*Université Montpellier-II*
*UFR des Sciences - Département Informatique*

*Licence Informatique - 2ième année*

# Programmation Applicative et Récursive

Notes de cours - 2007-2012

Christophe Dony

# Table des matières

# Chapitre 1

# Introduction.

Ceci est le premier d'une série de 10 cours de 1h30 sur la programmation applicative et récursive, donnés en licence seconde année dans le contexte global de l'initiation à la programmation.

## 1.1 Définitions

**Programmation Applicative**

— Programmation avec laquelle un texte de programme est un ensemble de définition de fonctions et où l'exécution d'un programme est une succession, d'**application** de fonctions à des arguments au sens algébrique du terme (calcul utilisant des opérations, des lettres et des nombres) et selon la sémantique opératoire de l'application du *Lamba-Calcul*;

— programmation utilisant potentiellement (mais sans obligation) des variables mutables (par opposition à la programmation fonctionnelle),

— programmation où toute instruction est une expression (dont le calcul a une valeur).

**Programmation récursive** : programmation utilisant des fonctions récursives i.e. s'appelant elles-mêmes.

Une fonction récursive est appropriée pour implanter un algorithme pour tout problème auquel peut s'appliquer une résolution par récurrence, ou pour parcourir et traiter des stuctures de données elles-mêmes récursives telles que les listes et les arbres.

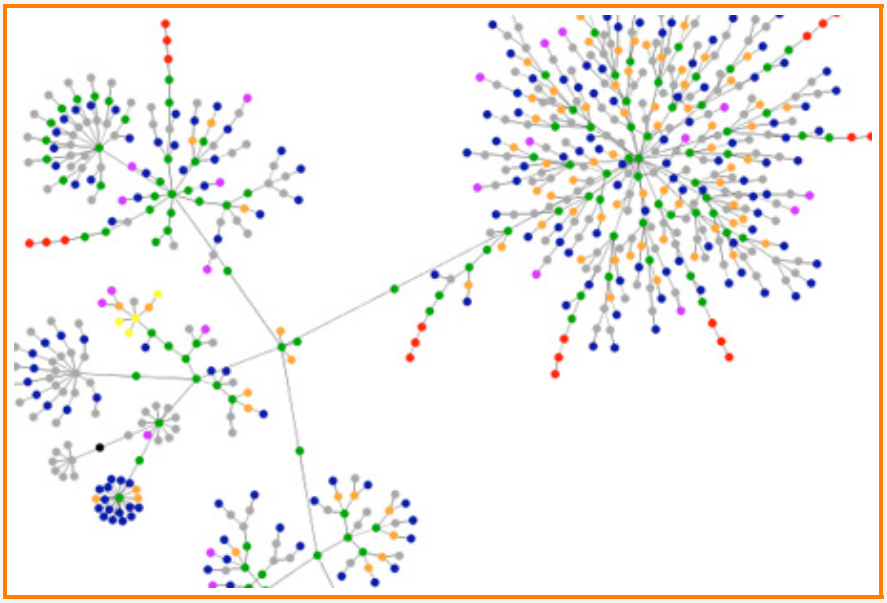

**Figure *(1.1)*** – *"Web site tree - une représentation graphique du code HTML des pages WEB" - journal "Libération", 23 janvier 2008*

— **Lisp**, **Scheme** : Langages historiquement majeurs, à la syntaxe simple, à fort pouvoir d'expression, dédiés plus spécifiquement à la programmation applicative.

— **Lisp**, 1960, John Mc Carthy : "*lists processing*"pour du calcul numérique classique mais aussi symbolique, calcul dont les données ne sont pas uniquement des nombres mais aussi des symboles et des collections de symboles.

— **Scheme**, 1970, Gerald J. Sussman et Guy L. Steele : Héritier de LISP à liaison strictement lexicale : enseignement de la programmation.

"Structure and Interpretation of Computer Programs - Abselson, Fano, Sussman - MIT Press - 1984".

— bases, préalable à l'étude d'autres formalismes, dont le formalisme objet plus large.

**Caractéristiques** de *Lisp* et des langages applicatifs :

— Toute phrase syntaxiquement correcte du langage (y compris une instruction) est une expression algébrique ayant une valeur calculable [1] [2].

— La mémoire est allouée et récupérée automatiquement

— si on le souhaite, programmation sans **Effets de bords**, lien avec la programmation dite "fonctionnelle pure" - Voir langage Haskell).

— De nombreuses abstractions simplifient la vie du programmeur. Exemple : les grands nombres, les itérateurs, etc.

```
1 (fact 30)
2 = 265252859812191058636308480000000
3 (map fact '(2 3 4 30))
4 = (2 6 24 265252859812191058636308480000000)
```

1. Même une expression réalisant un effet de bord (voir cours no 9).
2. Ceci n'est pas vrai dans les langages dits "impératifs".

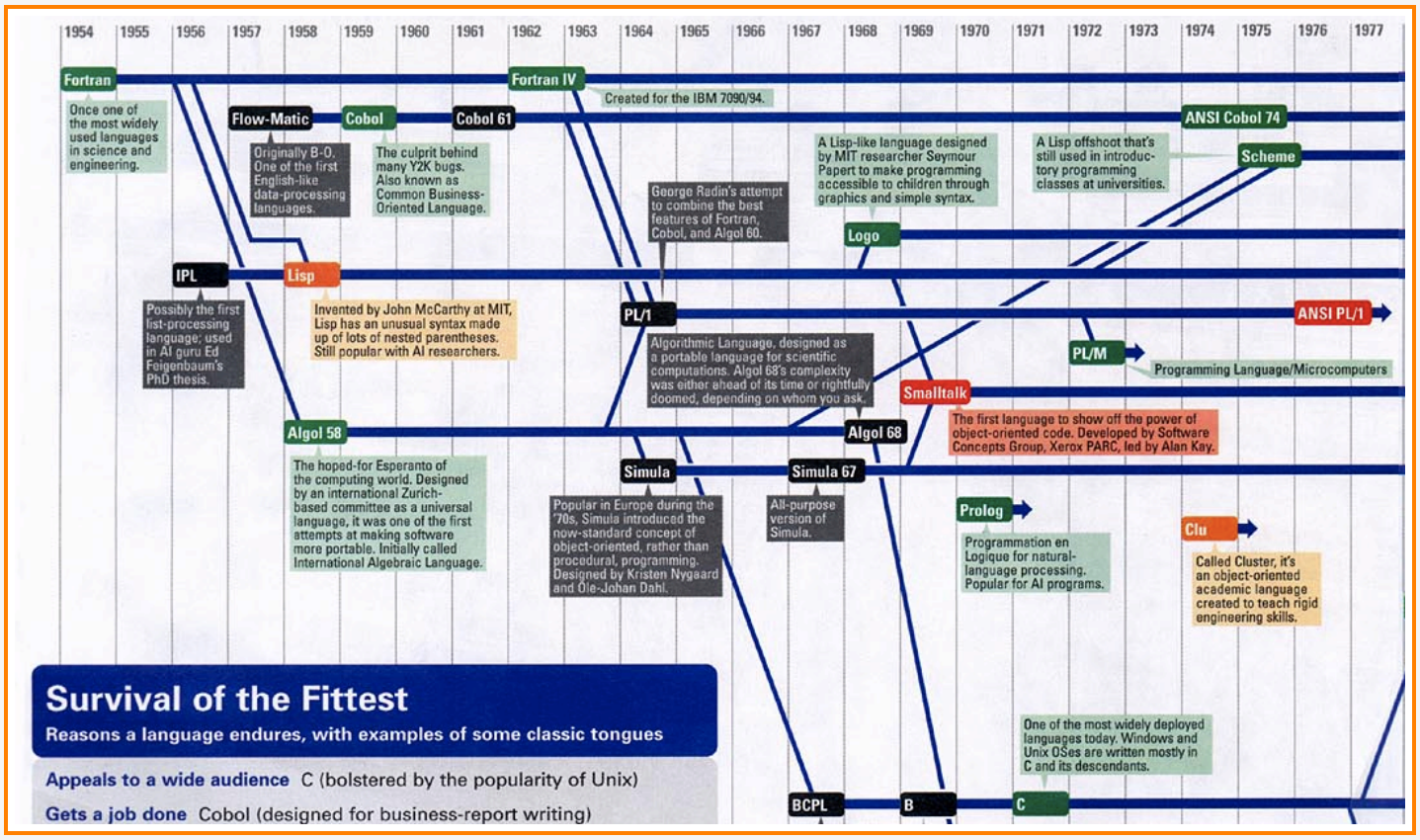


**Figure** *(1.2)* – *Les langages de programmation - Vision Historique - extrait1 - http ://www.digibarn.com/collections/posters/tongues/tongues.jpg*

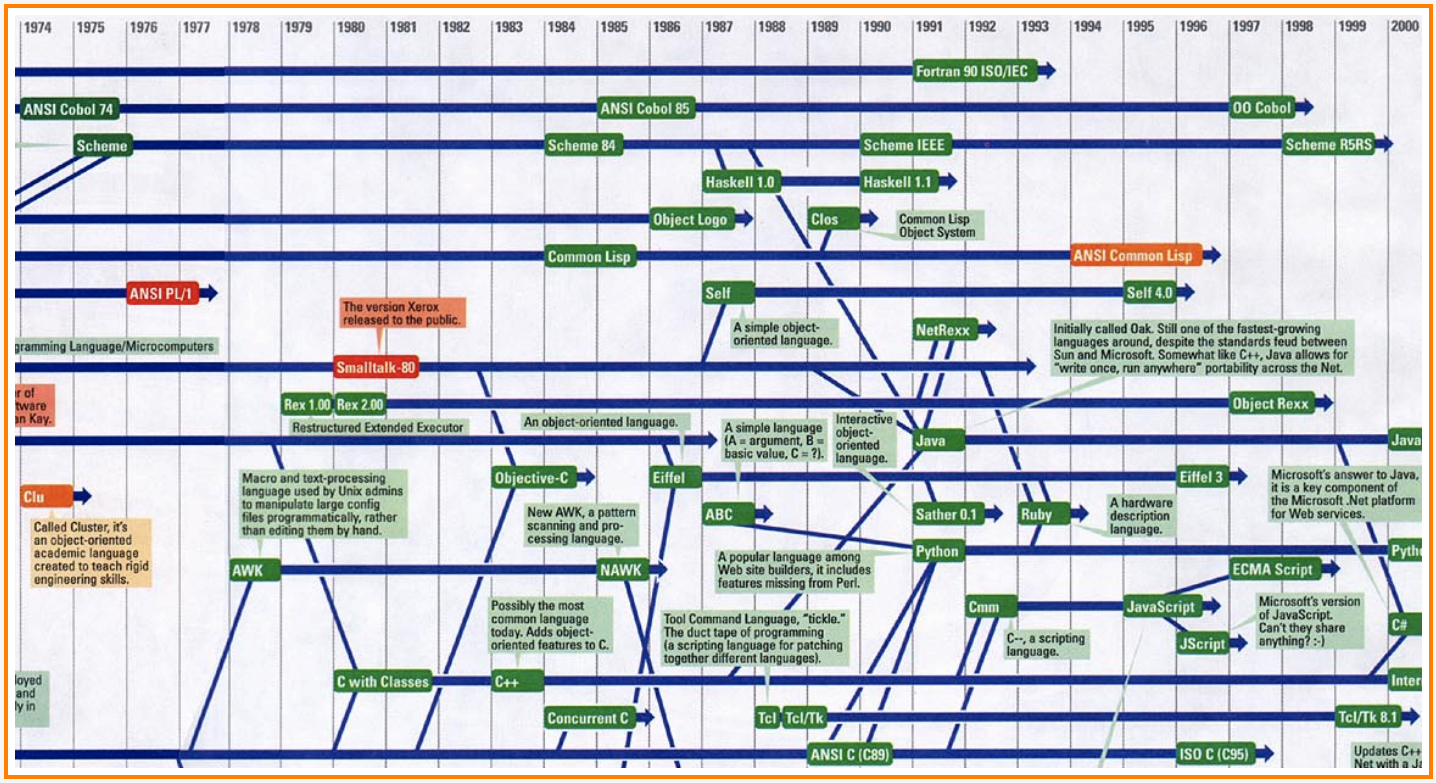


**Figure** ***(1.3)*** – *Les langages de programmation - Vision Historique - extrait 2- http ://www.digibarn.com/collections/posters/tongues/tongues.jpg*

