $\preceq$ donnée par $(a, b) \preceq (c, d)$ si et seulement si $a \leq c$, $b \leq d$ et $c \neq d$, cette relation est un ordre partiel bien fondé et les paramètres d'appel de la fonction `binome` satisfont la condition de décroissance de la proposition; comme de plus la valeur de la fonction est définie pour les éléments minimaux de $E, \preceq$, le calcul termine donc pour toutes les couples (n, p) d'entiers naturels.

2.4 Amélioration de l'efficacité sur quelques exemples

Certaines fonctions récursives donnent lieu à des temps de calcul prohibitifs. Tel est le cas des fonctions contenant deux appels récursifs successifs dans le corps de leur définition.

2.4.1 Calcul doublement récursif

Nous avons déjà vu que les nombres de Fibonacci sont calculés par la fonction:

```
(define (fibo n)
   (if (< n 2) 1
       (+ (fibo (- n 1))
          (fibo (- n 2)))))
```

Considérons l'évaluation de `(fibo 20)`, elle peut se décomposer ainsi:

évaluation de `(fibo 20)` dans ρ_g:

1. –
   évaluation de `(fibo n)` avec $n = 19$
   (a) – `(fibo n)` $\stackrel{n=18}{\Longrightarrow}$ 4181
       – `(fibo n)` $\stackrel{n=17}{\Longrightarrow}$ 2584
   (b) $4181 + 2584 = 6765$

   – `(fibo n)` $\stackrel{n=18}{\Longrightarrow}$ 4181
2. $6765 + 4181 = 10946$

On peut remarquer que l'évaluation de `(fibo n)` pour $n = 18$ est effectuée deux fois, un raisonnement simple montre que trois évaluations sont demandées pour $n = 17$, cinq pour $n = 16$, huit pour $n = 15$ etc ... On peut représenter par un arbre l'enchaînement des appels récursifs dans ce calcul (voir Fig. 2.1).
On peut montrer que le nombre d'additions a_n effectuées par cette fonction pour calculer le nombre de Fibonnacci $fib(n)$ satisfait la relation de récurrence:

$$a_0 = 0 \quad a_1 = 0 \qquad a_n = 1 + a_{n-1} + a_{n-2}$$

le nombre a_n est de l'ordre de $(\frac{1+\sqrt{5}}{2})^n$ qui croit très vite lorsque n est grand; ceci explique que le temps de calcul du nombre de Fibonnacci par cette méthode soit si important pour des valeurs de n supérieures à 30.

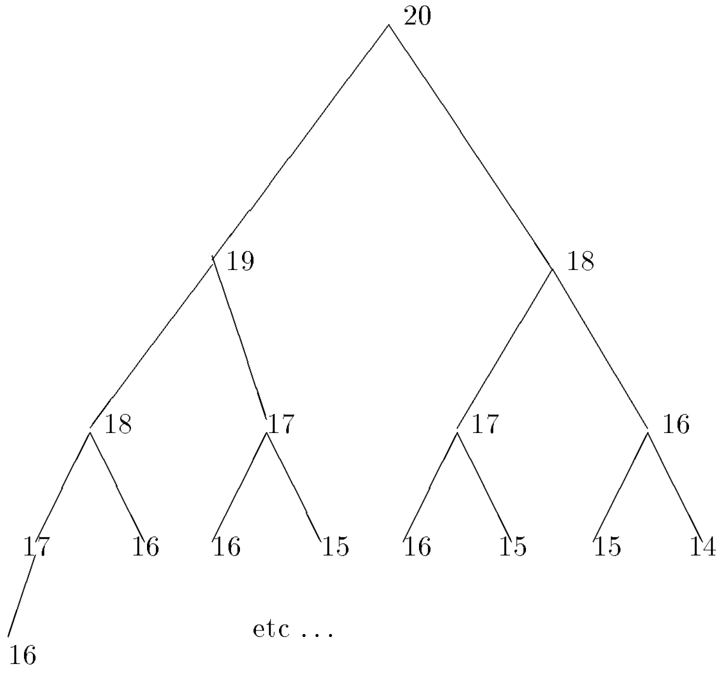


FIG. 2.1 - : Arbre des appels pour `(fibo 20)`

2.4.2 Suite de Fibonacci généralisée

Une amélioration de la définition de `fibo` nécessite une étude du schéma de calcul associé aux nombres de Fibonacci.
La définition de la suite de Fibonacci est composée de deux parties indépendantes:

- Les valeurs initiales : $f_0 = 1$ et $f_1 = 1$
- Le schéma de calcul: $f_n = f_{n-1} + f_{n-2}$ $(n > 1)$

On peut remarquer qu'en donnant pour valeurs initiales 1 et 2 à f_0 et f_1 on effectue un appel récursif en moins dans tous les appels de `fibo` ce qui se traduit par un gain de temps d'autant plus important que ce nombre d'appels est grand, le remplacement de 1 et 2 par 2 et 3, puis par 3 et 5 fera gagner encore plus de temps. Il vient ainsi l'idée de considérer les valeurs initiales de la suite comme des variables et d'introduire la suite suivante:

> La suite de Fibonacci généralisée $f^{a,b}$ est définie pour tout couple d' entiers naturels a et b par:

$$\begin{cases} f_0^{a,b} = a \\ f_1^{a,b} = b \\ f_n^{a,b} = f_{n-1}^{a,b} + f_{n-2}^{a,b} \quad \textbf{si } n > 2 \end{cases}$$

On vérifie alors la relation (preuve par récurrence sur n):

$$f_n^{a,b} = f_{n-1}^{b,a+b} \quad (n > 0)$$

la suite de Fibonacci est elle donnée par:

$$f_n = f_n^{1,1}$$

Le programme ci-dessous est une traduction de ces égalités, il contient une définition récursive locale de la fonction calculant le n-ième terme de la suite de Fibonacci généralisée.

```
> (define (fibonacci-rapide n)
      (define (generalise-fibo n a b)
        (if (= n 1) b
            (generalise-fibo (- n 1) b (+ a b))
        )
    )
     (if (= n 0) 1 (generalise-fibo n 1 1)))

> (rapide-fibonacci 20)
 10946
```

Ce dernier résultat est obtenu plus rapidement que la précédente évaluation de `(fibo 20)`. Une *trace* du calcul décrit la suite des appels des fonctions `fibonacci-rapide` et `generalise-fibo`. Cette trace est présentée dans la figure 2.2.

Remarque.

L'évaluation de `(rapide-fibonacci 20)` demande 21 appels de `generalise-fibo` à la place de 10946 appels de `fibo` dans le premier programme. On peut cependant noter que la valeur 17711 a été calculée "en trop". On peut modifier légèrement la définition de `rapide-fibonacci` pour éviter cette addition inutile.

2.4.3 Calcul de x^n

Nous appelons "naïve" une définition de fonction, traduction directe du problème à résoudre, sans souci d'efficacité. Il ne faut pas avoir peur de commencer par donner une solution naïve à un problème donné. En principe, une solution naïve est correcte. C'est souvent en étudiant les raisons du manque d'efficacité d'une programmation que l'on peut trouver de substantielles améliorations. La définition ci-dessous, version naïve de la fonction, est la traduction directe des égalités:

$$x^0 = 1 \quad x^n = x \times x^{n-1}$$

```
(define (puissance x n)
    (if (= n 0) 1
        (* x (puissance x (- n 1)))))
```

- appel `(rapide-fibonacci 20)`
- appel `(generalise-fibo 20 1 1)`
- appel `(generalise-fibo 19 1 2)`
- appel `(generalise-fibo 18 2 3)`
- appel `(generalise-fibo 17 3 5)`
- appel `(generalise-fibo 16 5 8)`
- appel `(generalise-fibo 15 8 13)`
- appel `(generalise-fibo 14 13 21)`
- appel `(generalise-fibo 13 21 34)`
- appel `(generalise-fibo 12 34 55)`
- appel `(generalise-fibo 11 55 89)`
- appel `(generalise-fibo 10 89 144)`
- appel `(generalise-fibo 9 144 233)`
- appel `(generalise-fibo 8 233 377)`
- appel `(generalise-fibo 7 377 610)`
- appel `(generalise-fibo 6 610 987)`
- appel `(generalise-fibo 5 987 1597)`
- appel `(generalise-fibo 4 1597 2584)`
- appel `(generalise-fibo 3 2584 4181)`
- appel `(generalise-fibo 2 4181 6765)`
- appel `(generalise-fibo 1 6765 10946)`
- appel `(generalise-fibo 0 10946 17711)`
- Résultat `10946`

FIG. 2.2 - : Trace de l'évaluation de (fibonacci-rapide 20)

On remarquera que pour calculer x^n, n multiplications par x sont effectuées. Considérons maintenant les égalités suivantes:

$$x^0 = 1 \qquad x^{2n+1} = x \times x^{2n} \qquad x^{2n} = (x^2)^n$$

Celles-ci permettent de donner une nouvelle version du calcul de x^n qui est plus efficace:

```
(define (puissance x n)
  (cond ((= n 0) 1)
        ((odd? n) (* x (puissance x (- n 1))))
        (else (puissance (* x x) (/ n 2)))))
```

La trace des appels à `puissance` pour le calcul de 2^{10} est donnée par:

- appel `(puissance 2 10)`
-    appel `(puissance 4 5)`
-       appel `(puissance 4 4)`
-          appel `(puissance 16 2)`
-             appel `(puissance 256 1)`
-                appel `(puissance 256 0)`
-                résultat: 1
-             résultat: $1 \times 256 = 256$
-          résultat: 256
-       résultat: 256
-    résultat: $4 \times 256 = 1024$
- résultat: 1024

en effet:

$$2^{10} = 4^5 = 4 \times 4^4 = 4 \times 16^2 = 4 \times 256^1 = 4 \times 256 \times 1 = 1024$$

On peut montrer que le nombre d'appels de `puissance` pour le cacul de x^n par cet algorithme est donné par la représentation de n en base 2. Si l'ecriture de n en base 2 contient a fois le chiffre 1 et b fois le chiffre 0 alors le nombre d'appels sera $2a + b - 2$. On voit que ce nombre est en géneral bien plus petit que n, il est de l'ordre de $\log(n)$ pour n grand.

