INITIATION À LA PROGRAMMATION FONCTIONNELLE

Myriam Desainte-Catherine

ENSEIRB – LaBRI – Université Bordeaux I

Première année ENSEIRB

Bibliographie

– Notes de cours personnelles sur la programmation en SCHEME
– "Traité de Programmation en Common Lisp" de R. Strandh et I. Durand
– "Common lisp" Guy Steele, digital Press. ISBN 1-55558-041-6

Table des matières

Chapitre 1

Introduction

1.1 Histoire du langage

1.1.1 Les langages des années 1950

1. FORTRAN : calcul scientifique. Données et calculs numériques.
2. Lisp : calcul symbolique. Données et algorithmes complexes issus de l'IA. Démonstrations automatiques, jeux etc. C'est un langage de programmation utilisant des listes comme structures de données et dont les calculs sont exprimés au moyen d'expressions et de récursivité.

1.1.2 Histoire des langages lisp

- Vers 1956-1958, John Mac Carthy, chercheur au MIT, développe le premier interprète lisp (pour LISt Processing language) lisp 1.5 qui est le premier à être diffusé.
- Plusieurs milliers de dialectes lisp sont développés au sein des laboratoires. Parmis eux, Maclisp (MACSYMA).
- Un premier effort de standardisation en 1969 : Standard Lisp. C'est un sous-ensemble de lisp 1.5 et autres dialectes. Portable Standard Lisp est ensuite construit à partir de Standard Lisp en l'enrichissant. Franz lisp : le premier lisp sur les machines unix.
- Les lisps actuels
  - Le dialecte Scheme apparut vers la fin des année 70 (Steele et Sussmsn. MIT). C'est une version plus propre et mieux définie sémantiquement. Concepts de portee lexicale, fermeture et continuations de première

classe.
- En 1986 un groupe de travail fut formé popur définir un langage ANSI Common lisp standard. La standardisation regroupe les concepts des diverses implémentations. Objectifs principaux
  - la portabilité
  - la consistence (une sémantique consistante, alors que les implémentations précédantes ne l'étaient pas toujours),
  - l'expressivité : l'expérience a montré l'intérêt de certaines formes par leur utilité et leur simplicité. Les autres ont été supprimées.
  - Efficacité ' : compilateur.
  - Stabilité : changements lents et précédés d'expérimentations et de délibérations.

  Le processus est toujours en marche. Chaque ajout suit lee étapes suvantes : besoin identifié, plusieurs propositions d'integration, une proposition est implémentée sous forme de macro et diffusée, exprérimentation et critiques, standardisation.
- emacs-lisp : langage d'extension de gnu-emacs. Le premier emacs fut développé par Richard Stallman en 1975. James Gosling (créateur de Java) développa le premier emacs en C pour Unix en 1982. Gosling permit intialement la libre distribution du code source de Gosling Emacs, que Stallman utilisa en 1985 pour la première version (15.34) de GNU Emacs. Emacs-lisp vu donc le jour le 20 mars 1985.

Aujourd'hui, on distingue deux sortes de langages lisp
- Le langage Common lisp est destiné à l'écriture d'applications en laboratoires et dans l'industrie.
- Les langages Scheme et e-lisp sont destinés à être utilisés en tant que langages d'extension d'applications. L'interprète Guile (dialecte Scheme) est utilisé dans le projet GNU et à l'oeuvre dans les logiciels gimp (Gnu Image Manipulation Program) et snd (édition de sons).

Problème du choix du langage fonctionnel pour illustrer ce cours...

1.2 Autres langages fonctionnels

- ML : R. Milner propose ML en 1978. CaML est développé et distribué par l'INRIA depuis 1984 et il est disponible gratuitement pour les ma-

chines Unix, PC ou Macintosh. Il existe deux dialectes de Caml : Caml Light et Objective Caml. Caml Light est un sous-ensemble d'Objective Caml, plus spécialement adapté à l'enseignement et à l'apprentissage de la programmation. En plus du coeur du langage de Caml Light, Objective Caml comporte un puissant système de modules, des objets et un compilateur optimisant. Projet Coq. En 1985 : D. Turner propose Miranda (ML avec lazy evaluation).
- Haskell : 1987 purement fonctionnel, paresseux. Programmation purement fonctionnelle.