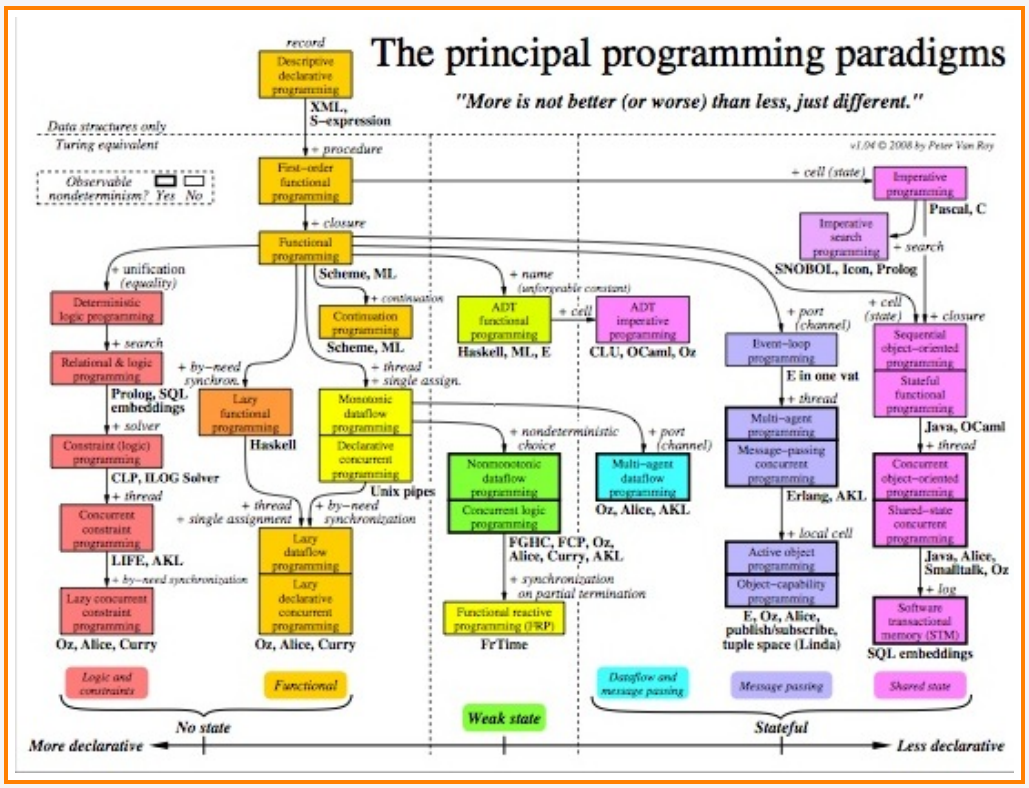


**Figure** ***(1.4)*** – *Vision Historique et Thématique - extrait de :* `http://www.ctm.info.ucl.ac.be`, *"Concepts, Techniques, and Models of Computer Programming", P. Van Roy et S. Haridi.*

Langage Universel. Tous secteurs d'applications ouverts.

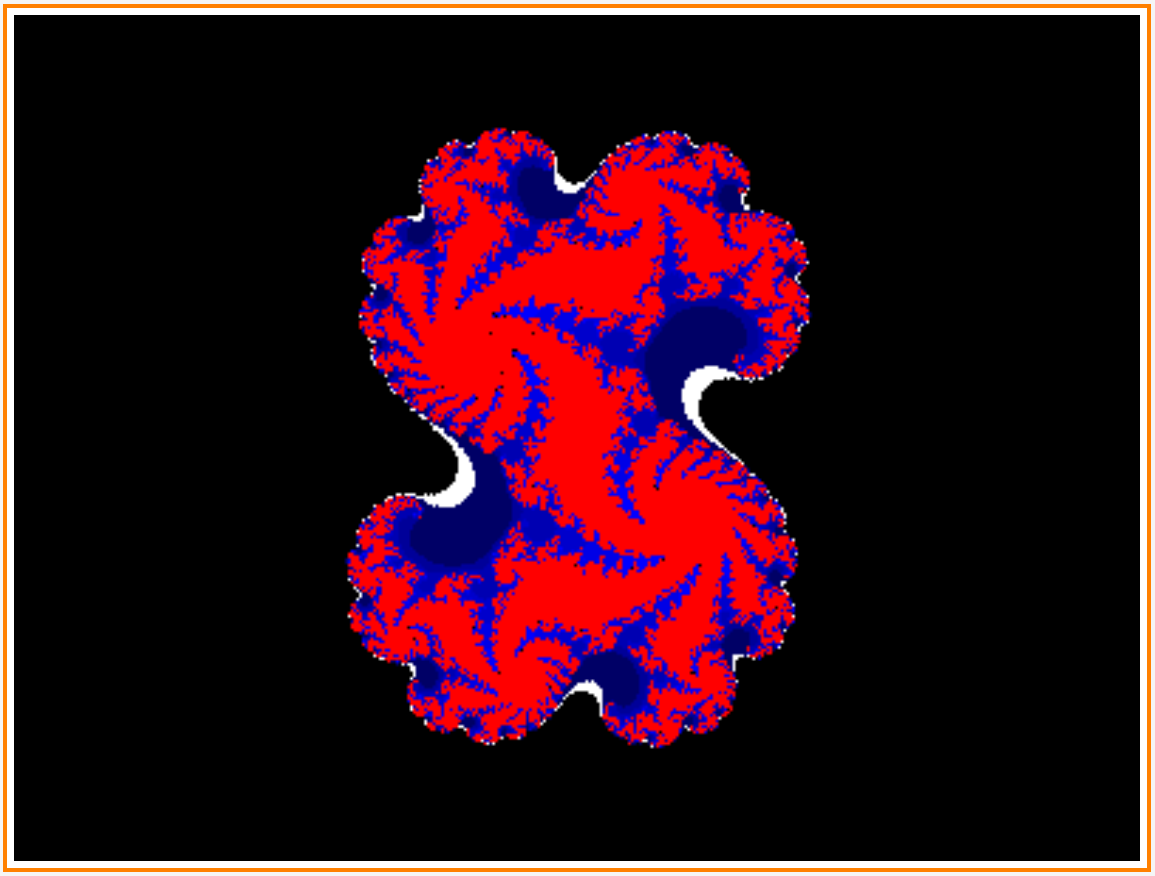

**Figure *(1.5)*** – *Image Fractale dessinée par un programme Scheme - Girault Geoffroy - TER L2 2008*

**Contenu du cours :**

- Syntaxe.
- Types prédéfinis, Typage, Base de l'interprétation des expressions.
- Identificateurs, Fonctions, Premières Structures de contrôle.
- Blocs, Environnements, Liaisons.
- Fonctions Récursives.
- Types structurés - Listes
- Listes et Symboles : Calcul Symbolique
- Fonctions récursives sur les listes.
- Arbres, Récursions arborescentes.
- Optimisation des fonctions récursives. Récursion terminales, enveloppées.
- Techniques de dérécursivation, transformations, memoization, continuations.
- Abstraction de données. Encapsulation.
- Introduction à l'interprétation des langages, eval comme exemple de fonction récursive.

## 1.2 Lectures associées

Livres ou polycopiés conseillés et également utilisés pour construire ce cours :

— "Structure and Interpretation of Computer Programs" de Abelson, Fano, Sussman.

— "Passeport pour l'algorithmique et Scheme" de R. Cori et P. Casteran.

— "Programmation récursive (en Scheme), Cours et exercices corrigés", Anne Brygoo, Titou Durand, Maryse Pelletier, Christian Queinnec, Michèle Soria ; voir page de C. Queinnec.

— "La programmation applicative, de LISP à la machine en passant par le lambda-calcul", E. Saint-James, 1993, Editions Hermès.

# Chapitre 2

# Syntaxe des expressions. Types prédéfinis. Bases de l'interprétation des expression.

## 2.1 Syntaxe de Scheme

**Syntaxe** : ensemble de règles de combinaison correcte des mots d'un langage pour former des phrases. (Syntaxe du français ou syntaxe de C ou de Scheme).

Les phrases du langage de programmation Scheme sont appelées **S-expressions** (expressions symboliques). La syntaxe pour les exprimer peut être décrire en notation BNF.

**Notation BNF (extraits)** BNF signifie "Backus Naur Form". John Backus and Peter Naur ont imaginé une notation formelle, un méta-modèle, pour décrire la syntaxe d'un language de programmation donné. Cette notation a été utilisée en premier lieu pour Algol-60.

| | |
|---|---|
| : := | se définit comme |
| \| | ou |
| <symbol> | symbole non terminal |
| { } | répétition de 0 à n fois |
| ... | suite logique |

**Description BNF de la syntaxe des S-expressions de Scheme - version simplifiée**

```
1  s_expression ::= <atom> | <expression_composee>
2  expression_composee ::== ( s_expression {s_expression} )
3  atom ::= <atomic_symbol> | <constant>
4  constant ::= <number> | <litteral_constant>
5  atomic_symbol ::== <letter>{<atom_part>}
6  atom_part ::== empty | letter{atom_part} | digit{atom_part}
7  letter ::= "a" | "b" | ... | "z"
8  digit ::= "1" | "2" | ... | "9"
9  empty = ""
```

### 2.1.1 Exemple d'expressions bien formées

```
123
true
"paul"
"coucou"
(+ 1 2)
(+ (- 2 3) 4)
(lambda (x y) (+ x y))
(sin 3.14)
```

### 2.1.2 Syntaxe préfixée et totalement parenthésée

La syntaxe de `Scheme` est dite **préfixée** et **totalement parenthésée**, ce qui se comprends intuitivement.

En conséquense, l'application d'une opération à ses opérandes etant notée de la façon suivante,

```
(operateur opérande1 ... opérandeN)
```

il n'y a jamais aucune ambiguïté liée à la priorité ou à la précédence ou à l'arité des opérations.

Par exemple, pour des opérations binaires, on ne peut pas écrire : "2 + 3 * 5".

On doit écrire "(+ 2 (* 3 5))" ou "(* (+ 2 3) 5)".

Note : Une opération est dite *unaire*, *binaire* ou *n-aire* selon son nombre d'opérandes. Le nombre d'opérande définit l'*arité* de la fonction.

## 2.2 Premières phrases

Ecrire les premières expressions du langage suppose de connaître quelques types prédéfinis, la façon dont on note leurs constantes littérales et quelques noms de fonctions. Comprendre l'interprétation des phrases, c'est à dire la façon dont sont calculées les valeurs des expressions, suppose de de connaître quelques règles d'interprétation.

### 2.2.1 Types de données

**Donnée** : entité manipulée dans un programme, définie par un type.

ex : Les entiers 1, 3 et 5 sont des données de type `integer`.

**Type de donnée** : entité définissant via une représentation et un ensemble de fonctions pour les manipuler un ensemble de données utilisables dans les programmes écrits avec un langage donné.

Par exemple en *Scheme*, le type `Integer` définit une représentation en machine des nombres entiers et un ensemble de fonctions : `+`, `-`, `integer?`, etc.

### 2.2.2 Sortes de types - exemples

**Type prédéfini** : Type livré avec un langage donné, connu de l'interpréteur et/ou du compilateur.

En scheme les principaux types prédéfinis sont : `integer, char, string, boolean, list, procedure, ...`

**Type scalaire** : Type dont tous les éléments sont des constantes.

exemple : `integer, char, boolean`.

**Type structuré** : Type dont les éléments sont une aggrégation de données

exemple : `pair, array, vector`.

### 2.2.3 Les constantes littérales des types prédéfinis

Pour chaque type prédéfini d'un langage il existe des valeurs constantes qu'il est possible d'insérer dans le texte des programmes, on les appelle des constantes littérales.

ex : 123, `#\a`, "bonjour tout le monde", `#t`, `#f`

### 2.2.4 Interprétation et valeur d'une expression : règles d'évaluation

**Interpréteur** : programme transformant un texte de programme syntaxiquement correct en actions. L'interpréteur fait faire à l'ordinateur ce que dit le texte.

Si le texte de programme est une expression algébrique, l'interpréteur calcule sa valeur ou l'**évalue**. On peut alors parler au choix d'*interpréteur* ou d'*évaluateur*.[1]

La fonction d'évaluation est usuellement nommée `eval`.

Notation. On écrira `val(e) = v` pous signifier que la valeur d'une expression `e`, calculée par l'évaluateur, est `v`.

### 2.2.5 Evaluation des constantes littérales

Soit `c` une constante littérale, `val(c) = c`.

Par exemple, `33` est un texte de programme représentant une expression valide (une constante littérale) du langage scheme, dont la valeur de type entier est *33*.

### 2.2.6 Apparté : le *Toplevel*

Un toplevel d'interprétation (ou d'évaluation) est un outil associé à un langage applicatif qui permet d'entrer une expression qui est lue (analysée syntaxiquement stockée en mémoire - fonction *read*, puis évaluée (fonction *eval)* ; la valeur résultante est alors affichée (afficheur ou *printer*).

Le processus peut être décrit ainsi :

```
tantque vrai faire
  print ( eval ( read ()))
fin tantque
```

et être écrit en scheme :

```
(while #t (print (eval (read))))
```

1. Dans le cas d'un langage non applicatif, comme Java par exemple, on parle d'interpréteur des instructions.

Exemple d'utilisation du *toplevel* pour les constantes littérales :

```
> #\a
= a
> "bonjour"
= "bonjour"
> 33
= 33
```

### 2.2.7 Application de fonction prédéfinie

Une application de fonction à des arguments (opérandes) se présente syntaxiquement sous la forme d'une expression composée : `(nom-fonction argument1 argumentN)`

Appliquer une fonction à des arguments signifie exécuter le corps de la fonction dans un environnement ou les paramètres formels sont liés à des valeurs.