2.5 Coefficients binomiaux

Nous utilisons comme dernière illustration des problèmes d'efficacité le calcul du coefficient binomial $\binom{n}{p}$.

2.5.1 Première forme de calcul

Nous utilisons tout d'abord directement la formule

$$\binom{n}{p} = \frac{n!}{p!(n-p)!}$$

Dans ce calcul la définition de `(binomial n p)` contient une définition *locale* de la fonction factorielle:

```
(define (binomial n p)
    (define (fact n)
        (if (= n 0) 1 (* n (fact (- n 1))))
    )
    (/ (fact n)
       (* (fact p)
          (fact (- n p))))
)
```

Il est clair que cette évaluation est trop coûteuse en multiplications. Sur un exemple simple, le calcul de `(binomial 17 5)`, on comptabilise en effet:

- 17 multiplications pour calculer $17! = 355687428096000$
- 5 multiplications pour calculer $5! = 120$
- 12 multiplications pour calculer $12! = 479001600$
- 1 multiplication de 120 par $479001600 = 57480192000$
- 1 division de 355687428096000 par $57480192000 = 6188$

Soit 36 multiplications et divisions dont certaines portent sur des grands nombres.

2.5.2 Deuxième version: les arrangements

Une meilleure version peut utiliser l'égalité donnant le nombre d'arrangements de p objets pris parmi n:

$$A_n^p = \frac{n!}{p!} = (p+1)(p+2)\dots n$$

On a alors:

$$\binom{n}{p} = \frac{A_n^p}{A_p^1}$$

et le calcul de A_n^p demande simplement $n-p$ multiplications comme le montre la réalisation suivante:

```
(define (binomial n p)
    (define (arrangement n p)
        (if (= p 0) 1 (* n (arrangement (- n 1) (- p 1))))
    )
     (/ (arrangement n p)
         (arrangement p p))
)
```

Si l'on reprend le calcul de $\binom{17}{5}$, nous montrons ici les valeurs successives de `(arrangement n p)` pour $p = 12$ et n varie de 12 à 17:

- $A_n^p = 742560 \quad = 43680 * 17 \quad \mathtt{n} = 17 \quad \mathtt{p} = 5$
- $A_n^p = 43680 \quad = 2730 * 16 \quad \mathtt{n} = 16 \quad \mathtt{p} = 4$
- $A_n^p = 2730 \quad = 182 * 15 \quad \mathtt{n} = 15 \quad \mathtt{p} = 3$
- $A_n^p = 182 \quad = 13 * 14 \quad \mathtt{n} = 14 \quad \mathtt{p} = 2$
- $A_n^p = 13 \quad = 1 * 13 \quad \mathtt{n} = 13 \quad \mathtt{p} = 1$
- $A_n^p = 1 \quad \mathtt{n} = 12 \quad \mathtt{p} = 0$

Nous pouvons évaluer le coût en opérations arithmétiques obtenu:

- 5 multiplications pour calculer $A(17,5) = 742560$
- 5 multiplications pour calculer $A(5,1) = 5! = 120$
- 1 division de 742560 par 120 = 6188

Soit 11 multiplications/divisions portant sur des entiers de taille raisonnable.

2.5.3 Récurrence double

Pour calculer $\binom{n}{p}$, nous pouvons aussi utiliser les célèbres formules:

$$\binom{n}{n} = 1 \qquad \binom{n}{0} = 1$$

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p} \quad (0 < p < n)$$

Ceci donne le programme :

```
(define (binome n p)
    (cond ((or (= n p) (= p 0)) 1)
        (else (+ (binome (- n 1) p)
                 (binome (- n 1) (- p 1))))))
```

L'évaluation de `(binome 17 5)` donne bien sûr 6188 , mais le temps de calcul parait bien long.
La raison de l'inefficacité de cette programmation est profonde; comme pour la programmation naïve de la fonction de Fibonacci, l'application de nombreuses fois de la fonction `binome` aux mêmes arguments, provoque un grand nombre de calculs inutiles.
L'inefficacité de notre schéma de calcul peut être estimée, soit par un traitement d'analyse formelle, soit expérimentalement.

Une méthode de mesure intéressante est de définir une fonction `(appels-de-binome n p i j)` mesurant combien de fois l'appel `(binome n p )` provoque l'évaluation de `(binome i j)` $(0 \leq i \leq n \quad 0 \leq j \leq p)$.

```
> (appels-de-binome 17 5 6 4)
11
> (appels-de-binome 17 5 10 3)
21
> (appels-de-binome 17 5 3 2)
 364
```

2.5.4 Dernière version

Une autre façon de calculer $\binom{n}{p}$ consiste à utiliser la formule de récurrence

$$\binom{n}{p} = \binom{n}{p-1} \frac{n-p+1}{p}$$

$$\binom{n}{0} = 1$$

formule qui conduit à effectuer p multiplications et p divisions sur des entiers qui n'excèdent pas n fois le coefficient cherché (cette majoration n'est pas satisfaite par le calcul utilisant les arrangements), si deplus on remplace p par $n-p$ dans le cas ou $2p > n$ on obtient un des meilleurs algorithmes possibles pour le calcul du coefficient binomial:

```
(define (binomial n p)
    (cond ((> (* 2 p) n ) (binomial n (- n p)))
          ((= 0 p) 1)
```

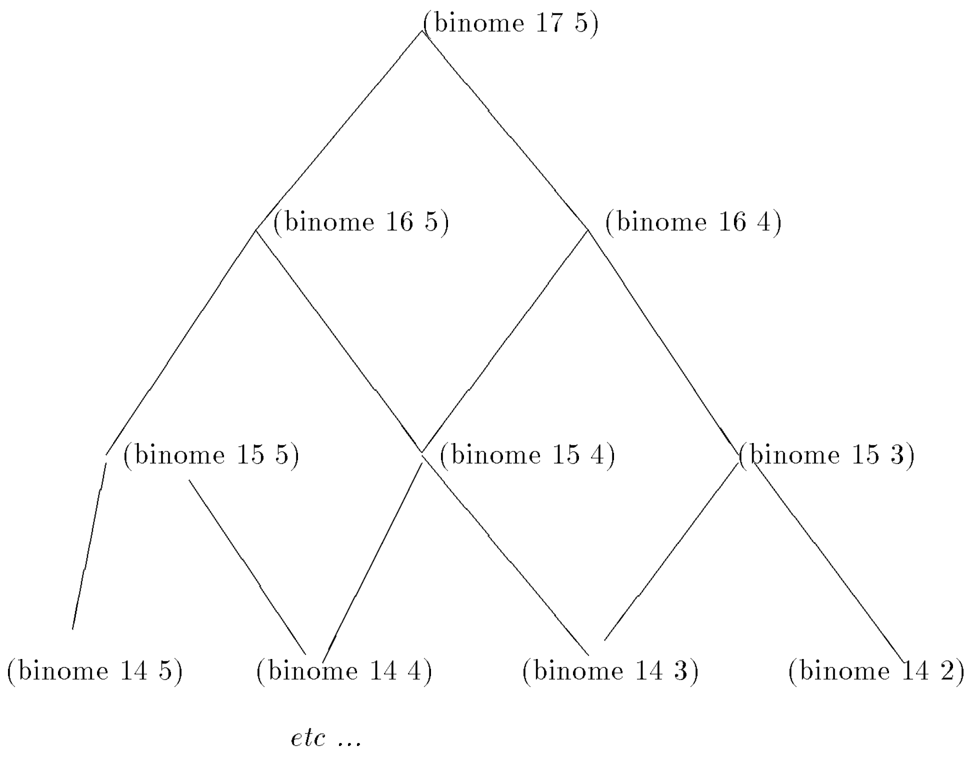


FIG. 2.3 - : Quelques appels provoqués par l'évaluation de `(binome 17 5)`

```
          (else (/ (* (binomial n (- p 1)) (+ n (- p) 1) ) p))
    )
)
```