1.3 Caractéristiques des langages lisp

- Expressions et instructions : évaluation mathématique et machines à états. Transparence référencielle.
- Effets de bord : séquences d'instructions et imbrication d'expressions.
- Fondements mathématiques : le langage lisp est basé sur la théorie des fonctions d'Alonzo Church (1940) appelée le λ-calcul. Deux concepts : l'abstraction fonctionnelle et la réduction d'expressions (dont l'application).
- similarité contrôle et données
- Typage dynamique et typage statique
- Garbage Collector et allocation manuelle

1.4 Modèle de développement

- Langage intéractif et incrémental.
- Boucle REP, top-level
- L'environnement de l'interprète peut à tout moment, au moyen de la boucle REP, être modifié par des ajouts de fonctions (toutes les fonctions globales sont au même niveau, pas de notion de `main`), de déclarations (initalisations de variables). Des tests peuvent être faits aussi interacivement.

1.5 Objectifs pédagogiques du cours

- Découverte d'une autre forme de programmation que la forme inpérative (on parle de paradigme de programmation). Cela ouvre de nouveau mécanismes mentaux d'analyse des problèmes et de synthèse résolutions.
- Approfondissement de la notion de fonction (utilisation et représentation).
- Découverte de nouveaux mécanismes de portées des variables (lexical et dynamique).
- Approfondissement de la notion de macroexpansion (mécanisme plus puissant que celui du langage C).
- Découverte du développement incrémental : REP

1.6 Contrôle

- Un projet (commun avec le module de graphes) doublement encadré.
- Un examen

Chapitre 2

Survol de la syntaxe du langage

Ceci est un survol des principales notions à acquérir pour démarrer, c'est-à-dire écrire quelques fonctions simples. Il y est décrit :

- La forme normale des expressions ainsi que l'évaluation des formes normales.
- Déclaration de variables et fonctions globales. La nuance entre variables spéciales et normales sera abordée dans le chapitre sur l'évaluation suivant. Les espaces de noms seront abordés dans le chapitre sur les fonctionnelles, ainsi que la lambda-expression.
- Les types de base numériques et booléens avec leurs opérations et prédicats (les caractères et les chaînes de caractères seront détaillés au besoin en TD).
- Les formes spéciales conditionnelles et de déclarations locales de variables et de fonctions.

Notation

```
* (+ 1 2)
3
```

- le caractère d'invite ou prompt : *
- affichage du résultat *3*

2.1 Les expressions

La construction de base du langage est l'expression, et non l'instruction. Un programme est un ensemble d'expressions juxtaposées ou imbriquées. On

distingue les expressions avec ou sans effets de bords.

Les expressions sont préfixées, complètement parenthésées et doivent être indentées.

On appelle **expressions symboliques** (sexpr) les formes syntaxiquement correctes :

- objet lisp (nb, chaîne, symbole)
- Expression composée (sexpr sexpr ... sexpr) : ceci est une liste d'éléments étant chacun une sexpr.

2.1.1 Définition de symboles globaux

L'environnement global est un dictionnaire formé de **liaisons**

$$symbole \longrightarrow valeur$$

On peut définir une liaison globale avec une déclaration **defparameter** ou **defvar** pour les variables et **defun** pour les fonctions. L'inconvénient de defvar est qu'elle ne réinitialise pas la variable (si celle-ci est déjà initialisée). Les symboles ne sont pas typés (non déclarés), mais leurs valeurs le sont. Il s'agit de **typage dynamiqyue**.

Il y a deux espaces de noms en lisp : celui des variables et celui des fonctions. En fait un symbole est représenté par une structure à plusieurs champs : valeur, fonction/macro, plist (liste de propriétés), paquetage (de noms). Ceci sera détaillé plus loin dans le cours.

```
* (defparameter a 1); valeur de type numérique
A
* a
1
* (defun f (x) x)
F
* (defparameter a ‘‘valeur de type chaine de caracteres’’)
```

2.1.2 Évaluation d'une expression

- Objets auto-évaluants : les nombres, booléens, caractères, chaînes de caractères, tableaux. Le résultat de l'évaluaiton d'un objet auto-évaluant est l'objet lui-même.