`nom-fonction` doit avoir pour valeur une fonction connue du système.

Exemples : `(sqrt 9)` ou `( 2 3)+` ou `(modulo 12 8)`

Valeur de l'expression `sqrt` : `<primitive:sqrt>`

ou `<primitive:sqrt>` est ce qui est affiché par la fontion *print* pour représenter la fonction prédéfinie calculant la racine carrée d'un nombre.

Ce que confirme l'expérimentation suivante avec le *toplevel* du langage :

```
> sqrt
<primitive:sqrt>
```

Evaluation d'une application de fonction

```
val ((f a1 ... aN)) = apply (val (f), val(a1), ..., val(aN))
```

où : `apply` applique la fonction `val (f)` aux arguments `val(a1) ... val(aN)`,

i.e.

évalue les expressions constituant le corps de la fonction dans l'environnement constitué des liaisons des paramètres de la fonction à leurs valeurs

L'ordre d'évaluation des arguments n'est généralement pas défini dans la spécification du langage. Cet ordre n'a aucune importance tant qu'il n'y a pas d'**effets de bord**.

Le passage des arguments s'effectue *par valeur*. Ceci signifie que l'argument est d'abord évalué et sa valeur est ensuite passée[2].

Exemples :

2. L'alternative à l'appel par valeur est l'appel *par nom* peu usité - Voir le langage Algol-60 pour la définition de ce concept.

```
> (* 3 4)
= 12
> *
\#<primitive:*>
> (* (+ 1 2) (* 2 3))
= 18
> (* (+ (* 1 2) (* 2 3)) (+ 3 3))
= 48
```

**Evaluation en pas à pas** montrant l'ordre des évaluations et les résultats intermédiaires

```
--> (* (+ 1 2) (* 2 3))
-->    *
<--    \#<primitive:*>
-->    (+ 1 2)
-->       +
<--       \#<primitive:+>
-->       1
<--       1
-->       2
<--       2
<--    3
-->    (* 2 3)
-->       *
<--       \#<primitive:*>
-->       2
<--       2
-->       3
<--       3
<--    6
>-- 18
```

### 2.2.8 Expressions dont l'évaluation provoque des erreurs ("erreurs à l'exécution")

```
> foo
reference to undefined identifier: foo
> (f 2 3)
reference to undefined identifier: f
> (modulo 3)
modulo: expects 2 arguments, given 1: 3
> (modulo 12 #\a)
modulo: expects type <integer> as 2nd argument, given: #\a;
> (log 0)
log: undefined for 0
```

La quatrième erreur illuste ce qu'est un langage dynamiquement typé. Toutes les entités manipulées ont un type mais ces derniers sont testés durant l'exécution du programme (l'interprétation des expressions) et pas durant l'analyse syntaxique ou la génération de code (compilation).

# Chapitre 3

# Identificateurs, Fonctions, Premières Structures de contrôle.

## 3.1 Lambda-calcul

Le **lambda-calcul** est un système formel (Alonzo Church 1932), (langage de programmation théorique [1]) qui permet de modéliser les fonctions calculables, récursives ou non, et leur application à des arguments. Le vocabulaire relatif et les principes d'évaluation des expressions en *Lisp* ou *Scheme* sont hérités du lambda calcul.

Les expressions du lambda-calcul sont nommées **lambda-expressions** ou *lambda-termes*, elles sont de trois sortes : variables, applications, abstractions.

* **variable** : équivalent des variables en mathématique (x, y ...)

* **application** : notée "uv", où u et v sont des lambda-termes représente l'application d'une fonction `u` à un argument `v` ;

* **abstraction** : notée "$\lambda x.v$" ou x est une variable et v un lambda-terme, représente une fonction, donc l'abstraction d'une expression à un ensemble de valeurs possibles. Ainsi, la fonction f qui prend en paramètre le lambda-terme x et lui ajoute 1 (c'est-à-dire la fonction $f : x \to x + 2$) sera dénotée en lambda-calcul par l'expression $\lambda x.(x + 2)$.

Le lambda-calcul permet le raisonnement formel sur les calculs réalisés à base de fonction grâce aux deux opérations de transformation suivantes :

— **alpha-conversion** :
$\lambda x.xv \equiv \lambda y.yv$

— **beta-réduction** :
($\lambda$ x.xv) a $\to$ av
$(\lambda x.(x + 1))3 \to 3 + 1$

Ce sont les mêmes constructions et opérations de transformations qui sont utilisés dans les langages de programmation actuels. Nous retrouvons en Scheme les concepts de variable, fonction, application de fonctions et le calcul de la valeur des expressions par des beta-réductions successives.

## 3.2 Identificateurs

1. Jean-Louis Krivine, Lambda-Calcul, types et modèles, Masson 1991

### 3.2.1 Définition

**identificateur** nom donné à un couple "emplacementMémoire-valeur".

Selon la façon dont un identificateur est utilisé dans un programme, il dénote soit l'emplacement soit la valeur [2].

Dans l'expression `(define pi 3.14116)`, l'identificateur `pi` dénote un emplacement en mémoire.

`define` est une structure de contrôle (voir plus loin) du langage *Scheme* qui permet de ranger la valeur `v` dans un emplacement mémoire (dont l'adresse est choisie par l'interpréteur et inaccessible au programmeur) nommé `id`.

Dans l'expression `(* pi 2)`, l'identificateur `pi` dénote la valeur rangée dans l'emplacement en mémoire nommé *pi.*

### 3.2.2 Evaluation d'un indentificateur

Soit *contenu* la fonction qui donne le contenu d'un emplacement mémoire et *caseMémoire* la fonction qui donne l'emplacement associé à un identificateur alors pour tout identificateur `ident` :

$$val(ident) = contenu(caseMemoire(ident))$$

Il en résulte que :

```
1 > (* pi 2)
2 6.28...
```

Erreurs liées à l'évaluation des identificateurs : *reference to undefined identifier :*

### 3.2.3 Environnement

**Environnement** : ensemble des liaisons "identificateur-valeur" définies en un point d'un programme.

Un environnement est associé à toute exécution de fonction (les règles de masquage et d'héritage entre environnements sont étudiées plus loin).

L'environnement associé à l'application de la fonction g ci-dessous est `((x 4) (y 2))`.

```
1 >(define x 1)
2 >(define y 2)
3 >(define g (lambda (x) (+ x y)))
4 >(g 4)
5 6
```

## 3.3 Définition de fonctions

### 3.3.1 Abstraction

Une fonction définie par un programmeur est une abstraction du lambda-calcul et prend la forme syntaxique suivante : `(lambda(param1 paramN) corps)`

2. voir termes L-Value (emplacement) et R-Value (contenu).

Une fonction possède des paramètres formels en nombre quelconque [3] et un corps qui est est une S-expressions.

La représentation interne d'une abstraction est gérée par l'interpréteur du langage, elle devient un élément du type prédéfini `procédure`.

```
> (lambda (x) (+ x 1))
#<procedure:15:2>
```

On distingue ainsi le texte d'une fonction (texte de programme) et sa représentation interne (codée, en machine). [4]

### 3.3.2 Application

L'application de toute fonction est identique à l'application des fonctions prédéfinies à ceci près que les premières ne sont pas nécessairement associées à un identificateur.

```
> ((lambda (x) (+ x 1)) 3)
4
> ((lambda (x) (* x x)) (+ 2 3))
25
```

**Pas à pas.**

```
-->((lambda (x) (+ x 1)) 3)
-->(lambda (x) (+ x 1))
<-- #<procedure:15:2>
-->3
<--3
-->(+ x 1)
-->+
<-- \#<primitive:+>
-->x
<--3
-->1
<--1
<--4
<--4
```

### 3.3.3 Identificateurs

Il est possible de dénoter une valeur par un identificateur en utilisant la structure de contrôle `define`.

Ceci permet de conserver la valeur d'un calcul déjà réalisé pour éviter d'avoir à le refaire.

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

3. L'extension a un nombre quelconque de paramètres est obtenu par un procédé dit de "Curryfication" - voir ouvrages sur le lambda-calcul

4. Ecrire un interprète d'un langage, c'est aussi choisir la représentation interne des fonctions créées par l'utilisateur

```
> (define carre (lambda (x) (* x x)))
> (carre 5)
25
> (define x 10)
> (carre 5)
25
```

**parametre formel** : identificateur dénotant, uniquement dans le corps de la fonction (portée) et durant l'exécution de la fonction (durée de vie), la valeur passée en argument au moment de l'appel.

**durée de vie d'un identificateur** : intervalle de temps pendant lequel un identificateur est défini.

**portée d'un identificateur** : Zone du texte d'un programme où un identificateur est défini.

## 3.4 Premières Structures de contrôle

**structure de contrôle** : fonction dont l'interprétation nécessite des règles spécifiques.

### 3.4.1 Séquence d'instructions

Donné pour info, inutile en programmation sans effets de bord, mais nécessaire par exemple pour réaliser des affichages.

```
(begin
    i1
    i2
    ...
    in)
```

Il y a un *begin* implicite dans tout corps de fonction.

```
(lambda () (display "la valeur de (+ 3 2) est : ") (+ 3 2))
```

**évaluation d'une séquence** :

val ( (begin inst1 ... instN expr) ) = val (expr)

avec comme effet de bord : $eval(inst1), ..., eval(instN)$

### 3.4.2 Conditionnelles

```
(define (abs x)
    (if (< x 0) (- 0 x) x))

(define (abs x)
    (cond ((> x 0) x)
          ((= x 0) 0)
          (t (- 0 x))))
```

```
(define (>= x y)
  (or (= x y) (> x y)))
```

**evaluation d'une conditionnelle** :

val ( (if test av af) ) = si $(val(test) = vrai)$ alors $val(av)$ sinon $val(af)$

## 3.5 Fonctions de base sur les types prédéfinis

### 3.5.1 Nombres

tests : integer ? rational ? real ? zero ? ; odd ? even ?

comparaisons `< > <= ...`

```
> (< 3 4)
#t
> (rational? 3/4)
#t
> (+ 3+4i 1-i)
4+3i
```

### 3.5.2 Caractères

constantes littérales : `#\a #\b`

comparaison : `(char<? #\a #\b) (char-ci<? #\a #\b)`

tests : `char? char-numeric?`

transformation : `char-upcase`.

### 3.5.3 Chaînes de caractères (Strings)

constantes littérales : `"abcd"`

comparaison : `(string<? "ab" "ba")`

tests : `(string=? "ab" "ab")`

accès :

```
(substring s 0 1)
(string-ref s index)
(string->number s)
(string-length s)
```

Exemple, fonction rendant le dernier caractère d'une chaîne :

```
(define lastchar
  (lambda (s)
    (string-ref s (- (string-length s) 1))))
```

## 3.6 Blocs, évaluations en liaison lexicale

### 3.6.1 Blocs

Un bloc (concept introduit par algol) est un séquence d'instructions à l'exécution de laquelle est associée un environnement propre.

En *Scheme*, toute fonction définit implitement un bloc.

```
1 > (define (f x) (* x x))
2 > (f 2)
3 = 3
```

La structure de contrôle **let** définit également un bloc.

Syntaxe :

```
1 (let ( <liaison>* ) expression*)

3 liaison ::= (identificateur s-expression)
```

Exemple :

```
1 > (let ((x (+ 2 3)) (y (+ 3 4)))
2     (* x y))
3 = 35
```

### 3.6.2 Evaluation d'un "let"

.

Soit i une instruction de la forme `(let ((v1 expr1) ... (vn exprN)) expr)`, on a $val(i) = val(expr)$, avec *expr* évalué dans l'environnement du let augmenté des liaisons v1 à $val(expr1)$, ..., vn à $val(expnN)$.

### 3.6.3 Bloc et environnement

L'environnement associé à un bloc de code est initialisé au moment de l'exécution du bloc, (exécution du corps d'une fonction ou du corps d'un *let*.

Pour une fonction, les paramètres formels sont liés aux arguments dans l'environnement associé au corps de la fonction.

**define** : est une structure de contrôle permettant d'ajouter un identificateur ou d'en modifier un dans l'environnement courant.

**Modularité** : structuration d'un programme en espaces de nommage.

**Espace de nommage** : partie d'un programme dotée d'un environnement propre.

Un bloc est un espace de nommage. Il est possible d'utiliser sans ambigüité le même identificateur dans des blocs différents.

```
(define x 12)
(define (carre x) (* x x))
(define (cube x) (* x x x))
```

### 3.6.4 Chaînage des environnements

**Environnement fils** : Un environnement E2 peut être chainé à un environnement E1, on dit que E2 étends E1, (on dit aussi que E2 est fils de E1). E2 hérite alors de toutes les liaisons de E1 et y ajoute les siennes propres (avec possibilité de masquage).