Chapitre 3

Portée des définitions

Dans ce chapitre on étudie une question classique dans tous les langages de programmation, il s'agit du domaine dans lequel un identificateur représente une valeur donnée. A l'intérieur d'une session de programmation un identificateur peut changer de signification; par exemple considérons la suite de définitions suivante:

```
> (define x '( 4 7 9))
X
> (cdr x)
(7 9)
> (define (f x) (* x x))
F
> (f 3)
9
> x
(4 7 9)
> (define x 4)
X
> (f x)
16
```

On constate que l'identificateur `x` représente trois objets différents au cours de la session. D'abord il s'agit d'une liste que l'on initialise et à laquelle on applique la fonction `cdr`. Ensuite pendant un très court passage `x` représente la variable permettant d'exprimer la façon de calculer la fonction `f`. Le fait d'avoir utilisé `x` dans ce cadre n'a pas altéré sa valeur qui était une liste. Par contre, le fait d'avoir utilisé un `define` pour attribuer à `x` la valeur 4 a détruit l'ancienne valeur de `x`. Comme il a été dit dans le premier chapitre la forme `define` détruit l'ancienne valeur de l'identificateur qu'elle redéfinit. Par contre l'utilisation d'un paramètre `x` dans la définition d'une fonction n'altère pas une variable dont le nom est `x`; on dit que la variable `x` est *locale* à la définition de la fonction. Il y a plusieurs façons d'utiliser des paramètres locaux en dehors des formes équivalentes à `lambda(x)`. Nous verrons ici deux exemples: la forme `let` et l'utilisation d'une définition locale à l'intérieur d'un `define`. Nous terminerons par une construction qui utilise ces définitions locales, il s'agit de la récursion terminale, méthode efficace d'écriture de procédures récursives.

3.1 La forme let

Dans une première approche on peut considérer cette forme comme un moyen de faire un calcul en laissant intact l'environnement d'une session. Voilà un exemple:
Supposons que l'on veuille calculer

$$\frac{1+\sqrt{2}}{\sqrt{2}-1}$$

en utilisant une variable à laquelle on va affecter la valeur $\sqrt{2}$ et ceci à l'intérieur d'une session où l'on a oublié le nom des variables que l'on a déjà utilisées et où l'on ne veut pas détruire celles-ci. On ne peut pas utiliser une forme (define x (sqrt 2)) car on ne sait pas si x n'est pas déjà utilisée. On peut par contre tirer profit de la forme let en écrivant:

```
(let ((x (sqrt 2))) (/ (+ 1 x) (- x 1)))
```

Ce qui donne le résultat souhaité. D'une manière générale, la forme let s'écrit:

```
(let ((x <a>) (y <b>) (z <c>) ...) <exp>)
```

où les variables x, y, z interviennent dans l'expression <exp>. Ceci signifie la demande à l'interpréteur de calculer l'expression <exp> en faisant prendre aux variables x,y,z les valeurs des expressions a, b,c. Un exemple de session éclaire cette notion:

```
> (let ((x 4) (y 5))
       (+ x y (* x y)))
29
```

Comme il a été dit plus haut, ceci n'altère pas des valeurs de variables qui auraient pu être définies avant:

```
> (define x 1)
X
> (define y 2)
Y
> (let ((x 4) (y 5))
       (+ x y (* x y)))
29
> x
1
> y
2
```

L'intérêt de cette forme est de donner des noms à des valeurs qui interviennent souvent dans un calcul sans pour autant utiliser des noms qui pourraient faire défaut par ailleurs.

Il est bon de faire ici un aparté sur le choix des noms; en effet en programmation, comme en mathématiques le choix des noms pour des variables est important pour la compréhension du programme (en informatique) ou de la démonstration (en mathématiques). Il est indispensable d'écrire des programmes clairs et facilement lisibles. Cela est particulièrement utile, tant pour une personne qui n'a pas écrit le programme que pour l'auteur lui même lorsque celui-ci se replonge dans sa lecture (par exemple pour le modifier) quelques temps après l'avoir écrit. Pour cela il faut se fixer des conventions précises et s'y tenir. Ainsi, il est bon de choisir pour des noms de fonctions les lettres `f,g h ...` pour noms de variables qui sont des nombres `x, y, z` ... En mathématiques une solide et ancienne tradition existe, ce qui rend les ouvrages de mathématiques plus facilement lisibles, il ne viendrait jamais à l'idée d'un mathématicien d'appeler une fonction `x` et sa variable `f` bien que rien ne l'interdise formellement. Il est indispensable de prendre le même style de conventions en programmation, c'est ce qui est constaté par expérience par tous ceux qui ont écrit des programmes de taille importante.

Donnons maintenant quelques exemples d'utilisation de la forme `let` à l'intérieur d'une session.

```
> (let ((x (sqrt 2)) (y (sqrt 5)))
       (+ x y (* x y)))

6.812559200041264

> (define (f x)
   (let ((a (sqrt 2)))
        (/ (+ a x) (- a x)) )
   )
F
> (f 0)
1.0
> (f 1)
5.828427124746188
```

La première forme permet de calculer une formule faisant intervenir $\sqrt{2}$ et $\sqrt{5}$ sans avoir à écrire plusieurs fois l'expression en question. Elle n'est ici utilisée qu'une seule fois, au moment où on l'écrit. Dans la deuxième forme le `let` intervient à l'intérieur d'un `define`, là encore il s'agit d'éviter de répéter plusieurs fois `(sqrt 2)` dans le corps de la définition de la fonction `f`. On peut aussi utiliser `let` pour effectuer des définitions locales de fonctions cela donne sur un exemple simple:

```
> (let ( (f (lambda(x)(+ x 1)) ) )
    (f 3))
4
```

3.2 Définition à l'intérieur d'une fonction

Une autre possibilité de définitions locales consiste à utiliser une forme `define` à l'intérieur d'une autre. Ceci permet d'utiliser des fonctions ou des constantes dont l'utilité est limitée à la définition d'une fonction; elles ne sont pas disponibles à l'extérieur de cette fonction. On appelle parfois ces fonctions des fonctions auxiliaires. Un exemple d'utilisation qui montre que la portée de la fonction auxiliaire se limite à l'intérieur de la définition qui l'englobe.

```
> (define (f x) (+ x 1))
F
> (define (g x)
      (define (f x ) (* x x))
      (+ (f x) (f (- x 1)))
  )
G
> (g 3)
13
> (f 7)
8
```

Une définition de fonction auxiliaire à l'intérieur d'une autre fonction doit suivre immédiatement l'entête de la fonction et ne peut pas être placée n'importe où . Sur cet exemple on voit que la définition de $f(x) = x^2$ à l'intérieur de la définition de $g(x)$ n'a pas altéré la définition de $f(x) = x + 1$ qui avait été faite auparavant.
Il est possible de définir des fonctions auxiliaires à l'extérieur des fonctions qui les utilisent mais ceci a pour inconvénient d'interférer avec d'autres fonctions qui auraient le même nom. Il est souhaitable le plus souvent de définir ces fonctions à l'intérieur des fonctions qui les utilisent. On peut noter que dans le cas qui est donné ici la forme `define` aurait très bien pu être remplacée par une forme `let` pour donner [1]:

```
> (define (g x)
      (let ((f (lambda (x) (* x x))))
```

[1]Ceci n'est possible avec `let` que si la fonction n'est pas récursive.

```
      (+ (f x) (f (- x 1)))
  )
G
> (g 3)
13
```

3.3 Récursion terminale

La notion de fonction auxiliaire est particulièrement utile pour réaliser la construction appelée *récursion terminale* permettant d'améliorer l'efficacité de certaines fonctions récursives. Prenons l'exemple du calcul de factorielle:

```
> (define (fact x)
      (if (= x 0) 1
          (* x (fact (- x 1)))
      )
)
```

Lors du calcul de factorielle de n par l'interpréteur, il y a un appel de la fonction `fact` avec le paramètre $n-1$ et mise en mémoire du fait que lorsque cette valeur sera trouvée il y aura une multiplication par n à effectuer. Ainsi les opérations à effectuer lors du retour des valeurs s'empilent, prennent de la place en mémoire et font perdre de l'efficacité. Par exemple, lors du calcul de `(fact 4)`, l'interpréteur commence par demander le calcul de `(fact 3)` et doit se rappeler qu'il doit multiplier le résultat par 4 pour obtenir la valeur, il y a ensuite demande du calcul de `(fact 2)` et mise en mémoire du fait qu'il faudra multiplier le résultat par 3, il y a ensuite successivement demande du calcul de `(fact 1)` puis `(fact 0)` et mise en mémoire des multiplications par 2 et 1. Ceci se présente sous la forme d'un schéma (voir Figure 3.1).
Une fonction récursive terminale est une fonction récursive dans laquelle le nom de la fonction intervient dans la définition de celle-ci mais pas à l'intérieur d'une opération. Un exemple de fonction récursive terminale est donné dans le calcul du dernier élément d'une liste:

```
(define (dernier u)
    (cond ((null? u) "erreur")
          ((null? (cdr u)) (car u))
          (else (dernier (cdr u)))
    )
)
```