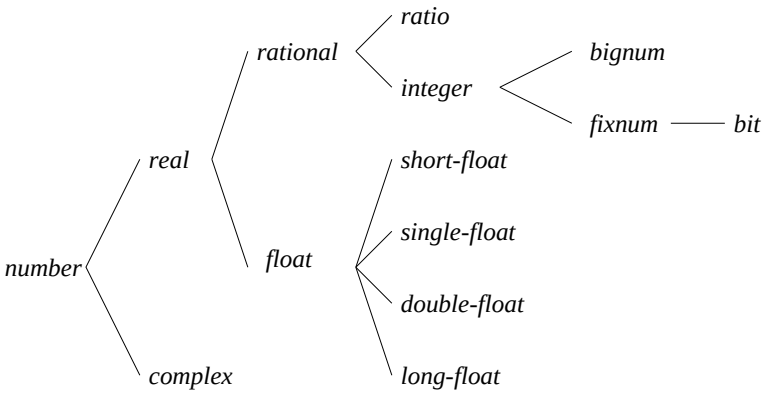

FIG. 2.1 – Arborescence des types numériques – *Grâcieusement cédée par R. Strandh*

- Symboles : identificateurs de variables ou de fonctions. Le résultat de l'évaluation d'un symbole est la valeur associée.
- Expression symbolique composée : si le premier objet de la liste est un symbole de **fonction**, la liste est considérée comme un appel à la fonction. On parle d"évaluation applicative d'une expression symbolique composée. Elle conssite à évaluer l'objet en position fonctionnelle (la première), évaluer les arguments puis appliquer la fonction aux arguments et renvoyer le résultat.
- Remarque : il existe des opérateurs spéciaux pour lesquels les arguments ne sont pas évalués selon l'algorithme précédent. Chacun a sa propre règle d'évaluation. Ex : if, cond... Ces expressions sont appelées **formes spéciales**. On y trouve aussi en particulier, les expressions avec effets de bord : affectation, déclarations, affichage.

2.2 Types simples

2.2.1 Les nombres

Entiers

Il y a deux sous-types **fixnum** et **bignum**. Les **fixnum** sont représentés sur un mot. La norme impose que fixnum soit sur au moins 16 bits, mais cela dépend des implémentations. Les **bignum** sont non limités en taille. Les

conversions sont automatiques.

- Deux constantes globales : **most-positive-fixnum** et **most-negative-fixnum**.
- Prédicats : **bignump** et **fixnump**

```
* most-positive-fixnum
536870911
* (log most-positive-fixnum 2)
29.0
* (fixnump 536870912)
NIL
```

- Réels : suite de chiffres contenant un point, notation 0.1e1
- Rationnels, complexes

Les ratios

Sous-type rational. Obtenus à partir des opérations : additions, soustractions, divisions et multiplications, et d'arguments entiers ou rationnels.

- Accesseurs : **numerator**, **denominator**
- Prédicats : **ratiop**

```
* (/ (* 536870911 10) 9)
5368709110/9
* (numerator 5368709110/9)
5368709110
* (denominator 5368709110/9)
9
* (ratiop 5368709110/9)
T
* (rationalp 5368709110/9)
T
```

Les flottants

Ex : 23.2e10

Le type **float** est distinct du type **rational**. Le type d'un résultat est choisi parmi les sous-types **short-float**, **single-float**, **double-float**, **long-float** et correspond au type le plus précis des arguments.

les complexes

```
* (sqrt -1)
#C(0.0 1.0)
* #C(1.0 0.0)
#C(1.0 0.0)
```

2.2.2 Les booléens

N'importe quel objet peut être considéré comme un booléen. Tous sauf le symbole **nil** sont assimilés à la valeur **vrai**.

L'objet **t** est utilisé comme valeur vrai, quand aucune autre valeur ne paraît plus pertinente.

Les symboles nil et t sont des variables globales, ayant pour valeur resp. nil et t.

Le symbole nil est aussi l'objet de terminaison des listes. Il est donc obtenu en tapant ().

- utiliser nil dans un contexte booléen
- utiliser '() pour signifier fin de liste dans une structure de données.
- utiliser () dans les macros.