La façon dont les environnements sont chaînés définit la portée des identificateurs.

### 3.6.5 Portée lexicale

Le chaînage est dit statique ou lexical quand un environnement d'un bloc est créé comme le fils de l'environnement du bloc englobant lexicalement (inclusion textuelle entre zones de texte).

La portée des identificateurs est alors lexicale, i.e. un identificateur est défini dans toutes les régions de programme englobée par le bloc ou se trouve l'instruction réalisant la liaison.

Scheme obéit à ce modèle (voir exemples) ainsi que la pluspart des langages.

### 3.6.6 Portée dynamique

Le chaînage entre environnements est dynamique, et la portée des identificateurs également, quand l'environnement associé à une fonction devient, à chaque exécution, le fils de celui associé à la fonction appelante.

La portée dynamique a presque disparue ...

### 3.6.7 Environnement lié à la boucle toplevel

**Environnement global de scheme** : environnement dans lequel sont définis tous les identificateurs liés aux fonctions prédéfinies. Tout autre environnement est indirectement (fermeture transitive) le fils, de cet environnement global.

**Environnement initial** : environnement affecté à la boucle toplevel i.e. dans lequel sont interprétées les expressions entrées au toplevel. Cet environnement est un fils de l'environnement global. Tout bloc définit "au toplevel" créé un environnement fils de cet environnement initial.

NB : les fonctions définies dans l'éditeur sont conceptuellement définies au toplevel.

### 3.6.8 exemples

#### Variable définie ou indéfinie

```
> (define x 22) ;; liaison de x à 22 dans l'env. courant
> (+ x 1) ;; une expression E
```

```
= 23 ;; ayant une valeur dans l'env courant
> (+ i 1) ;; une autre expression n'ayant pas de valeur
erreur : reference to undefined identifier: i
```

### Environnement fils

```
> (define x 10) ;; enrichir l'environnement courant
> (let ((y 30) (z 40)) ;; créer un environnement fils
    (+ x y z)) ;; et y faire un calcul
= 80
> x
= 10
> y
erreur : reference to undefined identifier: z
```

### Attention aux variables libres

```
(define x 1)

(define f (lambda (y) (g)))

(define g (lambda () y))

> (f 3)
*** reference to undefined identifier: y
```

```
(define x 1)
(define (f x) (g 2))
(define (g y) (+ x y)) ;; Une fonction avec une variable libre
> (f 5)
= 3 (vaudrait 7 avec portée dynamique)
```

### Utilisation préférentielle de variables locales

```
> (define (nb-racine a b c)
    (define delta (- (carre b) (* 4 a c))
    (if (= delta 0) ...)
  )
> delta
erreur : reference to undefined identifier: delta
```

```
(define (nb-racine a b c)
    (let ((delta (- (carre b) (* 4 a c))))
      (if (= delta 0)
```

```
        ...)
)
```

### Espaces de nommage

```
(define carre (lambda (x) (* x x)))

(define (sommeCarres x y) ; syntaxe simplifiée
    (+ (carre x) (carre y)))

> (sommeCarres (sin 60) (cos 60))
= 1
```

### Combinaisons plus complexes

```
(define x 1)

(define f (lambda() x))

(define g (let ((x 2)) (lambda() (+ x 2))))

(define h
  (lambda (uneFonction)
    (let ((x 3))
      (apply uneFonction ()))))

> (f)
1
> (g)
4
> (define x 5)
> (f)
5
> (g)
4
> (h g)
4
```

# Chapitre 4

# Fonctions Récursives.

## 4.1 Définitions

Une définition inductive d'une partie X d'un ensemble consiste à fournir la donnée explicite de certains éléments de X (base) et le moyen de construire de nouveaux éléments de X à partir d'éléments déjà construits.

Exemple : l'ensemble des valeurs de la fonction "factorielle" sur les entiers peut être donné par la fonction récursive suivante à partir de la donnée de base "$fact(0) = 1$" et de la règle "$fact(n) = n.fact(n-1)$".

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

Fonction récursive : fonction dont la définition inclus (au moins) un appel à elle-même.

Une autre version de factorielle, en C :

```
int fact(int n) {
  if (n == 0)
    return 1;
  else
    return n * fact(n-1); }
```

Réflexion : considérer le sens du calcul entre les versions itératives et récursives au vu de l'associativité de la multiplication.

Apparté : de l'intérêt du type abstrait "GrandNombre" (factorielle calculée avec la fonction C précédente).

```
factorielle de  5 = 120
...
factorielle de 10 = 3628800
...
factorielle de 12 = 479001600
factorielle de 13 = 1932053504
factorielle de 14 = 1278945280
factorielle de 15 = 2004310016
```

```
factorielle de 16 = 2004189184
factorielle de 17 = -288522240
```

## 4.2 Itération et récursion

Rappel : **Itérer** : répéter n fois un processus en faisant changer la valeur des variables jusqu'a obtention du résultat.

Calcul itératif de factorielle d'un nombre : $n! = \prod_{i=1}^{n} i$

Un calcul itératif se programme par une boucle (*for* ou *while* ou *repeat-until*).

Exemple de fonction itérative pour le calcul de factorielle (en C).

```c
int fact(n) { // n entier
    int i = 0;
    int result = 1;
    while (i < n){
        // result = fact(i) -- invariant de boucle
        i = i + 1;
        result = result * i;
        // result = fact(i) -- invariant de boucle
    }
    // i = n
    return(result); }
```

Inconvénient : nécessité de gérer explicitement l'évolution des variables, l'ordre des affectations et les invariants de boucle.

Autre version plus courte en C :

```c
int factorielle_iterative(int n) {
    int res = 1;
    for (; n > 1; n--) res *= n;
    return res;
}
```

## 4.3 Exemples de fonction récursives

Multiplication : $an = a + a(n-1)$

```
(define (mult a n)
  (if (= n 0)
      0
      (+ a (mult a (- n 1)))))
```

Puissance : $a^n = a.a^{n-1}$

```
(define (exp a n)
```

```
  (if (= n 0)
      1
      (* a (exp a (- n 1)))))
```

Inverser une chaîne

Idée : $inverse(n) = concatener(dernier(n), inverse(sauf Dernier(n)))$

```
(define (inverse s)
  ;;string-length est une fonction préfédinie
  (let ((l (string-length s)))
    (if (= l 0)
      s
      (string-append (inverse (substring s 1 l)) (substring s 0 1)))))
```

## 4.4 Autres exemples : Calcul des termes de suites récurrentes

Toute valeur d'une suite récurrente de la forme :

$u_0 = initial$ et pour $n > 1, u_n = \Phi(u_{n-1}, n)$

peut être calculée par une fonction (de n'importe quel langage de programmation autorisant la définition de fonctions récursives) similaire à la fonction *Scheme* suivante :

```
(define (u n)
  (if (= n 0)
      initial
      (PHI (u (- n 1)) n)))
```

Par exemple calcul de factorielle de 5 :

```
(define initial 1) (define PHI *)
(u 5) -->  120
```

### 4.4.1 Suites arithmétiques

Tout terme d'une suite arithmétique de raison $r$ de la forme :

$u_0 = initial$ et pour $n > 1, u_n = u_{n-1} + r$

peut être calculée par la fonction

```
(define (ua n r)
  (if (= n 0)
      initial
      (+ (ua (- n 1) r) r)))
```

Exemple : Multiplication de 4 par 6 : `(ua 6 4)` avec `initial = 0`

A noter que le code suivant ne fonctionne pas (voir cours No 3, liaison lexicale) :

```
(let ((initial 0)) (ua 3 4))
```

Pour **éviter de passer par une variable globale ou de passer un paramètre à chaque appel récursif**, on peut utiliser une version de la forme spéciale **let** permettant de définir des fonctions internes, temporaires et récursives.

```
(define (ua n r initial)
  (let f ((n n))
    (if (= n 0)
        initial
        (+ r (f (- n 1))))))

(ua 6 4 0)
= 24
```

### 4.4.2 Suites géométriques

Tout terme d'une suite géométrique de raison $q$ de la forme :

$u_0 = initial$ et pour $n > 1, u_n = q.u_{n-1}$ peut être calculée par la fonction ug suivante :

```
(define (ug q n initial)
  (let f ((n n))
    (if (= n 0)
        initial
        (* q (f (- n 1))))))
```

Exemple : 4 puissance 3,

```
(ug 4 3 1)
= 64
```

### 4.4.3 Calcul de la somme des termes d'une suite

#### Exemple historique

La flèche de Zénon (philosophe paradoxal) n'arrive jamais à sa cible (ou Achille ne rattrape jamais la tortue) si on décrit le mouvement comme une suite d'étapes : parcourir la moitié de la distance puis la moitié de ce qui reste, puis la moitié de ce qui reste, etc.

la flèche n'arrive jamais car : $\lim_{n\to+\infty} \sum_{i=1}^{n} 1/2^i = 1$.

Ce que l'on peut vérifier avec :

```
(define fzenon (lambda (n)
    ;; la part de distance parcourue par la flèche à l'étape n
    (if (= n 1)
        (/ 1 (expt 2 n))
        (+ (sz (- n 1)) (/ 1 (expt 2 n))))))
```

Plus lisible, utiliser une fonction annexe pour calculer "`(/1 (expt 2 n))`"

```
(define (f n) (/ 1 (expt 2 n)))

(define (fzenon1 n)
  (if (= n 1)
      (f 1)
      (+ (f n) (fzenon1 (- n 1)))))
```

Mais la fonction `f` ainsi isolée est polluante.

La solution suivante est plus élégante.

```
(define fzenon2
  (let ((f (lambda (n) (/ 1 (expt 2 n)))))
    (let fzenon ((n n))
      (if (= n 1)
          (f 1)
          (+ (f n) (fzenon (- n 1)))))))
```

Elle suppose un bonne compréhension de la notion de portée et durée de vie des identificateurs.

#### Généralisation au calcul de la somme des termes de toute suite

```
(define (sommeSuite n)
  (if (= n 0)
      (u 0)
      (+ (u n) (sommeSuite (- n 1)))))
```

A essayer avec : `(define (u n) (fact n))`

Optionnel : même fonctionnalité en n'écrivant qu'une seule fonction récursive, à condition de passer la fonction du calcul d'un terme en argument. La fonction somme devient une fonctionnelle ou fonction d'ordre supérieur.

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

On peut par exemple écrire :

```
(somme 10 (lambda (n) (/ 1 (exp 2 n))))
```

## 4.5 Interprétation d'un appel récursif

**Appel récursif** : appel réalisé alors que l'interprétation d'un appel précédent de la même fonction n'est pas achevé.

L'interprétation du code d'une fonction récursive passe par une phase d'expansion dans laquelle les appels

récursifs sont “empilés” jusqu'à arriver à un appel de la fonction pour lequel une condition d'arrêt est vérifiée, alors suivie par une phase de contraction dans laquelle les résultats partiels précédemments empilés sont utilisés.

## 4.6 Découverte d'une solution récursive à des problèmes

Disposer d'une solution récursive à un problème permet d'écrire simplement un programme résolvant (calculant quelque chose de relatif à) ce problème. La découverte de telles solutions est parfois complexe mais rentable en terme de simplicité d'expression des programmes.

Exemple : Algorithme récursif de calcul du pgcd de deux nombres non nuls :

**Require:** $b \neq 0$
  **if** $b$ *divise* $a$ **then**
    $pgcd(a, b) = b$
  **else**
    $pgcd(a, b) = pgcd(b, modulo(a, b)))$
  **end if**

Implantation :

```
1 (define (pgcd a b)
2   (if (= b 0)
3     (error "b doit être non nul")
4     (let ((m (modulo a b)))
5       (if (= m 0)
6         b
7         (pgcd b m)))))
```

## 4.7 Récursivité terminale et non terminale

**Appel récursif non terminal** : appel récursif argument d'un calcul englobant.

Exemple : l'appel récursif dans la définition de factorielle est non terminal car sa valeur est ensuite multipliée par n.

**Appel récursif terminal** appel récursif dont le résultat est celui rendu par la fonction contenant cet appel.

Exemple : appel récursif à `pgcd` dans la fonction précédente.

Propriété : l'interprétation d'un appel récursif terminal peut être réalisée sans consommer de pile.

Il est possible, en terme de mémoire, d'interpréter une fonction récursive terminale comme une fonction itérative car la gestion de la mémoire se déduit trivialement des transformations sur les paramètres.