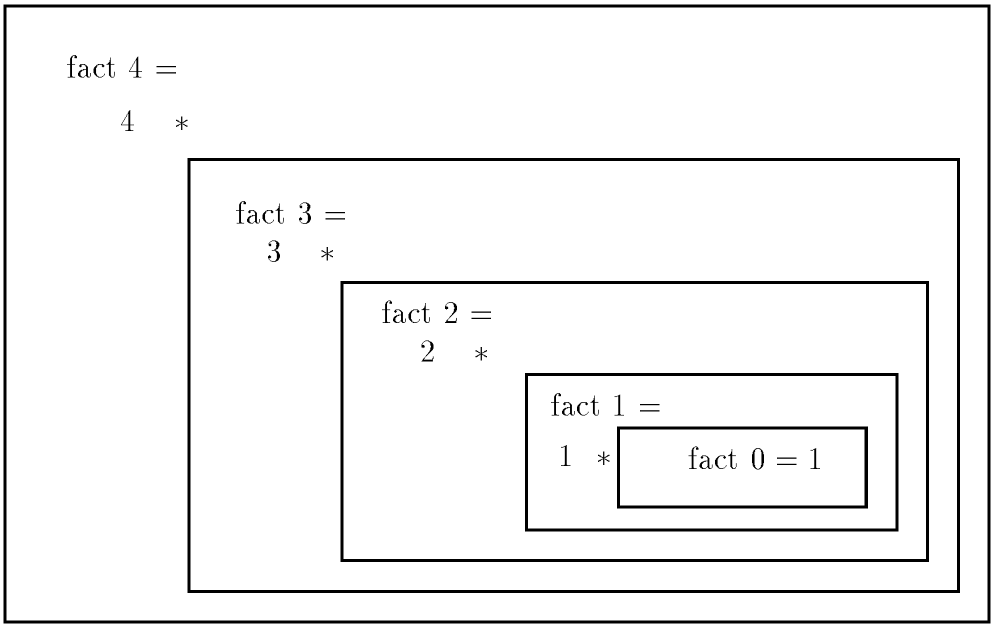


FIG. 3.1 - : Appels successifs lors du calcul de (fact 4)

On peut, pour améliorer l'efficacité des fonctions récursives et lorsque celles-ci sont utilisées pour des valeurs donnant lieu à un très grand nombre d'appels récursifs emboîtés les écrire sous forme récursive terminale. Ecrire une fonction sous forme récursive terminale [2] nécessite un savoir faire acquis à l'aide d'exemples typiques. Quelques techniques ont été mises en évidence. Une des plus utiles consiste à *généraliser la définition initiale* en y ajoutant un ou plusieurs paramètres supplémentaires. Dans notre exemple de factorielle n; on introduit le paramètre i qui croit de 1 à n et la variable a qui vaut $i!$. Cela donne une fonction `(fact-aux i a)` qui lorsqu'elle est appelée satisfait toujours $a = i!$, à chaque appel i est incrémenté de 1 et la valeur de a est multipliée par i.

```
(define (fact n)
    (define (fact-aux i a)
        (if (= i n) a
            (fact-aux (+ i 1) (* a (+ i 1))))
    )
    (fact-aux 1 1)
)
```

On peut aussi considérer que `(fact-aux i a)` calcule la fonction

$$a(i+1)(i+2)\dots(n-1)n$$

qui vaut bien $n!$ si $a = i = 1$ et qui est donnée par la récurrence:

```
(fact-aux i a)= (fact-aux i+1 a(i+1))
(fact-aux n a) = a
```

[2]il n'existe pas d'algorithme permettant d'améliorer automatiquement une définition récursive en la rendant récursive terminale

C'est cette récurrence qui est traduite littéralement dans la fonction écrite en Scheme plus haut.

Une autre façon de voir consiste à introduire la fonction g définie par

$$g(n,a) = n! \times a$$

on peut remarquer que

$$n! = g(n,1)$$

d'autre part elle est facile à calculer, en effet

$$g(0,a) = a$$

$$g(n,a) = g(n-1, n.a) \quad n > 0$$

Ceci se traduit directement sous la forme

```
(define (fact n)
    (define (g n a)
        (if (= n 0) a
            (g (- n 1) (* n a)))
    )
    (g n 1)
)
```

Dans les deux formes précédentes la variable `a` est appelée accumulateur.

Remarques
1. Il n'est pas toujours indispensable de transformer sous forme récursive terminale les fonctions récursives en Scheme, bien que cela améliore leur efficacité, l'inconvénient réside dans une certaine perte de clarté de la forme récursive terminale par rapport à la récursivité qui découle souvent directement de la définition de la fonction. Prenons l'exemple du calcul de la fonction `longueur` donnée dans le chapitre sur les listes

```
>(define (longueur x)
         (if (null? x) 0
             (+ 1 (longueur ( cdr x)))
         )
 )
```

Cette fonction n'est pas récursive terminale car l'appel récursif à `longueur` dans le corps de la fonction s'effectue à l'intérieur d'un `+` et il faut donc empiler ces additions pour le retour. Si on veut la rendre récursive terminale, il faut introduire une nouvelle variable `a` qui contient la longueur d'une partie de la liste; ceci donne la fonction suivante qui est moins claire

```
>(define (longueur x)
     (define (longueur-aux a x)
         (if  (null? x) a
```

```
            (longueur-aux (+ 1 a)  ( cdr x))
        ))
    (longueur-aux 0 x)
 )
```

2. La transformation sous forme terminale peut devenir excessivement utile si on en profite aussi pour supprimer des doubles rappels récursifs contenus dans une fonction comme par exemple la suite de Fibonacci donnée par:

```
(define (fib n)
        (if (or (= n 0) (= n 1)) 1
            (+  (fib (- n 1)) (fib (- n 2)) )
       )
)
```

Pour rendre cette fonction récursive terminale on introduit trois nouveaux paramètres: i qui croît de 0 à n au cours des différents appels, a qui vaut $fib(i)$ et b qui vaut $fib(i-1)$ on a alors:

```
(define (fib n)
    (define (fib-aux i a b)
        (if  (= n i) a
            (fib-aux (+ i 1) (+ a b) a)
        ))
     (fib-aux 1 1 1)
)
```

Dans ce cas on a à la fois supprimé le double appel récursif et rendu la récursivité terminale, cette transformation est donc particulièrement intéressante. Remarquons que a et b dans le programme précédent peuvent être considérés comme deux accumulateurs.

Chapitre 4

Listes généralisées

Les listes plates que nous avons introduites au chapitre 4 peuvent être généralisées. On définit ainsi des listes composées de listes, et on obtient ainsi un équivalent pour les structures de données à la récursivité des fonctions. Dans ces conditions les opérations `car` `cons` et `cdr` s'appliquent à des listes et ont pour résultat des listes. Nous verrons le parti que l'on peut tirer de cette structure pour diverses applications comme par exemple la réalisation des bases de données et la programmation symbolique. Dans ce dernier cas on considère les programmes comme des listes c'est à dire des données pour d'autres programmes.

4.1 Définition et fonctions élémentaires

On définit une liste comme une expression formée de:

- Une parenthèse ouvrante
- Une suite composée d'atomes ou de listes séparés par des espaces
- Une parenthèse fermante

Par exemple les expressions suivantes sont des listes

```
(1 4 5 7)
()
( (1 3 4) (5 2) 1 ())
(((((4 6 7))))
(+ (* 7 8) (* 9 5))
```

On s'aperçoit ainsi que tous les programmes que l'on peut écrire en Scheme sont des listes. C'est une des forces de ce langage que de pouvoir considérer un programme comme une donnée possible pour un autre.

Mais attention si l'on souhaite qu'une variable `u` ait pour valeur une liste donnée, il faut procéder comme pour les listes plates et faire précéder cette donnée d'une apostrophe afin d'empêcher l'évaluation de l'expression qui décrit la liste. Par exemple, si on définit la liste `u` suivante, en omettant l'apostrophe, l'interprète va considérer que `u` est un nombre comme le montre le résultat de la session suivante:

```
> (define u (* (+ 3 4) 1))
U
> u
7
```

Par contre, si on souhaite que `u` soit une liste, il faut écrire la définition suivante:

```
> (define u '(* (+ 3 4) 1))
U
> u
(* (+ 3 4))
```

Les opérations `car` et `cdr` sont définies sur les listes généralisées de la façon suivante: si `u1, u2, .. up` sont des listes alors `(u1 u2 ...up)` est aussi une liste et on a:

`(car (u1 u2 ...up))` = `u1` et `(cdr (u1 u2 ...up))` = `(u2 ...up)`

L'exemple de session ci-dessous illustre ces fonctions:

```
> (define u '(* (+ 5 6)(+ 3 4)))
U
> (car u)
*
> (cdr u)
((+ 5 6)(+ 3 4))
> (car (cdr u))
(+ 5 6)
> (cdr (cdr u))
((+ 3 4))
```

La construction des listes se fait aussi avec la fonction `cons` qui vérifie les mêmes propriétés que dans le cas des listes plates soit:

```
(car (cons u v)) = u     (cdr (cons u v)) = v      (cons (car u) (cdr u)) = u
```

Un exemple d'utilisation de la fonction `cons` est le suivant:

```
> (define u '((+ 5 6)(+ 3 4)))
U
> (define v '( + 7 8))
V
> (cons u v)
(((+ 5 6)(+ 3 4)) + 7 8)
> (cons v u)
>(+ 7 8 (+ 5 6)(+ 3 4)))
```

La liste vide`()` joue un rôle identique à celui qu'elle joue dans le cadre des listes plates.