2.2.3 Prédicats

- typage : **numberp**, **realp**, **complexp**, **rationalp**, **integerp**, **floatp**
- autres : **zerop**, **plusp**, **minusp**, **evenp**, **oddp**

2.2.4 Comparaisons

- = /= ¡ ¡= ¿ ¿= sur les réels.
- Égalités et inégalités sur les complexes.
- Nombre d'opérandes supérieur strictement à 0 (ex : ēst vrai si tous les opérandes sont différents).
- Égalité à nil (ou '()) : **null**.

2.2.5 Opérations

1. Opération numériques

- Arithmétiques :**+, -, *, /** (nb quelconque d'arguments)
  Soustraction (resp. division) : premier argument - (resp. /) somme (resp. produit) des arguments suivants. – mod, rem.
- **max** et **min** : arguments réels.
- trigonométrie : **sin**, **cos**, **tan**, **asin**, **acos**, **atan**, **sinh**, **cosh**, **tanh**, **asinh**, **acosh**, **atanh**
- **log** (calcule le log népérien par défaut, accepte un deuxième argument optionnel servant de base), **exp** (naturelle), **expt** (d'une base arbitraire).
- opérations bit à bit (sur les entiers) : **logand**, **logandc1** (et entre complément du premier et le deuxième), etc. p64
- conversions, troncatures, arrondis : **floor** (infini négatif), **ceiling** (infini positif), **truncate** (troncature vers 0), **round** (arrondi entier le plus proche).

2. Opérations booléennes : ce sont des opérateurs spéciaux (voir section 2.5)
   - **and** : s'arrete au premier faux
   - **or** : s'arrete au premier vrai
   - **not**

   Les opérateurs *and* et *or* admettent n arguments, $n \geq 0$.

```
* (and)
T
*  (or)
NIL
```

2.3 Les symboles

Suite de caractères quelconques (parenthèses, espace, etc. précédés d'un antislash, ou bien entouré de ——) ne pouvant pas être interprétée comme un nombre. Ils servent d'identificatuers de variables et de fonctions. Ils représentent aussi des données (symboliques ou non numériques).

- Prédicat de type : **symbolp**
- Prédicat d'égalité : **eq**
- variables spéciales et lexicales : globales, locales, paramètres de fonctions. **defvar**, **defparameter**

```
* (defvar |Je suis un symbole| 1)
```

```
|Je suis un symbole|
* |Je suis un symbole|
1
```

2.4 Les chaînes de caractères

- Caractère : #\a
- Chaîne : "de caractères"
- Prédicats : **characterp**, **stringp**
- Comparaisons : char=, char/=, char¡, char¿, char¡=, char¿=, string=, string/=, string¡, string¿, string¡=, string¿=.

2.5 Les expressions spéciales

2.5.1 Les expressions conditionnelles

if

```
(if <condition> <alors> <sinon>)
(if <condition> <alors>)
```

Remarque. Dans la 2ième forme, si la condition est fausse, NIL est retourné.

cond

Il correspond à l'imbrication de **if**.

```
(cond ((numberp x)  ''X est un nombre'')
      ((stringp x) ''X est une chaine'')
      ((symbolp x) ''X est un symbole'')
      (t (...)))
```

2.5.2 Les expressions avec déclarations locales

Déclaration de variables

- La forme **let**

```
(let ((a 2)
      (b 3)
      (c 0))
  (- (* b b)
     (* 4 a c)))
```

Signifie : Soient a=2, b=3, c=0, $b^2 - 4ac$. L'évaluation des valeurs d'initialisation est effectuée en premier (dans l'ordre de l'écriture), puis les variables locales sont créees. Ce qui implique que les valeurs des variales locales définies dans un let ne sont pas utilisées dans l'évaluation des expressions d'initialisation.

```
(defparameter a 0)
(let ((a 2)
      (b 3)
      (c a))
  (- (* b b)
     (* 4 a c)))
```

Le symbole a associé à c dans le let prend la valeur 0 et non 2.

– La forme **let***

```
(let ((a 2)
      (b 3)
      (c a))
  (- (* b b)
     (* 4 a c)))
```

L'évaluation des expressions d'initialisation est effectuée après la création des variables locales.

Déclaration de fonctions

– La forme **flet**

```
(flet (fn-1
       fn-2
       ...
       fn-k)
  expr1
  expr2
  ...
  exprn)
```

Chaque fn-i est de la forme suivante

```
(nom (p1 p2 ... pn) e1 e2 ... en)
```

Les *ei* ne peuvent pas référencer les fonctions locales.