## 4.8 Récursivité croisée

Exemple d'école “pair-impair” sur les entiers naturels

```
1 (define (pair n)
2   (or (= n 0) (impair (- n 1))))
```

```
(define (impair n)
  (and (not (= n 0)) (pair (- n 1))))
```

Intéressant de découvrir “letrec” pour définir des récursions croisées.

```
(letrec <bindings> <body>)

Syntax: <Bindings> should have the form ((<variable1> <init1>) ...),
and <body> should be a sequence of one or more expressions. It is an
error for a <variable> to appear more than once in the list of
variables being bound.

Semantics: The <variable>s are bound to fresh locations holding
undefined values, the <init>s are evaluated in the resulting
environment (in some unspecified order), each <variable> is assigned
to the result of the corresponding <init>, the <body> is evaluated in
the resulting environment, and the value(s) of the last expression in
<body> is(are) returned. Each binding of a <variable> has the entire
letrec expression as its region, making it possible to define mutually
recursive procedures.
```

```
(define (pair? n)
  (letrec ((est-pair? (lambda (n)
                (or (= n 0) (est-impair?(- n 1)))))
           (est-impair? (lambda (n)
                (and (not (= n 0)) (est-pair? (- n 1))))))
    (est-pair? n)))
```

## 4.9 Exemples : dessins de figures fractales

Voir, http://classes.yale.edu/fractals/.

Vidéo : “Fractales à la recherche de la dimension cachée”, Michel Schwarz et Bill Jersey, 2010.

Autre cours : “Les images fractales en Scheme, Une exploration des algorithmes récursifs” - Tom Mens - University de Mons-Hainaut (U.M.H.).

Programmation avec effets de bord (impressions à l’écran) :

```
;; choisir langage PLT
(require (lib "graphics.ss" "graphics"))
(open-graphics)
(define mywin (open-viewport "Triangle de Sierpinsky" 600 600))
(define uneCouleur (make-rgb .5 0 .5))
```

Exememple des triangles de *Sierpinski* :

```
(define (s-carré n x y cote)
  ;; n nombre d'itérations
  ;; x, y : coordonées du coin supérieur gauche de la figure
  ;; cote : longueur du carré
  (if (= 0 n)
```

```
6       ((draw-solid-rectangle mywin) (make-posn x y) cote cote uneCouleur)
7       (let ((moitié (/ cote 2)) (quart (/ cote 4)))
8         (s-carré (- n 1) (+ x quart) y moitié)
9         (s-carré (- n 1) x (+ y moitié) moitié)
10        (s-carré (- n 1) (+ x moitié) (+ y moitié) moitié))))
```

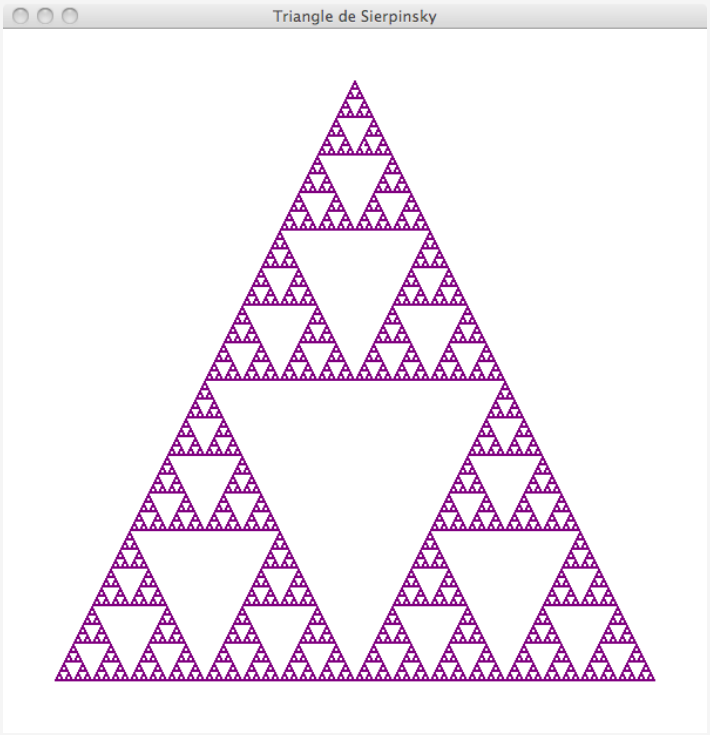


**Figure *(4.1)*** – *(s-carré 8 44 44 600) : sont tracés* $3^{n_{initial}}$ *carrés de coté "*$cote_{initial}/2^{n_{initial}}$*", soit pour* $n_{initial} = 8$ *et* $cote_{initial} = 512$, $3^8 = 6561$ *carrés, de côté* $512/2^8 = 2$.

# Chapitre 5

# Listes, Symboles, calcul Symbolique.

## 5.1 Types structurés

**Type structuré** : type dont les éléments sont des aggrégats de données.

### 5.1.1 Types structurés primitifs

- **Paire** : aggrégat de deux données accessibles par leur position (première ou seconde).
- **Tableau** : aggrégat d'un nombre quelconque prédéfini de données généralement homogènes (tableau d'entiers) accessibles via un index.
- **Enregistrement (record)** : aggrégat de données généralement hétérogènes accessibles via un nom. Exemple, une personne est un aggrégat d'un nom, d'un prénom (chaîne), d'un age (int), etc.
- **Sac** : collection quelconque non ordonnée
- **Ensemble** : collection quelconque sans ordre ni répétition.
- **Séquence - Liste** : collection ordonnée soit par l'ordre d'insertion par une relation entre les éléments.

### 5.1.2 Les paires (doublets) en Scheme

**Pair (paire)** ou **Cons** est le type structuré de base.

Une paire, également dit "cons" (prononcer "conce") ou doublet est un aggrégat de deux données.

Exemple d'utilisation : représentation d'un nombre rationnel, aggrégation d'un numérateur d'un dénominateur.

Notation : "(el1 . el2)".

**Fonctions de construction** :

1. `cons`

```
(cons 1 2)
= (1 . 2)

(cons (+ 1 2) (+ 2 3))
= (3 . 5)
```

**Fonctions d'accès** :

1. `car` accès au premier élément.

```
(car (cons 1 2))
= 1
```

2. accès au second élément : `cdr`

```
(cdr (cons 1 2))
= 2
```

## 5.2 Listes

**Liste :** collection de données ordonnée par l'ordre de construction (ordre d'insertion des éléments).

Une liste est une **structure de donnée récursive** : soit la liste vide soit un doublet dont le second élément est une liste.

Les fonctions de construction et de manipulation de base sont les mêmes que celles des doublets.

```
(cons 1 (cons 2 (cons 3 ())))
= (1 . (2 . (3 . ())))
```

Notation simplifiée : `(1 2 3)`.

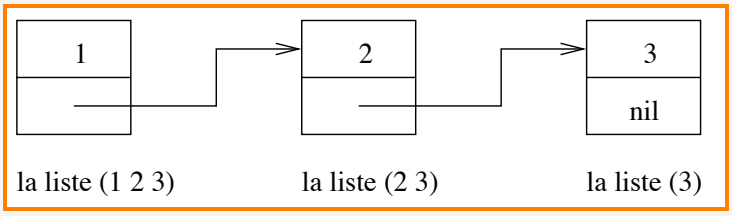


**Figure *(5.1)*** – *Vue de la représentation interne d'une liste*

— `car` rend le premier élément du premier doublet constituant la liste donc le premier élément de la liste.

— `cdr` rend le second élément du premier doublet c'est à dire la liste privée de son premier élément.

— `list` est une fonction n-aire réalisant n "cons" successifs.

## 5.3 Symboles et Calcul

Le calcul symbolique, manipule des données non uniquement numérique ; il utilise des listes, des chaînes de caractères et des symboles.

### 5.3.1 Symboles

Définition : chaîne de caractères alphanumériques unique en mémoire utilisable :

- stricto sensu

- comme identificateur (de fonction ou de variable).

Exemples :

— un symbole évoquant une couleur : `rouge`

— une liste de symboles évoquant une jolie maison rouge : `(une jolie maison rouge)`

— une liste de plein de choses différentes : `(jean dupond 20 marié jeanne 2 jeannette jeannot (2 bis rue des feuilles))`

— une liste d'associations : `((dupont 10) (durand 16) (martin 12))`

— une autre liste d'associations : `((Hugo (ecrivain francais 19ieme)) (Moliere (...)) ...)`

— une liste de symboles et de nombres représentant la définition d'une fonction scheme : `(lambda (n) (if (= n 0) 1 ...)`

— un symbole utilisé comme identificateur dénotant une fonction : `(define fact (lambda (n) (if ...)`

```
1 > fact
2 #<procedure:fact>
```

## 5.4 Calcul symbolique, Guillemets, Citations

En français mettre un texte ou un mot entre guillemets permet de faire une citation. Les guillemets empêchent le texte qu'ils délimitent d'être interprété de façon standard.

```
1 dis moi quel est ton age.
2 dis moi : "quel est ton age"!.
```

En programmation, les guillements sont utiles dés lors qu'il y a ambiguïté possible dans l'interprétation d'une expression,

En *Scheme*, un appel de fonction se présente sous forme d'une liste mais toutes les listes présentes dans le texte d'un programme ne sont pas des appels de fonctions.

Pour signifier à l'interpréteur qu'une expression parenthésée n'est pas un appel de fonction, il faut la mettre entre guillemets.

De même en *Scheme*, un identificateur est un symbole mais tous les symboles présents dans le texte d'un programme scheme ne sont pas des identificateurs. Pour indiquer à l'interpréteur qu'une symbole n'est pas un identificateur, il faut le mettre entre guillemets.

La structure de controle scheme permettant de mettre une expression entre guillemets est "`quote`".

Soit e une expresion de la forme "(quote exp)", on a $val(e) = exp$.

```
1 > (+ 2 3)
2 = 5
3 > (quote (+ 2 3))
4 = (+ 2 3)
5 > '(+ 2 3) ;; racourci syntaxique
6 = (+ 2 3)
```

```
> (list (quote un) (quote joli) (quote chien))
= (un joli chien)
> (define un 1)
> (define deux 2)
> (list (quote un) deux)
= (un 2)
```

### 5.4.1 Manipulation de listes

— tests : null ?, list ?, cons ?, pair ? member ?

— renversement, concaténation.

```
(reverse '(1 2 3))
= (3 2 1)
(append '(1 2) '(3 4))
= (1 2 3 4)
```

- *reverse* d'une liste de n éléments consomme n nouveaux doublets.

- *append* de *l1* de taille $n$ et de *l2* de taille $m$ consomme n nouveaux doublets.

### 5.4.2 Egalité physique ou logique

Egalité physique : `eq?`

Egalité logique : `equal?`

A tester :

```
(equal? (list 'a 'b) (list 'a 'b))

(eq? (list 'a 'b) (list 'a 'b))
```

Extension au problème de l'appartenance à une liste : `member?` versus `memq?`

### 5.4.3 Fonctionnelles - Itérateurs

- Appliquer une fonction à tous les éléments d'une liste.

A tester :

```
(map number? '(1 "abc" (3 4) 5))

(map fact '(0 1 2 3 4 5))
```

- Il n'est pas indispensable de nommer une fonction pour la passer en argument à une fonctionnelle.

A tester :

```
(map (lambda (x) x) '(a b c))
```

```
(eq? '(a b c) (map (lambda (x) x) '(a b c)))
```

### 5.4.4 Listes généralisées

Liste généralisée ou liste non plate ou arbre : liste dont chaque élement peut être une liste.

Exemple : liste d'associations.

```
(define unDico '((hugo ecrivain) (pasteur médecin) (knuth
algorithmicien)))

(assq 'pasteur unDico)
= médecin
```

Arbres, voir chapitres suivants.

# Chapitre 6

# Récursivité suite, Fonctions récursives sur les listes et arbres, Récursions arborescentes.

## 6.1 Fonctions récursives sur les listes

Exemple : Recherche d'un élément

```
(define (member? x l)
  (cond ((null? l) nil) ; ou #f
        ((equal? x (car l)) l) ; ou #t
        (#t (member? x (cdr l)))))
```

Exercice : Recherche du nième cdr d'une liste.

### 6.1.1 Un programme utilisant des listes

Réalisation d'un dictionnaire et d'une fonction de recherche d'une définition.

```
(define dico '((Hugo Lavoisier Einstein Poincaré Knuth)
               (écrivain chimiste physicien mathématicien informaticien)))

(define (donneDef nom)
  (letrec ((chercheDef (lambda (noms definitions)
     (cond ((null? noms) 'non-défini)
           ((eq? (car noms) nom) (car definitions))
           (#t (chercheDef (cdr noms) (cdr definitions)))))))

    (chercheDef (car dico) (cadr dico))))

(donneDef 'joliot)
= mathématicien
```

## 6.2 Récursivité enveloppée sur les listes : schéma 1

```
(define (recListe l)
  (if (null? l)
      (traitementValeurArrêt)
      (enveloppe (traitement (car l))
                 (recListe (cdr l)))))
```

### Exemple 1

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

- traitementValeurArret : rendre 0

- traitement du *car* : ne rien faire

- enveloppe : `(lambda (x) (+ x 1))`

```
(longueur '(1 3 4))
-->(+ 1 (longueur '(3 4)))
  -->(+ 1 (+ 1 (longueur '(3))))
    -->(+ 1 (+ 1 (+ 1 (longueur ()))))
      -->(+ 1 (+ 1 (+ 1 0)))
...
3
```

### Exemple 2

```
(define (append l1 l2)
  (if (null l1)
      l2
      (cons (car l1)
            (append (cdr l1) l2))))
```

- traitement traitementValeurArret : rendre l2

- traitement du `car` : ne rien faire

- enveloppe : `cons`

Consommation mémoire : taille de l1 doublets.