4.2 Fonctions récursives sur les listes généralisées

On peut définir sur le même principe que pour les listes plates des fonctions récursives sur les listes généralisées. La base de l'induction est toujours la liste vide `()`, et l'on définit l'action d'une fonction sur une liste `x` à partir de son action sur `(car x)` et `(cdr x)`. Nous donnons ici quelques exemples. La notion de longueur a en fait deux généralisations pour les listes quelconques, on peut soit considérer que la longueur de la liste `(u1 u2 ...up)` est p soit que c'est le nombre d'atomes qui figurent dans `(u1 u2 ...up)`. Ceci donne lieu à deux fonctions que nous appellerons respectivement `longueur` et `nombre-atomes`.
En ce qui concerne `longueur`, pour démarrer la récursivité on peut noter que la longueur de la liste vide est égale à 0 et que la longueur de `u` est égale à la longueur de `(cdr u)` augmentée de 1. Ceci donne en Scheme un fonction identique mot pour mot à celle que nous avons donnée pour les listes plates:

```
> (define (longueur u)
       (if (null? u) 0
           (+ 1 (longueur (cdr u)) )
       )
)
```

Pour compter le nombre d'atomes de la liste `u`, on remarque assez simplement que si `u` est la liste vide alors ce nombre est 0 sinon c'est la somme du nombre d'atomes de `(car u)` et de celui de `(cdr u)`. Mais cette remarque ne suffit pas puisque si `u` est une liste, tel n'est pas nécessairement le cas de `(car u)` qui peut être réduite à un seul atome. Il faut donc prévoir le cas où l'on calcule le nombre d'atomes d'un atome, ce nombre est bien entendu égal à 1. On obtient finalement la fonction suivante en Scheme:

```
> (define (nombre-atomes  u)
       (cond ((null? u) 0)
             ((atom? u) 1)
             (else (+ (nombre-atomes (car u))  (nombre-atomes (cdr u))))
       )
)
```

Une autre fonction intéressante sur les listes généralisées est la notion de *profondeur* celle-ci décompte le nombre maximum de parenthèses imbriquées qui se trouvent dans la liste. Ainsi la profondeur de la liste vide est 0, et celle d'une liste `u= (u1 u2 ...up)` quelconque, 1+ la plus grande profondeur des `ui`. Dans ce décompte il faut considérer que la profondeur d'un atome est 0. Pour écrire la fonction Scheme correspondante, on remarque que si une liste `u` n'est pas vide, la profondeur de u est égale au maximum de la profondeur de (cdr u) et de 1+ la profondeur de (car u).

```
> (define (maxi a b)
      (if (> a b) a b))


> (define (profondeur u)
       (if (or (null? u) (atom? u)) 0
           (maxi (profondeur (cdr u))  (+ 1 (profondeur (car u))))
       )
)
```

Deux autres fonctions permettant de construire des listes sont étendues des listes plates aux listes généralisées. Il s'agit des fonctions append et list.

La fonction append appliquée à deux listes généralisées (u1 u2 ...up) et (v1 v2 ...vq) donne la liste (u1 u2 ...up v1 v2 ...vq). Ainsi la longueur de (append u v) est égale à la somme des longueurs de u et de v; il en est de même pour les nombres d'atomes qui s'ajoutent. La profondeur de (append u v) est égale à la plus grande des profondeurs des deux listes u et v. La fonction append peut s'appliquer à plus de deux listes avec une signification évidente.

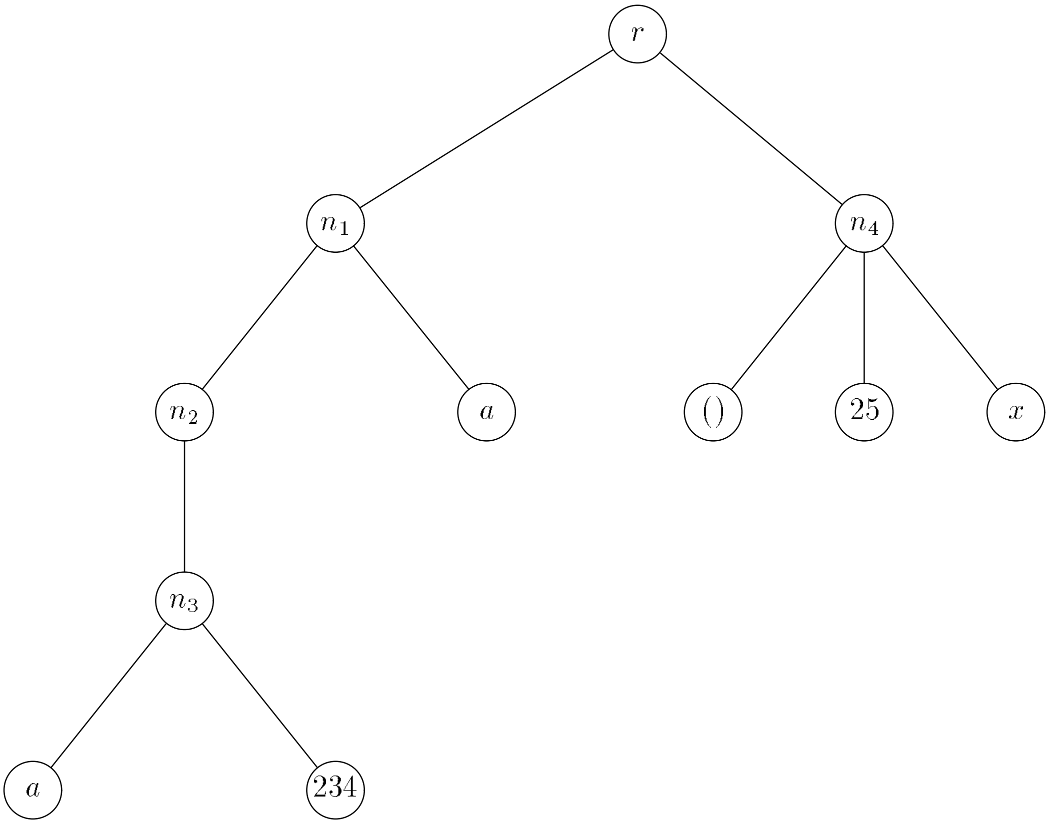
La fonction list appliquée à n listes généralisée u1, u2, ... un donne la liste (u1 u2 ... un). Cette fonction est différente de append, en effet pour les deux listes (u1 u2 ...up) et (v1 v2 ...vq) list donne la liste ((u1 u2 ...up) (v1 v2 ...vq)) (alors que append donnait: (u1 u2 ...up v1 v2 ...vq)). Ainsi la longueur de (list u v) est égale à 2 et les nombres d'atomes de u et v s'ajoutent dans (list u v). Enfin la profondeur de (list u v) est égale à la plus grande des profondeurs des deux listes u et v augmentée de 1. Un exemple de session illustrant les constructions précédentes

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

> (list '(a (b c) (d)) '( (b d) (a e) f) )
((a (b c) (d)) ((b d) (a e) f))

> (longueur  (append '(a (b c) (d)) '((b d) (a e) f) ) )
6
> (longueur   (list '(a (b c) (d)) '((b d) (a e) f)) )
2
> (profondeur  '( a (b c) (d)) )
2
> (profondeur '( (b d) (a e) f) ))
2
> (profondeur  (append '(a (b c) (d)) '((b d) (a e) f) ) )
```

```
2
> (profondeur  (list '(a (b c) (d)) '((b d) (a e) f) ) )
3
>
```

4.3 Représentation des listes par des arbres

Un arbre est une structure de base en informatique. C'est un objet composé d'une *racine* r, de *nœuds* et de *feuilles*. Chaque nœud possède un certain nombre de *fils* qui sont des nœuds ou des feuilles, les feuilles n'ont pas de fils. Chaque feuille possède une étiquette qui est un atome. Un *chemin* est une suite $x_1, x_2, \ldots x_p$ telle que pour tout $i > 1$, x_i soit un fils de x_{i-1}. Dans un arbre, pour chaque élément x de l'arbre (nœud ou feuille), il existe un chemin *unique* qui mène de la racine r à x.
Un arbre se représente par un schéma sur lequel sont illustrées toutes les notions décrites ci-dessus. Ainsi sur la figure ci-dessous on a un arbre qui contient 5 nœuds (dont la racine) et 6 feuilles.



On peut aussi définir l'ensemble des arbres de façon récursive ce qui montre bien le lien avec les listes:

> Un arbre est ou bien composé d'une seule feuille, ou bien formé d'une racine et d'une suite de sous arbres.

A chaque arbre on peut associer une liste par l'algorithme suivant:

- A une feuille est associé un atome portant le nom de la feuille
- A un arbre composé des sous arbres $A_1, A_2, \ldots A_i \ldots A_p$ est associé la liste `u = (u1 u2 ... ui ...up)` où pour tout $i = 1, p$, `ui` est la liste associée à l'arbre A_i.

Pour l'exemple d'arbre de la figure, la liste associée est;