### Exemple 3

```
(define (add1 l)
```

```
    (if (null l)
        ()
        (cons (+ (car l) 1) (add1 (cdr l)))))
```

```
(add1 '(1 2 3))
= (2 3 4)
```

- traitementValeurArret : rendre ()

- traitement (car l) : +1

- enveloppe : cons

### Exemple 4 : tri par insertion

La fonction `insertion`

```
(define (tri-insertion l)

  (define (insertion x l2)
      (cond ((null? l2) (list x))
            ((< x (car l2)) (cons x l2))
            (#t (cons (car l2) (insertion x (cdr l2))))))

  (if (null? l)
      ()
      (insertion (car l) (tri-insertion (cdr l)))))
```

Consommation mémoire : chaque insertion consomme en moyenne $taille(l2)/2$ doublets ; il y a $taille(l)$ insertions. La consommation mémoire est donc en $O(l^2/2)$

Consommation pile : pour `tri-insertion`, $taille(l)$. Pour `insertion`, en moyenne $taille(l2)/2$

## 6.3 Récursivité enveloppée - schéma 2

```
(define (recListe2 l)
  (if (null? l)
      (traitement ())
      (enveloppe (recListe2 (cdr l))
                 (traitement (car l)))))
```

Exemple.

```
(define (reverse l)
    (if (or (null l) (null (cdr l)))
        l
        (append (reverse (cdr l)) (list (car l)))))
```

Consommation mémoire : 'taille de l' doublets.

## 6.4 Récursivité arborescente

Fonction récursive arborescente : fonctions récursives contenant plusieurs appels récursifs, éventuellement enveloppés, ou dont l'enveloppe est elle-même un appel récursif

Fonction dont l'interprétation nécessite un arbre de mémorisation.

### 6.4.1 L'exemple des suites récurrentes à deux termes

Exemple du calcul des nombres de fibonacci.

Problème initial"Si l'on possède initialement un couple de lapins, combien de couples obtient-on en n mois si chaque couple engendre tous les mois un nouveau couple à compter de son deuxième mois d'existence".

Ce nombre est défini par la suite récurrente linéaire de degré 2 suivante :

$F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$

Les nombres de Fibonacci sont célèbres en arithmétique et en géométrie pour leur relation avec le nombre d'or, solution de l'équation $x^2 = x + 1$. La suite des quotients de deux nombres de Fibonacci successifs a pour limite le nombre d'or.

Les valeurs de cette suite peuvent être calculées par la fonction scheme suivante.

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (#t (+ (fib (- n 1)) (fib (- n 2))))))
```

**Complexité** :

L'interprétation de cette fonction génére un arbre de calcul (récursivité arborescente). L'**arbre de calcul** de `(fib n)` possède $fib(n+1)$ feuilles. Exemple, fib(4) : 5 feuilles.

Ceci signifie que l'on calcule $fib(n+1)$ fois fib(0) ou fib(1), et que l'on effectue $fib(n+1) - 1$ additions.

Par exemple, pour calculer $fib(30) = 842040$, cette fonction effectue $fib(31) - 1$ soit 1346268 additions.

### 6.4.2 Exemple des listes généralisées

Recherche d'un élément dans une liste généralisée (une liste contenant des listes)

```
(define (genmember? x lgen)
  (cond ((null? lgen) #f)
        ((equal x (car lgen)) #t)
        ((list? (car lgen))
         (or (genmember? x (car lgen))
             (genmember? x (cdr lgen))))
        (#t (genmember? x (cdr lgen)))))
```

Exercice : écrire une fonction qui "applatit" une liste généralisée.

```
(flat '((1 2) 3 (4 (5 6) 7)))
= (1 2 3 4 5 6 7)
```

### 6.4.3 Exemple sur un problème combinatoire

#### Problème

L'exemple suivant, tiré de "structure and interpretation of computer programs" montre la réduction d'un problème par récursivité.

Problème : soit à calculer le nombre `N` de façon qu'il y a de rendre une somme `S` en utilisant `n` types de pièces.

Il existe une solution basée sur une réduction du problème jusqu'à des conditions d'arrêt qui correspondent au cas ou la somme est nulle ou au cas où il n'y plus de types de pièces à notre disposition.

#### Réduction du problème

Soient

- $V$ l'ensemble ordonné des valeurs des différentes pièces,
- $n$ le nombre de types de pièces (égal au cardinal de $V$),
- $n_i$ le ième type de pièce
- $v_i$ la valeur du ième type de pièce.

Par exemple, avec l'euro $V = \{1, 2, 5, 10, 20, 50, 100, 200\}$ et $n = 8$ (il y a 8 types de pièces de valeurs respectives 1, 2, 5, 10, 20, 50, 100 et 200 cts)

On peut réduire le problème ainsi :

$$NFRendre(S, n) = NFRENDRE(S - v_1, n) + NFRendre(S, n - 1)$$

#### Valeurs initiales

si $S = 0$, 1
si $S < 0$, 0 (on ne peut pas rendre une somme négative - ce cas ne peut se produire que pour des monnaies qui ne possèdent pas la pièce de 1 centime)
si $S > 0, n = 0$, 0 (aucune solution pour rendre une somme non nulle sans pièce)

#### Exemple

```
Rendre 5 centimes avec (1 2 5) =
  Rendre 4 centimes avec (1 2 5)
  + Rendre 5 centimes avec (2 5)
soit en développant le dernier, =
  Rendre 4 centimes avec (1 2 5) ...
  + rendre 3 centimes avec (2 5)
  + Rendre 5 centimes avec (5)
soit en développant le dernier, =
  Rendre 4 centimes avec (1 2 5) ...
  + rendre 3 centimes avec (2 5) ...
  + rendre 0 centimes avec (5)
  + Rendre 5 centimes avec ()
soit en appliquant les tests d'arrêt, =
  Rendre 4 centimes avec (1 2 5) ...
  + rendre 3 centimes avec (2 5) ...
  + 1
  + 0
```

= 4 ((1 1 1 1 1) (1 1 1 2) (1 2 2) (5))

## Algorithme

Soient :

- $S$ la somme à rendre

- $V$ l'ensemble ordonné des valeurs des différentes pièces

- $n$ le nombre de types de pièces (égal au cardinal de $V$)

- $n_i$ le ième type de pièce et $v_i$ la valeur de ce ième type de pièce.

NFRendre(S, n, V) =
si $S = 0$ alors 1
sinon si $S < 0$ alors 0 (on ne peut pas rendre une somme négative)
sinon si $n = 0$ alors 0 (on ne peut pas rendre une somme sans pièce)
sinon
$NFRendre(S - v_1, n, V) + NFRendre(S, n - 1, V\ v_1)$

## Implantation

```
(define (NFRendre somme nbSortesPieces valeursPieces)
  (letrec ((rendre (lambda (somme nbSPieces)
                (cond ((= somme 0) 1)
                      ((or (< somme 0) (= nbPieces 0)) 0)
                      (#t (+ (rendre somme (- nbSPieces 1))
                             (rendre (- somme (valeurPiece nbSPieces)) nbSPieces)))))))
    (rendre somme nbSortesPieces))

(define (valeurPiecesEuro piece)
  (cond ((= piece 1) 1) ;; la piece no 1 vaut 1 centime, etc
        ((= piece 2) 2)
        ((= piece 3) 5)
        ((= piece 4) 10)
        ((= piece 5) 20)
        ((= piece 6 ) 50)
        ((= piece 7 ) 100)
        ((= piece 8 ) 200)
        ))
```

```
(define (valeurPiecesDollar piece)
  (cond ((= piece 1) 1) ;; la piece no 1 vaut 1 centime, etc
        ((= piece 2) 5)
        ((= piece 3) 10)
        ((= piece 4) 25)
        ((= piece 5) 50)
        ))

(define (rendreEuro somme)
  (NFRendre somme 8 valeurPiecesEuro))

(define (rendreDollar somme)
```

```
13  (NFRendre somme 5 valeurPiecesDollar))
```

Exercice : Ecrivez une variante du programme qui rend la liste des solutions.

La complexité est du même ordre que celle de la fonction fibonacci mais pour ce problème il est beaucoup plus difficile d'écrire un algorithme itératif.

# Chapitre 7

# Optimisation des fonctions récursives.

## 7.1 Rappels sur itérations

Processus de calcul itératif : processus basé sur une suite de transformation de l'état de l'ordinateur spécifié par au moins une boucle et des affectations.

### Affectation

Définition : instruction permettant de mofifier la valeur stockée à l'emplacement mémoire associé au nom de la variable.

```
x := 12; ;; en Algol, Simula, Pascal (syntaxe classique)
x = 12; ;; en C, Java, C++
(set! x 12) ;; en scheme
(set! x (+ x 1))
```

### Une structure de contrôle pour écrire les boucles en Scheme

```
(do (initialisation des variables)
    (condition-d'arret expression-si-condition-d'arret-vérifiée)
  (instruction 1)
  ...
  (instruction n))
```

## 7.2 Equivalence itération - récursions terminales

Du point de vue de la quantité d'espace pile nécessaire pour leur interprétation, les deux fonctions suivantes sont équivalentes.

```
;; fonction explicitement itérative
(define (ipgcd a b)
   (do ((m (modulo a b)))
       ((= m 0) b)
       (set! a b)
       (set! b m)
       (set! m (modulo a b))))
```

```scheme
;; fonction récursive terminale
(define (rpgcd a b)
  (let ((m (modulo a b)))
    (if (= m 0)
        b
        (rpgcd b m))))
```

## Transformations

Note : Il n'est pas utile de vouloir à tout prix dérécursiver. La taille des mémoires et l'efficacité des processeurs rendent de nombreuses fonctions récursives opérantes.

Cependant certaines transformations sont simples et il n'est pas couteux de les mettre en oeuvre. Les récursions à complexité exponentielles doivent être simplifiées quand c'est possible.

Pour dérécursiver, il y a deux grandes solutions : soit trouver une autre façon de poser le problème soit, dans le cas général, passer en argument le calcul d'enveloppe.

## 7.3 Transformation des récursions enveloppées simples

Exemple de factorielle.

### Une version itérative (en C)

```c
int fact(n){
  int cpt = 0;
  int acc = 1;
  while (cpt < n){
    // acc = fact(cpt) -- invariant de boucle
    cpt = cpt + 1;
    acc = acc * cpt;
 n // acc = fact(cpt) -- invariant de boucle
  }
  // en sortie de boucle, cpt = n, acc = fact(n)
  return(acc)
}
```

### Une version itérative (en scheme)

```scheme
(define (ifact n)
  (do ((cpt 0) (acc 1))
    ;; test d'arret et valeur rendue si vrai
    ((= cpt n) acc)
    ;; acc = fact(cpt) -- invariant de boucle
    (set! cpt (+ cpt 1))
    (set! acc (* acc cpt))
    ;; acc = fact(cpt) -- invariant
   ))
```

### 7.3.1 Calcul d'enveloppe via les arguments

Solution générique à la dérécursivation : passer, lorsque c'est possible, le calcul d'enveloppe (ce qui restera à calculer une fois l'appel récursif terminé) en argument de l'appel récursif.

Deux paramètres vont prendre au fil des appels récursifs les valeurs successives de `cpt` et `acc` de la version itérative.

```
(define (ifact n cpt acc)
   ;; acc = fact(cpt)
   (if (= cpt n)
       acc
       (ifact n (+ cpt 1) (* (+ cpt 1) acc))))
```

Il est cécessaire de réaliser le premier appel de la fonction avec les valeurs initiales correctes de `cpt` et `acc` : `(ifact 7 0 1)`

Version plus élégante, qui évite le passage du paramètre `n` à chaque appel récursif et évite l'appel initial complexe.

```
(define (rtfact n)
  (let boucle ((cpt 0) (acc 1))
    ;; acc = fact(cpt)
    (if (= cpt n)
        acc
        (boucle (+ cpt 1) (* (+ cpt 1) acc)))))
```

La multiplication étant associative, on peut effectuer les produits de n à 1. Ce qui donne une version encore plus simple.

```
(define (rtfact2 n)
  (let boucle ((cpt n) (acc 1))
    ;; acc = fact(n-cpt)
    (if (= cpt 0)
        acc
        (boucle (- cpt 1) (* cpt acc)))))
```

Trouver ce que calcule la fonction interne boucle revient à trouver l'invariant de boucle en programmation impérative. A chaque "tour de boucle", on vérifiera pour cette version que $acc = fact(n - cpt)$.

### 7.3.2 Dérécursivation par "passage de continuation"

Lorsqu'il n'est pas possible de réaliser un calcul partiel (menant au résultat final) à chaque étape de récursion, on peut utiliser une technique plus brutale : le passage de la continuation en argument.

Le CPS (Continuation passing style) est une généralisation de la technique de résorbtion des enveloppes. Avec le CPS, on passe en argument à l'appel récursif la continuation du calcul.

**Continuation** : pour un calcul donné, fonction qui utilisera le résultat.

Une mise en pratique avec "factorielle" :

```
(define (kfact n)
  (let boucle ((n n) (k (lambda(x) x)))
    (if (= n 0)
      (k 1)
      (boucle (- n 1) (lambda(x) (k (* n x)))))))
```

Pour comprendre comment le CPS fonctionne, il est est intéressant de visualiser la continuation sous forme lisible via une liste au lieu de calculer sa valeur.

```
(define (kfactExplication n)
```

```
  (let boucle ((n n) (k (list 'lambda (list 'x) 'x)))
    (if (= n 0)
      (list k 1)
      (boucle (- n 1) (list 'lambda (list 'x) (list k (list '* n 'x)))))))
```

```
(kfactExplication 0)
= ((lambda (x) x) 1)
(kfactExplication 1)
= ((lambda (x) ((lambda (x) x) (* 1 x))) 1)
(kfactExplication 2)
= ((lambda (x) ((lambda (x) ((lambda (x) x) (* 2 x))) (* 1 x))) 1)

(eval (kfactExplication 3))
= 6
```

### 7.3.3 Quelques mesures comparatives

Mesures de la différence de temps d'exécution entre versions (utilisation de la fonction `time`).

Utilisons pour cela une fonction naïve de multiplication `mult` qui ne construit ni de très grand nombres, ni de liste, pour laquelle on mesurera donc essentiellement le coût des appels récursifs.

```
(define (mult-it x n)
  ;; version explicitement itérative
  (do ((n n) (acc 0))
    ((= n 0) acc)
    (set! n (- n 1))
    (set! acc (+ acc x))))

(define (mult-rt x n)
  ;; version récursive terminale
  (let boucle ((n n) (acc 0))
    (if (= n 0)
        acc
        (boucle (- n 1) (+ acc x)))))

(define (mult-k x n)
  ;; version rendue récursive terminale par CPS
  (let boucle ((n n) (k (lambda(x) x)))
    (if (= n 0)
        (k 0)
        (boucle (- n 1) (lambda (y) (k (+ x y)))))))
```

```
n = 4000000
(time (mult-r 0 n))
cpu time: 31054 real time: 31216 gc time: 23742

(time (mult-k 0 n))
cpu time: 1411 real time: 1437 gc time: 541

(time (mult-rt 0 n))
cpu time: 756 real time: 773 gc time: 0

(time (mult-it 0 n))
cpu time: 1123 real time: 1157 gc time: 58
```

```
n = 50000000
(time (mult-k 0 n))
cpu time: 65938 real time: 66278 gc time: 55422

(time (mult-rt 0 n))
cpu time: 9187 real time: 9276 gc time: 0

(time (mult-it 0 n))
cpu time: 14036 real time: 14182 gc time: 1190
```

## 7.4 Recursion vers iteration, le cas des listes

### 7.4.1 Longueur d'une liste

Mêmes principes que précédemment, seuls les opérateurs changent.

Longueur d'une liste.

```
(define (longueur l)
  (let boucle ((l2 l) (acc 0))
    ;; acc = taille(l) - taille(l2)
    (if (null? l2)
        acc
        (boucle (cdr l2) (+ 1 acc)))))
```

### 7.4.2 Autres exemples

#### Somme des éléments d'une liste

Version explicitement itérative :

```
(define (do-somme-liste l)
    (do ((l1 l) (acc 0))
        ((null l1) acc)
        (set! accc (+ acc (car l)))
        (set! l1 (cdr l1))))
```

Version récursive terminale :

```
(define (i-somme-liste l)
  (let boucle ((l l) (acc 0))
    (if (null? l)
        acc
        (boucle (cdr l) (+ (car l) acc)))))
```

#### Renversement d'une liste

Version explicitement itérative.

```
(define (do-reverse l)
  (do ((current l) (result ())))
```

```
    ((null? l) result)
    (set! result (cons (car l) result))
    (set! l (cdr l))
    ;; invariant : result == reverse (initial(l) − current(l))
    ))
```

Version récursive terminale. L'accumulateur est une liste puisque le résultat doit en être une. A surveiller le sens dans lequel la liste se construit par rapport à la version récursive non terminale.

```
(define (ireverse l)
  (let boucle ((l l) (acc ()))
    (if (null? l)
        acc
        (boucle (cdr l) (cons (car l) acc)))))
```

## 7.5 Autres améliorations de fonctions récursives

Etude mathématique ou informatique du problème.

Exemple avec la fonction puissance : amélioration des formules de récurrence conjuguée à une dérécursivation.

— Version récursive standard. Cette version réalise n multiplications par `x` et consomme n blocs de pile.

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

— Une version récursive terminale conforme au schéma de dérécursivation précédent nécessite toujours n appels de fonction, n multiplications par x mais ne consomme plus de pile.

```
(define (puissance-v2 x n)
  (let boucle ((cpt n) (acc 1))
    ;; puis(x,n−cpt) = acc
    (if (= cpt 0)
        acc
        (boucle (− cpt 1) (* acc x)))))
```

— Une version[1] récursive "dichotomique" :

si $n$ est pair, $\exists m = n/2$ et $x^n = x^{2m} = (x^2)^m$
si n est impair, $x^n = x^{2m+1} = x.x^{2m} = x.(x^2)^m$

```
(define (puissance-v3 x n)
  (cond ((= n 0) 1)
        ((even? n) (puissance-v3 (* x x) (quotient n 2)))
        ((odd? n) (* x (puissance-v3 (* x x) (quotient n 2))))))
```

Fait passer de $O(n)$ à $O(log_2(n))$ appels récursifs et multiplications.

— Couplage des deux améliorations
Une version itérative de la fonction puissance dichotomique suppose à nouveau le passage par un accumulateur

1. Attention, utiliser "quotient" plutôt que "/" car quotient rend un entier compatible avec les fonctions "even" et "odd".

```
(define (puissance-v4 x n)
  (let boucle ((x x) (n n) (acc 1))
    (cond ((= n 0) acc)
          ((even? n) (boucle (* x x) (quotient n 2) acc))
          ((odd? n) (boucle (* x x) (quotient n 2) (* x acc))))))
```

## 7.6 Transformation des récursions arborescentes, l'exemple de "fib"

### 7.6.1 Inefficacité du CPS

"Continuation passing slyle" appliqué à la fonction `fib`.

```
(define (k-fib n k) ; la continuation est appelee k
  (cond
        ((= n 0) (k 0)) ; application de la continuation à 0
        ((= n 1) (k 1)) ; application de la continuation à 1
        (#t (k-fib (- n 1)
             (lambda (acc1) ; construction de la fonction de continuation
                (k-fib (- n 2)
                       (lambda (acc2)
                         (k (+ acc1 acc2)))))))))
```

Cette transformation est moins efficace que pour factorielle :

— il reste un appel récursif dans la continuation,

— la consommation en pile est remplacée par une consommation mémoire via le code des continuations (voir ci-après).

Elle peut être utile si elle bénéficie d'une prise en charge spéciale par le compilateur.

### 7.6.2 Une solution en 0(n) : la "mémoization"

Cette solution s'applique globalement au calcul de toutes les suites.

**Memoization** : Du latin "memorandum". "In computing, memoization is an optimization technique used primarily to speed up computer programs by storing the results of function calls for later reuse, rather than recomputing them at each invocation of the function" - Wikipedia

Il est amusant d'écrire une solution purement applicative à ce problème :

```
(define (memo-fib n)

  (define cherche-dans-alist assq)

  (define (val n memoire) (cadr (cherche-dans-alist n memoire)))

  (define (memo n memoire)
    ;; rend une liste dans laquelle la valeur de fib(n) et toutes les
    ;; précédentes sont stockées
    (let ((dejaCalcule (cherche-dans-alist n memoire)))
      (if dejaCalcule
          memoire
          ;; il suffit de calculer fib(n-1), cela impliquera le calul
          ;; de fib(n-2) et son stockage dans ''memoire''.
          (let ((memoire (memo (- n 1) memoire)))
             (let ((fibn (+ (val (- n 1) memoire) (val (- n 2) memoire))))
                ;; on ajoute à la liste memoire la dernière valeur calculée et on la rend
```

```
18                  (cons (list n fibn) memoire))))))

20  (val n (memo n '((1 1) (0 0)))))
```

### 7.6.3 Une version itérative en O(n)

Transformation du problème : utiliser des variables ou la pile d'exécution pour conserver en mémoire les deux dernières valeurs qui sont suffisantes pour calculer la suivante.

Observons les valeurs successives de deux suites :

$a_n = a_{n-1} + b_{n-1}$ avec $a_0 = 1$

et

$b_n = a_{n-1}$ avec $b_0 = 0$

On note que pour tout n, $b_n = fib(n)$.

On en déduit la fonction récursive terminale suivante :

```
1  (define (ifib n)
2     (let boucle ((cpt 0) (a 1) (b 0))
3               (if (= cpt n)
4                   b
5                   (boucle (+ cpt 1) (+ a b) a)))))))
```

# Chapitre 8

# Abstraction de données, Arbres binaires, Dictionnaires

## 8.1 Abstraction de données

### 8.1.1 Définitions

**Donnée** : entité manipulée dans un programme, définie par un type.

ex : Les entiers 1, 3 et 5 sont des données d'un programme réalisant des calculs.

**Abstraction de données** : processus permettant de rendre des données abstraites, c'est à dire utilisables via un ensemble de fonctions constituant leur **interface** sans que l'on ait à connaître ou sans que l'on puisse connaître (selon la philosophie "prévention/répression" du langage) leur représentation en machine (dite **représentation interne**).

**Type Abstrait** : Définition abstraite (et par extension implantation) d'un type de données sous la forme de l'ensemble des opérations que l'on peut appliquer à ses éléments indépendamment de l'implantation de ces opérations et de la représentation interne de ces éléments.

**Interface** : ensemble des fonctions que l'on peut appliquer aux éléments d'un type abstrait.

On peut éventuellement distinguer dans l'**interface**

- la partie *création*, permettant de créer de nouveaux éléments du nouveau type,
- la partie *accès* pour accéder aux propriétés (lorsque ces fonctions d'accès appartiennet à l'interface) des éléments
- la partie *utilisation* ou *fonctionnelle* ou encore *métier* permettant de les utiliser.

La programmation par objet a généralisé la création de nouveaux types de données. L'interface de création est constituée de la fonction `new` et d' un ensemble de fonctions d'initialisation (généralement nommées *constructeurs*). On y parle également d'*accesseurs* et de fonctions "métier".

## 8.2 Définition de nouveaux types abstraits

Pascal : *Enregistrement - Record*

C, Scheme : *Structure - Struct*

JAVA : *Classe - Class* - Réalisation de l'encapsulation "données - fonctions".

### 8.2.1 Un exemple : les nombres rationnels (d'après Abelson Fano Sussman

#### Interface

*création* : make−rat
*accès* : denom numer
*métier* : +rat −rat ∗rat /rat =rat

## Implantation

```
(define (+rat x y)
    (make-rat
        (+ (∗ (numer x) (denom y))
            (∗ (denom x) (numer y)))
        (∗ (denom x) (denom y))))

(define (∗rat x y)
    (make-rat
        (∗ (numer x) (numer y))
        (∗ (denom x) (denom y))))
```

## Interface de création, Représentation No 1 : doublets, sans réduction

```
(define (make-rat n d) (cons n d))
(define (numer r) (car r))
(define (denom r) (cdr r))
```

## Les différents niveaux d'abstraction

```
utilisation des rationnels
−−−−−−−− +rat −rat make-rat numer denom
représentation des rationnels, Utilisation des doublets
−−−−−−−− doublets : cons, car, cdr
représentation des doublets
−−−−−−−− ???
```

## Interface de création, Représentation no 2, vecteurs et réduction

**Vecteur** : Collection dont les éléments peuvent être rangés et retrouvés via un index, qui occupe moins d'espace qu'une liste contenant le même nombre d'éléments.

Manipulation de vecteurs.