```
(( (( a 234 )) a) ( () 25 x ) )
```

Réciproquement à toute liste est associé un arbre obtenu en inversant la construction précédente. On remarque que si A est un arbre associé à la liste `x`, alors la profondeur de `x` est égale à la longueur du plus long chemin qui mène de la racine de A à une feuille, le nombre d'atomes de `x` est égal au nombre de feuilles A et les fonctions `cons, car, cdr, append, list` s'illustrent par des dessins sur les arbres (voir Figures 4.1-4.4).

4.4 Un exemple d'utilisation de listes générales

Considérons une gestion informatique simplifiée d'étudiants. Chaque étudiant est représenté par une liste contenant les informations suivantes sous forme de plusieurs sous listes. Le nom et le prénom forment une première sous liste; l'année de naissance, et le diplôme préparé sont les deux informations suivantes, elles se présentent chacune sous une forme d'un atome. Les modules obtenus constituent la dernière information elle est représentée sous la forme d'une liste. Un exemple type est la liste suivante

```
((dupont alain) 1973 mi (m2 p2 i1 p5 m5 a1 ))
```

L'ensemble des étudiants constitue une liste de taille importante composée d'autant de sous-listes qu'il y a d'étudiants, on suppose dans la suite que cette liste est appelée `list-etud`. Examinons les différentes fonctions que l'on peut construire sur cet exemple

4.4.1 Recherche d'informations

Un première fonction consiste à rechercher toutes les informations concernant un étudiant dont le nom ou le prénom est donné. Il faut commencer par écrire les fonctions *sélecteurs* d'information sur la sous liste formée d'un étudiant. Cela donne[1]:

```
(define (nom x) (caar x))
```

[1] rappelons que (caar x) est une abréviation de (car (car x)) etc ...

FIG. 4.1 - : Le car et cdr d'une liste sur l'arbre associé

FIG. 4.2 - : La construction d'une liste par cons

FIG. 4.3 - : La fonction append sur les arbres associés

FIG. 4.4 - : La fonction list sur les arbres associes

```
(define (prenom x) (cadar x))
(define (naissance x) (cadr x))
(define (diplome x) (caddr x))
(define (modules-obtenus x) (car (cdddr x)))
```

Cet ensemble de sélecteurs permet d'accéder à chaque information séparément concernant un étudiant donné. Il est maintenant possible de rechercher un étudiant de nom donné x en parcourant toute la liste `list-etud`, cela donne:

```
(define (recherche-nom x lis)
         (cond ((null? lis) "Il n y a pas ce nom la")
               ((eqv? (nom (car lis)) x) (car lis))
               (else (recherche-nom x (cdr lis)))
         )
)
```

L'utilisation se fait de la façon suivante:

```
> (recherche-nom 'dupont list-etud)
```

qui répond

```
((dupont alain) 1973 mi (m2 p2 i1 p5 m5 a1))
```

4.4.2 Statistiques

Une des applications importante de la gestion informatisée consiste à pouvoir obtenir des données statistiques. Par exemple rechercher le nombre d'étudiants ayant un module donné. Ceci peut se faire à l'aide de la fonction suivante qui utilise la fonction `appartient` définie sur les listes plates dans le Chapitre 4 de la Partie 1:

```
(define (a-le-module m x) (appartient m (modules-obtenus x)))
(define (stat-module  m lis )
 (cond ((null? lis) resultat)
       ((a-le-module m (car x)) (+ 1 (stat-module m (cdr x)))  )
       (else (stat-module m (cdr x)))
 )

)
```

Beaucoup d'autres exemples de calcul statistiques peuvent être effectués, ceux-ci sont laissées à titre d'exercice.

4.4.3 Constitution de sous listes

Une fonction d'un type différent consiste à construire la sous liste des étudiants nés une année donnée ou préparant un diplôme donné cela donne:

```
(define (recherche-annee a lis resultat)
 (cond ((null? lis) resultat)
       ((eqv? (naissance (car lis)) a) (recherche-annee a (cdr lis)
                                        (cons (car lis) resultat)))
       (else (recherche-annee a (cdr lis) resultat)))
)
```

```
(define (recherche-naissance a)
         (recherche-annee a list-etud '()))
```

Ou pour

```
(define (recherche-dip a lis resultat)
 (cond ((null? lis) resultat)
       ((eqv? (diplome (car lis)) a) (recherche-dip a (cdr lis)
                                         (cons (car lis) resultat)))
       (else (recherche-dip a (cdr lis) resultat)))
)
```

```
(define (recherche-diplome a)
         (recherche-dip a list-etud '()))
```

4.4.4 Mise-à-jour d'information

Une fonctionnalité importante pour le type d'application étudié consiste à pouvoir modifier la liste `list-etud` en fonction d'éléments nouveaux intervenant comme par exemple l'ajout d'un étudiant dans la liste, le changement de la situation d'un étudiant: obtention d'un nouveau module principalement. Pour cela il faut redéfinir la variable `list-etud` comme cela est fait dans l'exemple ci-dessous:

```
(define x '((martin pierre) 1972 pi (m3 p6 i1 a8)))

(define list-etud (cons x list-etud))
```

La mise à jour faisant intervenir une modification concernant un étudiant donné est plus compliquée.

Partie III

Algorithmes

Chapitre 1

Techniques de tri

Le problème du tri est un des grands classiques de la programmation. Il donne lieu à des développements intéressants d'un point de vue algorithmique: apprendre à écrire un programme de tri est une excellente préparation pour s'entraîner à résoudre des problèmes plus complexes. C'est en effet l'occasion de rencontrer des algorithmes de natures diverses dont les paradigmes se rencontrent aussi dans beaucoup d'autres situations. Sur le plan pratique, de nombreuses applications nécessitent de trier des éléments d'un ensemble, citons en quelques unes: tri d'une liste d'étudiants suivant un ordre alphabétique, tri d'un ensemble de villes suivant leur nombre d'habitants, présentation d'un tableau de données statistique par ordre d'indice croissants, etc ..

Dans ce chapitre on se contentera de résoudre un problème présenté sous une forme réduite simple. On se donne un ensemble E, muni d'une relation d'ordre totale que l'on note $\prec$ et qui dans la programmation sera notée `Rel`. On souhaite réaliser une fonction qui à une liste d'éléments appartenant à E associe une liste contenant les mêmes éléments, mais en ordre croissant. On prendra pour convention que si un élément figure en plusieurs exemplaires dans la liste à trier il doit se retrouver le même nombre de fois dans celle triée. La relation d'ordre `Rel` est supposée définie par une fonction séparée. Pour les questions évoquées ci-dessus il faudra construire une fonction `Rel` particulière au problème donné, dans le cas où l'on souhaite trier des nombres en ordre croissant il faut donner la valeur `<` au paramètre `Rel`, et la valeur `>` pour le cas de nombres en ordre décroissant.

La fonction Scheme à écrire est illustrée par l'exemple d'utilisation suivant :

```
> (Tri  < '(45  56 1 34 21 76 2 56  21 89 23 56))
(1 2 21 21 23 45 56 56 56 76 89)
```

Il y a plusieurs algorithmes classiques de tri, nous ne les étudions pas tous; nous nous limitons au tri par insertion, au tri par sélection et à deux algorithmes plus efficaces: le tri par fusion et celui communément appelé Quicksort. Ceux-ci suffisent à donner un panorama de l'ensemble des algorithmes existants. Dans les langages de programmation impératifs ces algorithmes sont réalisés à l'aide de tableaux, nous les présentons en détail ici sur les listes. Une écriture en utilisant les vecteurs de Scheme est fournie plus loin, cette dernière s'apparente beaucoup plus à la version donnée habituellement dans le cadre des langages impératifs. L'expérience montre que le temps de calcul n'est pas significativement meilleur sur les vecteurs de Scheme que sur ses listes, et ceci même pour les algorithmes utilisant largement l'accès aux $i - eme$ élément d'un tableau de façon systématique.

De toutes façons , c'est l'aspect algorithmique qui nous intéresse ici et Scheme avec ses listes est particulièrement agréable pour comprendre les principes de fonctionnement d'un algorithme de tri.

1.1 Tri par Insertion

1.1.1 Description

La technique utilisée dans ce tri consiste e à insérer successivement chaque élément de la liste `lu` dans une liste `lw` qui est initialisée à `'()`. La liste `lw` est une liste triée, elle augmente de taille au fur et à mesure où celle de `lu` diminue. A chaque étape du tri l'élément `(car lu)` est inséré à sa bonne place dans la liste `lw` (celle qui conserve la propriété d'être triée pour `lw`), et la liste `lu` est remplacée par `(cdr u)`. Ceci nous amène à revoir l'insertion d'un élément dans une liste triée que nous avons vue dans le premier chapitre sur les listes. Si la liste est vide on constitue une liste formée seulement de cet élément sinon on agit différemment suivant que l'élément à ajouter est plus petit que le premier élément la liste ou pas, dans le premier cas on l'insère en tête à l'aide d'une opération `cons` dans le second on réitère l'opération sur le `cdr` de cette liste. Ce qui donne:

```
  (define (Liste-triee-insert Rel a lu);; insertion dans une liste ordonne'e
                                        ;; de   l'e'le'ment a
      (cond ((null? lw )    (list a))
            ((Rel  a (car lu))  (cons a lu))
            (else (cons (car lu) (Liste-triee-insert Rel a (cdr lu)))))
  )
```

La procédure de tri auxiliaire utilise la fonction d' insertion que nous venons d'écrire et les deux listes `lv` et `lw`, la première diminue de taille et la seconde grossit. Enfin le tri lui même consiste à initialiser le processus en démarrant avec la liste `v` vide ::

```
(define (Tri-insert Rel lu)
    (define (aux lv lw)      ;; on fait passer les e'le'ments de lv vers lw
        (if (null? lv)  lw
            (aux (cdr lv) (Liste-triee-insert Rel (car lv) lw)))
    )
    (aux lu '())
)
```

:

1.1.2 Complexité

Pour mesurer la complexité d'un algorithme de tri, on détermine classiquement le nombre total de comparaisons nécessaires pour effectuer le tri de n éléments. Ceci se traduit dans le cadre de la programmation donnée plus haut comme le nombre de fois où la fonction `Rel` est appelée. En effet on considère que le nombre des autres opérations est lié linéairement à celui des comparaisons : si le nombre de comparaisons à effectuer pour trier une liste de longueur n, est égal à $f(n)$, le nombre d'opérations d'accès est le plus souvent de la forme $Cf(n)$ où C est une constante qui dépend de l'algorithme et qui ne dépasse pas 10 en général. Or ce qui nous intéresse c'est l'ordre de grandeur de $f(n)$ lorsque n est grand, cet ordre de grandeur ne change pas si on le multiplie par une constante.

Illustrons ces remarques sur l'exemple du tri par insertion. Commençons par étudier le nombre d'opérations effectuées par la procédure `Liste-triee-insert` en fonction de la longueur, notée ici p de la liste où l'on doit insérer l'élément `a`. Dans le meilleur des cas, si `(Rel a (car lu)` est égal à *vrai*, ou de manière équivalente si `a` est "plus petit " que tous les éléments de la liste, il n'y a qu'une comparaison effectuée. Le pire des cas est celui où l'élément `a` est "plus grand" que tous les éléments de la liste, le nombre de comparaisons est alors p. On note que dans cette procédure, le nombre d'appels à `car` est deux fois celui des comparaisons, et le nombre des appels à `cdr` ou `cons` est égal au nombre de comparaisons ce qui confirme la remarque précédente. Nous sommes en mesure de déterminer le nombre de comparaisons de l'algorithme de tri par insertion pour une liste de longueur n dans le pire des cas, c'est à dire celui où l'insertion s'effectue toujours en queue de la liste. La fonction `(aux lv lw)` est appelée n fois la liste `lw` est d'abord vide puis sa longueur augmente d'une unité à chaque appel. Lors d'un appel on effectue `List-triee-insert` sur la liste `lw` ainsi, le nombre total de comparaisons est

$$\sum_{p=0}^{n-1} p = \frac{n(n-1)}{2}$$

Ce nombre de comparaisons dans le cas le pire ne donne pas une information complète sur la complexité de l'algorithme. Il faut aussi déterminer le nombre moyen de comparaisons, c'est à dire faire la moyenne du nombre de comparaisons sur toutes les distributions possibles de suites d'éléments à trier. Le nombre moyen de comparaisons pour insérer un élément dans une liste de longueur p est dans ces conditions $p/2$, car on peut regrouper les cas où on insère en position i qui demande i comparaisons et celui où l'on insère en position $p-i$ qui en demande $p-i$. La complexité moyenne de l'algorithme de tri par insertion s'obtient comme il a été fait pour la complexité dans le cas le pire en effectuant la somme des complexités de l'insertion dans une liste triée pour les longueurs $0, 1 \cdots (n-1)$ soit :

$$\sum_{p=0}^{n-1} \frac{p}{2} = \frac{n(n-1)}{4}$$

1.2 Tri par sélection

1.2.1 Description

Cet algorithme consiste à sélectionner le plus grand élément de la liste `lu` à trier, le supprimer de cette liste et de constituer une liste `lw` qui ne contient que lui. Ensuite on répète l'opération de recherche du plus grand élément de `lu`, puis de suppression de cet élément de `lu` et d'ajout en tête de `lw`. Dans cette méthode aussi la liste `lw` augmente de taille alors que `lu` diminue. Le travail le plus long consiste cette fois à rechercher ce plus grand élément dans la liste `lu` et de le supprimer; par contre l'ajout de cet élément à `lw` est très simple puisqu'il s'effectue en tête, et la fonction `cons` fait bien l'affaire.

La recherche de l'élément le plus grand d'une liste a été vue dans le chapitre sur les listes. Nous modifions légèrement cette fonction pour faire quelle construise une liste qui contient les mêmes éléments mais dans laquelle le plus grand élément se retrouve en tête. Si la liste `lu` ne comporte qu'un seul élément, il n'y aura bien entendu rien à faire, sinon il faut comparer `(car lu)` et le plus grand élément de `(cdr lu)` et mettre le plus grand des deux en tête pour obtenir le résultat. On donne ci-dessous la programmation de cette fonction en supposant que l'on se trouve dans un environnement où la relation `Rel` est définie.

```
(define (Plus-grand-en-tete  lu)
    (if (null? (cdr lu)) lu
        (let* ((lv (Plus-grand-en-tete (cdr lu))) (b (car lu)) (a (car lv)))
            (if (Rel a b) (cons b (cons a (cdr lv)))
                          (cons a (cons b (cdr lv))))))

)
```

Une fois écrite cette fonction, le tri par sélection devient simple à mettre en œuvre, il suffit de faire comme pour le tri par insertion en faisant grossir une liste `lw` l'insertion consiste simplement à utiliser la fonction `cons`, car on sait que l'on insère les éléments dans l'ordre décroissant de leurs valeurs.

```
(define (Tri-select Rel lu)
    (define (Plus-grand-entete lv)
        (if (null? (cdr lv)) lv
            (let* ((lw (Plus-grand-en-tete (cdr lv))) (b (car lv)) (a (car lw)))
                  (if (Rel a b) (cons b (cons a (cdr lw)))
                                (cons a (cons b (cdr lw))) )))
    )


    (define (aux lu lv)
        (if (null? lu) lv
            (let ((lw (Plus-grand-en-tete  lu)))
                (aux (cdr lw) (cons (car lw) lv)))))
    )


   (aux lu '())
)
```

Complexité Commençons par examiner la fonction `Plus-grand-en-tete`, lorsqu'elle opère sur une liste de longueur $p, p > 1$ elle effectue une comparaison et s'appelle récursivement sur une liste de longueur $p-1$. Le nombre de comparaisons pour mettre le plus grand en tête est donc égal à $p-1$ et ceci quelle que soit la distribution des éléments dans la liste.

Pour la fonction `Tri-select`. Chaque étape de `aux` consiste à mettre en tête le plus grand élément d'une liste dont la taille diminue au fur et à mesure de l'exécution de l'algorithme. Ainsi le nombre de comparaisons tri par sélection d'une liste de taille n est :

$$\sum_{p=1}^{n} p-1 = \frac{n(n-1)}{2}$$

On remarque que la complexité en moyenne est la même que la complexité dans le cas le pire, car l'algorithme effectue systématiquement toutes les comparaisons quelle que soit la liste. Ainsi le nombre de comparaisons en moyenne est de l'ordre de 2 fois supérieur à celui du tri par insertion .

Remarque. On constate en pratique sur des listes constituées de nombres entiers tirés au hasard et que l'on trie suivant l'ordre habituel que le temps mis par le tri par sélection est trois fois supérieur à celui mis par le tri par insertion. Ceci est dû au fait que les opérations sur les listes sont dans le cas de la sélection plus nombreuses que dans celui de l'insertion (en particulier il y a deux fois plus d'appels à `cons`) cela influe donc sur le temps effectif. Par contre les expérimentations réalisées pour des relations `Rel` qui demandent un temps de calcul important donnent exactement le coefficient 2 dans les rapports entre les temps. Il est aussi intéressant de constater que le temps mis pour le tri par sélection est pratiquement constant pour des listes tirées au hasard de me6me taille alors que les variations sont plus sensibles pour le tri par insertion.

1.3 Tri par Fusion

1.3.1 Description

Cette méthode de tri est récursive, on remarque d'abord que si la liste à trier est vide ou si elle ne contient qu'un seul élément, il n'y rien à faire pour la trier. Dans les autres cas la liste est divisée en deux parties sensiblement égales, en mettant les éléments de rang impair dans une liste `limpairs` et ceux de rang pair dans une autre `lpairs`. Chacune de ces deux listes est ensuite triée en appliquant récursivement l'algorithme; on fusionne enfin les deux listes triées. La division est simple à réaliser par parcours de la liste à trier et suppression de deux éléments à chaque itération lesquels sont placés dans deux listes `limpairs` et `lpairs`.

L'écriture de la fonction qui fusionne deux listes triées demande un peu plus de réflexion mais ne présente pas de difficulté particulière, en effet il suffit de construire un processus itératif qui supprime le plus petit élément de l'une des deux listes pour le placer en tête de la liste fusion et de terminer quand l'une des deux listes est vide.

Une remarque supplémentaire, lorsque la division est terminée les deux listes obtenus sont stockées dans une liste `listdiv` qui contient chacune des deux comme sous-liste. Pour cela effectue la construction `(list lpairs limpairs)`, l'accès a `lpairs` se fait par `(car listdiv)` et celui à `limpairs` par `(cadr listdiv)`.