```
(make-vector taille) : création vide
(vector el1 ... eln) : définition par extension
(vector-ref v index) : accès indexé en lecture
(vector-set! v index valeur) : accès indexé en écriture
```

```
(define (make-rat n d)
  (let ((pgcd (gcd n d)))
    (vector (/ n pgcd) (/ d pgcd))))
```

```
(define (numer r) (vector-ref r 0))

(define (denom r) (vector-ref r 1))
```

## 8.3 L'exemple des Arbres binaires

**Graphe** : ensemble de sommets et d'arêtes

**Graphe orienté** : ensemble de sommets et d'arcs (arrête "partant" d'un noeud et "arrivant" à un noeud)

**Arbre** : graphe connexe sans circuit tel que si on lui ajoute une arrête quelconque, un circuit apparaît. Les sommets sont appelés **noeuds**.

Exemple d'utilisation : Toute expression scheme peut être représentée par un arbre.

**Arbre binaire** :

- soit un arbre vide

- soit un noeud ayant deux descendants (fg, fd) qui sont des arbres binaires

**Arbre binaire de recherche** : arbre dans lesquels une valeur$v$ est associé à chaque noeud $n$ et tel que si $n1 \in fg(n)(resp.fd(n))$ alors $v(n1) < v(n)$ (resp. $v(n1) > v(n)$)

**Arbre partiellement équilibré** : la hauteur du SAG et celle du SAD diffèrent au plus de 1.

### 8.3.1 Arbres binaires de recherche

- Création :
  (make-arbre v fg fd)
- Accès aux éléments d'un noeud n :
  (val-arbre n), (fg n), (fd n)
- Test
  (arbre-vide ? n)
- interface métier :
  - (inserer valeur a) : rend un nouvel arbre résultat de l'insertion de la valeur n dans l'arbre a. L'insertion est faite sans équilibrage de l'arbre.
  - (recherche v a) : recherche dichotomique d'un élément dans un arbre
  - (afficher a) : affichage des élements contenus dans les noeuds de l'arbre, par défaut affichage en profondeur d'abord.

#### Représentation interne - version 1

Consommation mémoire, feuille et noeud : 3 doublets.

Simple à gérer.

```
(define (make-arbre v fg fd)
  (list v fg fd))

(define (arbre-vide? a) (null? a))

(define (val-arbre a) (car a)) ;;

(define (fg a) (cadr a))

(define (fd a) (caddr a))
```

### Insertion sans duplication

```
(define (insere n a)
  (if (arbre-vide? a)
      (make-arbre n () ())
      (let ((valeur (val-arbre a)))
        (cond ((< n valeur) (make-arbre valeur (insere n (fg a)) (fd a)))
              ((> n valeur) (make-arbre valeur (fg a) (insere n (fd a))))
              ((= n valeur) a)))))
```

```
(define a (make-arbre 6 () ()))
(insere 2 (insere 5 (insere 8 (insere 9 (insere 4 a)))))
```

### Recherche dans un arbre binaire

Fonction booléenne de recherche dans un arbre binaire. La recherche est dichotomique.

```
(define (recherche v a)
  (and (not (arbre-vide? a))
      (let ((valeur (val-arbre a)))
        (cond ((< v valeur) (recherche v (fg a)))
              ((> v valeur) (recherche v (fd a)))
              ((= v valeur) #t)))))
```

### Seconde représentation interne

Coût mémoire, noeud (3 doublets), feuilles (1 doublet).

Nécessite un test supplémentaire à chaque fois que l'on accède à un fils.

```
(define (make-arbre v fg fd)
  (if (and (null? fg) (null? fd))
      (list v)
      (list v fg fd)))

(define (fg n) (if (null? (cdr n)) () (cadr n)))

(define (fd n) (if (null? (cdr n)) () (caddr n)))
```

Toutes les autres fonctions de manipulation des arbres binaires (interface fonctionnelle) sont inchangées.

## 8.4 L'exemple des Dictionnaires

Le type dictionnaire est un type récurrent en programmation, on le trouve dans la plupart des langages de programmation (java.util.Dictionary).

En scheme, il a été historiquement proposé sous le nom de listes d'association. Une liste d'associations est représentée comme une liste de doublets ou comme une liste de liste à deux éléments.

### Interface de manipulation

```
>(define l1 '((a 1) (b 2)))
>(assq 'a l1)
(a 1)
>(assq 'c l1)
#f
>(assoc 'a l1)
(a 1)
>(define l2 '( ((a) (une liste de un élément)) ((a b) (deux éléments)) ))
>(assq '(a b) l2)
#f
>(assoc '(a b) l2)
((a b) (deux éléments))
```

### Interface : construction et manipulations

Scheme n'a pas prévu d'interface de construction, on construit directement les listes.

Ceci implique que l'on ne peut pas changer la représentation interne des dictionnaires.

Imaginons une interface de construction

```
(define prem-alist car)
(define reste-alist cdr)
(define null-alist? null?)
(define make-alist list)
(define (add clé valeur l)
    ;; rend une nouvelle aliste incluant une nouvelle liaison clé−valeur
    ;; ne vérifie pas si la clé était déjà utilisée
    (cons (list clé valeur) l))
```

```
(define (my-assoc clé al)
 (let loop ((al al))
    (cond ((null-alist? al) #f)
       ((equal? (car (prem-couple al)) clé) (prem-couple al))
       (else (loop (reste-alist al))))))
```

# Chapitre 9

# Séquences et Effets de bord en programmation récursive.

## 9.1 Définitions

**Effet de bord** : modification explicite de l'état de l'ordinateur (de la mémoire ou de l'affichage).

**Séquence** : suite d'instructions.

Une instruction effectuant un calcul sans rendre explicitement une valeur effectue un effet de bord.

Fonction récursive avec effet de bord :

```
(define (display-elements l)
    (if (not (null? l))
        (begin
            (display (car l)) ;; effet de bord
            (display "\n") ;; effet de bord
            (display-elements (cdr l)))))
```

Exemple : Affichage en profondeur d'abord des données contenues dans un arbre binaire.

```
(define (affichage a)
  (or (arbre-vide? a)
      (begin (affichage (fg a))
             (display (val-noeud a))
             (affichage (fd a)))))
```

Dessins Récursifs

```
(define (feuille long angle)
  (if (> long 1)
      (begin
        (draw (/ long 2))
        (turn (- angle))
        (feuille (/ long 2) angle)
        (turn (* angle 2))
        (feuille (/ long 2) angle)
        (turn (- angle))
        (draw (/ long 2))
        (turn (- angle))
```

```
        (feuille (/ long 2) angle)
        (turn angle)
        ;(feuille (/ long 2) angle)
        (turn angle)
        (feuille (/ long 2) angle)
        (turn (- angle))
        (move (- long)))))

(define (naperon long angle)
  (let ((n (/ 360 angle)))
    (define (boucle long angle times)
      (if (> times 0)
          (begin
            (feuille long angle)
            (turn angle)
            (boucle long angle (- times 1)))))

    (boucle long angle n)))

(turtles)
(turn 90)
(naperon 200 60)
```

### 9.1.1 Modification physique de la mémoire

Les procédures set-car ! et set-cdr !

La fonction append destructrice.

```
(define (d-append l1 l2)
  (cond ((null? l1) l2)
        ((null? (cdr l1)) (set-cdr! l1 l2) l1)
        (#t (d-append (cdr l1) l2))))

(define matin (list 1 2 3 4 5 6 7 8 9 10 11 12))
(define soir (list 13 14 15 16 17 18 19 20 21 22 23 24))
(define heures (append matin soir))
```

Les listes circulaires.

```
(d-append heures heures)
```

#### Un arbre binaire impératif

A jout de nouvelles fonctions d'interface pour pouvoir modifier le fils gauche et le fils droit.

(define (set-fg ! a a2) (set-car ! (cdr a) a2))

(define (set-fd ! a a2) (set-car ! (cddr a) a2))

```
(define (p-insere n a)
  ;; version n'acceptant pas les insertions successives
  (display a) (display n) (display "\n")
  (if (arbre-vide? a)
      (make-feuille n))
```

```
    (let ((valeur (val-noeud a)))
      (cond ((< n valeur)
             (if (arbre-vide? (fg a))
                 (set-fg! a (make-feuille n))
                 (p-insere n (fg a))))
            ((> n valeur)
             (if (arbre-vide? (fd a))
                 (set-fd! a (make-feuille n))
                 (p-insere n (fd a))))
            ((= n valeur) a))))
```

```
(define l2 (make-feuille 5))
(p-insere 4 l2)
(p-insere 7 l2)
(p-insere 8 l2)
(p-insere 1 l2)
(p-insere 2 l2)
```

Exercice : Une version de la fonction p-insere qui autorise l'écriture de (set ! l (insere 2 (insere 1 (insere 8 (insere 7 (insere 4 l))))))

# Chapitre 10

# Fonctions récursives et Interprétation des programmes

## 10.1 Récursivité et Interprétation des programmes

Réalisons un langage de programmation défini par sa fonction d'interprétation de ses expressions. Dotons le minimalement pour débuter.

- — Des types de base et leurs fonctions primitives : entiers, booléens
  - — des identificateurs, définis par "let"
  - — Une conditionnelle, nommée "si", ayant la même sémantique que la conditionnelle usuelle.
- — Une syntaxe. Pour simplifier, prenons celle de Scheme. Nous disposons d'un analyseur syntaxique prêt à l'emploi, implanté par la fonction "read".
- — Une sémantique des constructions du langage.
  Prenons celle de Scheme et mettons la en oeuvre via une fonction d'interprétation : `eval302`.
  Rendons la fonction d'interprétation utilisable via une boucle "toplevel" implantée par la fonction `scheme302`.

```
1  (define (scheme304)
2
3    ;; sans fonction utilisateur
4    ;; sans environnement, hors global
5    ;; sans gestion de la pile
6
7    (let ((**global-env (list (list '+ +) (list '- -) (list '* *) (list '/ /) (list '= =)
8                              (list 'car car) (list 'cdr cdr) (list 'cons cons) (list 'append append) (list
                                   'list list)))
9          (**primitives '(+ - * / =))
10         (**returnToToplevel #f)
11         (**fin? #f))
12
13     (define (primitive? f) (memq f **primitives))
14
15     ;; Gestion des environnements
16     (define (env-value symbol env)
17       (let ((liaison (assq symbol env)))
18         (if liaison
19             (cadr liaison)
20             (error304 "variable_indéfinie" symbol))))
```

```
(define (eval304 e env)
  (cond ((or (number? e) (boolean? e) (string? e) (procedure? e)) e)
        ((symbol? e) (env-value e env))
        ((list? e)
         ;;les formes spéciales du langage
         (case (car e)
           ((fin) (set! **fin? #t) 'Au-revoir)
           ((si) (eval-si e env))
           ((let) (eval-let e env))
           ((definir) (eval-definir e env))
           (else
            (cond
              ((primitive? (car e))
               ;; c'est une fonction primitive de notre langage, on passe la main
               ;; à notre interpréteur hôte pour appliquer la fonction, en prenant soin
               ;; d'évaluer préalablement les arguments
               (apply (eval304 (car e) env) (evlis (cdr e) env)))
              (#t (error304 "fonction inconnue" (car e)))))))
        (#t (error304 "construction_inconnue" e))))

(define (eval-cite e env) (cadr e))

(define (eval-si e env)
  (if (eval304 (cadr e) env)
      (eval304 (caddr e) env)
      (eval304 (cadddr e) env)))

(define (eval-let e env)
  (let ((newenv (append (parallel-eval-bindings (cadr e) env) env)))
    (eval-sequence (cddr e) newenv)))

(define (parallel-eval-bindings bindings env)
  ;; rend un environnement augmenté des nouvelles liaisons des variables au
  ;; résultat de l'évaluation des expressions correspondantes, dans l'environnement
  ;; env
  (if (null? bindings)
      ()
      (append (list (list (caar bindings) (eval304 (cadar bindings) env)))
              (parallel-eval-bindings (cdr bindings) env))))

(define (eval-sequence lexp env)
  ;; evalue les expressions en séquence et rend
  ;; la valeur de la dernière
  (letrec ((boucle (lambda (lexp value)
                     (if (null? lexp)
                         value
                         (boucle (cdr lexp) (eval304 (car lexp) env))))))
    (boucle lexp ())))

(define (evlis l env)
  ;; reçoit une liste d'expressions et rend la liste de leurs valeurs
  (map (lambda (e) (eval304 e env)) l))

(define (eval-definir e env)
  ;; permet d'ajouter une liaison
  3
```

```
78      )

80      (define (error304 s l)
81        ; gestion primitive des exceptions
82        ; display + retour au toplevel
83        (display "Erreur: ") (display s) (display ", ") (display l) (display ".\n")
84        (**returnToToplevel **returnToToplevel))

86      (define (print304 e)
87        (display "= ") (display e) (display "\n"))

89      ;; Corps du let principal

91      (do ()
92        ;; la boucle while(true) selon Scheme

94        ;; pour quitter la boucle toplevel, écrire "(fin)"
95        ;; si la variable **fin vaut #t, sortie du toplevel
96        (**fin? 'A_bientôt) ;;

98        ;; gestion de base des exceptions.

100       ;; capture du point courant (une seule fois) pour y revenir après une exception
101       ;;
102       (or **returnToToplevel
103           (begin
104            (set! **returnToToplevel
105                  (call-with-current-continuation
106                   (lambda (k) k)))
107            (display "... Extra-ball\n")))

109       (print304
110        (eval304
111         (read)
112         **global-env))
113       )
114     )
115   )

117 (scheme304)
```