```
(define (Tri-fusion Rel lu)

    (define (Fusion lu lv) ;; fusion de deux listes trie'es lu et lv
        (cond ((null? lu) lv)
      ((null? lv) lu)
      ((Rel (car lu) (car lv)) (cons (car lu)  (fusion (cdr lu) lv)))
      (else (cons (car lv)  (fusion lu  (cdr lv)))))
    )

    (define (Division lu lpairs limpairs)
            (cond ((null? lu) (list lpairs limpairs))
                  ((null? (cdr lu)) (list  (cons (car lu) lpairs) limpairs))
                  (else (Division (cddr lu)
                                  (cons (car lu) lpairs)
                                  (cons (cadr lu) limpairs))))
    )

        (if (or (null? lu) (null? (cdr lu))) lu
            (let ((listdiv (Division  lu '() '())))
                (Fusion
                    (Tri-fusion Rel (car listdiv) )
                    (Tri-fusion Rel (cadr listdiv)))))
)
```

Complexité La fusion de deux listes demande un nombre de comparaisons égal à la somme des longueurs des deux listes. La division d'une liste de longueur n ne demande aucune comparaison, par contre elle nécessite n opérations `cons` dont il faudrait tenir compte dans une évaluation du temps effectif de calcul mais qui ne change rien quand à la fonction donne la complexité. Si on note f_n le nombre de comparaisons nécessaires au tri par fusion d'une liste de longueur n on a $f_1 = 1$ et

$$f_n = n + 2f_{n/2}$$

où $n/2$ désigne la valeur entière par excès du quotient de n par 2. Cette récurrence n'est pas immédiate à obtenir, on peut commencer à calculer sa valeur lorsque n est une puissance de 2, posons ainsi $n = 2^p$ et $f_n = g_p$ dans ce cas, on obtient la récurrence $g_0 = 0$:

$$g_p = 2^p + 2g_{p-1}$$

Ainsi on obtient

$$g_p = p2^p$$

Pour une valeur quelconque de n on peut noter p la valeur de la partie entière de $\log_2(n)$ on a alors :

$$2^{p-1} < n < 2^p$$

Il est facile de vérifier que le nombre de comparaisons de l'algorithme croît avec n ainsi :

$$g_{p-1} < f_n \leq g_p$$

et

$$(p-1)2^{p-1} < f_n \leq p2^p$$

soit

$$f_n \equiv nlog_2(n)$$

1.4 Tri rapide

Cette méthode consiste à considérer le premier élément `a` de la liste et à partager la liste en deux autres listes contenant l'une les éléments plus petits que `a` l'autre les éléments plus grands. On trie ensuite récursivement chacune des deux listes ainsi obtenues et on les concatène en mettant `a` entre les deux. Ce style d'algorithme consistant à diviser une donnée en deux parties à peu près égales pour reconstruire la solution à partir des solutions sur chacune des deux parties s'appelle *Diviser pour Régner* L'algorithme ne fait pas d'appel récursif quand la liste à trier est vide ou bien quand elle ne contient qu'un seul élément. La fonction `Partition-aux` découpe la liste en deux, la procédure `Quick-tri` effectue le tri récursivement.

```
 (define (Partition-liste Rel lu)
    (let ((a (car lu)))
        (define (aux lu lpti lgrd)
            (if (null? lu) (list lpti lgrd)
                (let ((b (car lu)) (lv (cdr lu)))
                    (if (Rel  b  a) (aux lv (cons b lpti) lgrd)
                        (aux lv lpti (cons b lgrd))))))
         )
         (aux (cdr lu) '() '()))
)
(define (Tri-quick-sort Rel lu)
     (if (or (null? lu) (null? (cdr lu))) lu
         (let ((listpart (Partition-liste Rel  lu)) (a (car lu)))
             (append (Tri-quick-sort Rel (car listpart))
                     (list a)
                     (Tri-quick-sort Rel (cadr listpart)))))
)
```

1.4.1 Complexité

1.5 Tri de vecteurs

```
(define (vector-Tri-insert Rel v  n)
    (do ((i 1 (+ 1 i)))
        ((= i n) v)
        (let ((a (vector-ref v i)))
             (do ((j i (- j 1)))
                 ((or (= j 0) (Rel (vector-ref v (- j 1)) a))
                 (vector-set! v j a) )
                 (vector-set! v j (vector-ref v (- j 1))))
        )
     )
)


;; Tri nume'ro 6 s\'election sur vecteurs version let nomme'


(define (vector-Tri-select Rel v)
 (let ((n (- (vector-length v) 1)))

   (define (update-v! i x j y) ; puts x in v[i] and y in v[j]
     (vector-set! v i x)
     (vector-set! v j y))

  (define (find-minimum-and-exchange! from)
     (let ((val0 (vector-ref v from)))
       (let parcours ((i from)
      (m (vector-ref v from))
      (j (+ 1 from))) ; indice courant
           (if (> j n)
       (when (< from i)
  (update-v! from m i val0))
       (let ((z (vector-ref v j)))
          (if (Rel z m)
     (parcours j z (+ j 1))
     (parcours  i m (+ j 1))))))))

   (define (trier from)
     (cond ((= from n) v)
   (else (find-minimum-and-exchange! from  )
 (trier (+ 1 from)))))
   (trier 0)))



;;;;; Tri nume'ro 60 S\'election sur vecteurs sans let nomme
```

```
(define (vector-Tri-select-bis Rel v n)
    (define (Echange! i j)
        (let ((a (vector-ref v i)))
             (vector-set! v i (vector-ref v j))
             (vector-set! v j a))
    )

    (define (Plus-petit-entete i a j k)
        (if (= j n) (Echange!  i k)
            (let ((b (vector-ref v j)))
                 (if (Rel a b) (Plus-petit-entete i a (+ j 1) k)
                     (Plus-petit-entete i b (+ j 1) j))))
    )

    (define (aux i)
          (if (= i n) v
              (begin (Plus-petit-entete i (vector-ref v i) (+ i 1) i)
                     (aux (+ i 1))))
    )

    (aux 0)
)


;;;;;;;;;;; Tri nume'ro 7 : fusion sur vecteurs
 (define (Recopie! v i j w k)
      (define (aux l m)
           (if (= l j) w
               (begin (vector-set! w m (vector-ref v l))
                      (aux (+ 1 l) (+ 1 m)))))
      (aux i k))

 (define (vector-fusion Rel v1 n1 v2 n2 v)
         (define (aux i i1 i2)
              (cond ((= i1 n1) (Recopie! v2 i2 n2 v i))
                    ((= i2 n2) (Recopie! v1 i1 n1 v i))
                    (else (let ((a (vector-ref v1 i1))
                                (b (vector-ref v2 i2)))
                           (if (Rel a b)
                               (begin (vector-set! v i a)
                                      (aux (+ i 1) (+ i1 1) i2))
                               (begin (vector-set! v i b)
                                      (aux (+ i 1) i1 (+ i2 1)))))))
           )
           (aux 0 0 0))

(define (vector-tri-fusion Rel v n)
```

```
      (if (= n 1) v

      (let*   ((n1 (truncate (/ n 2))) (n2 (- n n1))
               (v1 (make-vector n1)) (v2 (make-vector n2)))
      (begin
             (recopie! v 0 n1 v1 0)
             (vector-tri-fusion Rel v1 n1)
             (recopie! v n1 n v2 0)
             (vector-tri-fusion Rel v2 n2)
             (vector-fusion Rel v1 n1 v2 n2 v))))
)




;;;;;;;;Tri nume'ro 8 : quicksort sur des vecteurs


(define (vector-Partition Rel v deb fin)

      (define (aux  i j)
         (let ((a (vector-ref v deb)))
          (if (>= j fin) (begin (vector-set! v deb (vector-ref v i))
                                (vector-set! v i a) i)
               (let ((b (vector-ref v j)))
                            (if (Rel a b) (aux i (+ j 1))
                               (begin (vector-set! v j (vector-ref v (+ i 1)))
                                      (vector-set! v (+ i 1) b)
                                       (aux (+ i 1)(+ j 1))))
                            ))))
         (aux deb (+ deb 1))
)


 (define (vector-Tri-quick-sort Rel v n)
        (define (aux deb fin)
            (if (< (- fin deb) 1)  v
             (let ((i (vector-Partition  Rel v deb fin)))
                  (aux  deb i) (aux   (+ i 1) fin))))
        (aux 0 n)
)
```

Liste des figures