– La forme **labels** permet la récursivité entre les fonctions et a la même forme syntaxique que $flet$.

Chapitre 3

L'évaluation

Ce chapitre est destiné à clarifier la gestion des environnements. Comment la valeur d'un symbole est-elle recherchée ? La récursivité terminale est aussi abordée.

3.1 Les environnements

Définition Un environnement est un ensemble de liaisons.

Il y a un environnement global constitué des liaisons construites par les définitions globales : defvar, defparameter, defun.

Il y a des environnements locaux fabriqués avec les expressions let, let*, flet et labels. Ces derniers peuvent être imbriqués.

3.1.1 Portées des liaisons

Définition La portée d'un symbole est l'ensemble des expressions du programme pouvant l'utiliser.

- Expressions let et flet : la portée des liaisons établies dans un let est limitée à l'intérieur du corps. Cela correspond aux déclarations locales à un bloc dans le langage C.
- Expressions defvar, defparameter, defun : la portée d'une liaison établie par ces formes couvre tout le programme. Cela correspond aux déclarations en dehors de tout bloc en langage C.

3.1.2 Durée de vie des liaisons

Définition La durée de vie d'un symbole correspond aux instants de l'éxecution qui sont situés entre le moment où il est crée et celui où il est détruit.

En général, la durée de vie d'une liaison globale est égale à la durée de la session, alors que celle d'iune liaison locale est limitée à l'éxecution du bloc où elle est définie. C'est le cas des variables du langage C, à part les static automatiques (qui sont conservées entre deux appels).

En Common Lisp, la durée de vie des liaisons locales est potentiellement égale à la durée de la session, qu'elles soient locales ou globales. En effet, les liaisons locales ne sont pas détruites immédiatement après l'évaluation de l'expression, mais sont conservées jusqu'à ce que le ramasse-miettes les ramasse. Ce dernier les détruit uniquement si elles sont inutilisées. On verra plus loin qu'il est possible de les maintenir au moyen d'une référence externe à l'environnement local.

3.1.3 Stratégies de recherche d'une liaison

Position du problème Pour évaluer une expression, il faut pouvoir associer une valeur à chaque symbole. Cela revient à rechercher une liaison correspondant au nom du symbole recherché. Dans l'environnement total (l'ensemble des liaisons globales et locales), il peut y avoir plusieurs liaisons admettant le même nom de symbole. Ceci est un problème qui se pose dans tous les langages de programmation. Deux sortes de stratégies existent et sont implémentées dans les différents langages existants. Common Lisp présente la particularité d'implémenter les deux.

Ces exemples introductifs sont exprimés en langage C.

– Cas des paramètres de fonctions.

```
int i=0;

int
f(int i)
{
  return i; /* Que vaut i ? */
}
```

– Cas des variables locales à une fonction.

```
int i=0;
```

```
int f(int x)
{
  int i=3;

  return i*x;  /* Que vaut i ? */
}
```

– Cas des **variables libres** d'une fonction.

```
int i=0;

int
int f(int x)
{
  return i*x;
}
```

C'est dans ce cas uniquement que se pose le problème, car le résultat dépend du langage. Soit l'appel suivant

```
int
g()
{
  int i=2;
  return f(3);
}
```

Le résultat est-il : `0*3` *ou bien* `2*3` ?

La stratégie lexicale donne le premier résultat, la stratégie dynamique le deuxième. Bien sûr, le langage C implémente la stratégie lexicale.

Stratégies lexicale et dynamique

– Pour chercher la liaison correspondant à l'occurrence d'un symbole dans une expression, la **stratégie lexicale** consiste à remonter les environnements locaux englobants du plus proche jusqu'à l'environnement global. La première liaison dont le nom de symbole correspond est retenue. Cette stratégie s'applique aussi à l'évaluation du corps d'une fonction lors d'une application. En effet, celui-ci est évalué dans l'environnement englobant de la fonction, dit **environnement lexical**. Cette stratégie

correspond au langage C et aux langages impératifs en général, ainsi qu'au langage fonctionnel Scheme.
- Pour chercher la liaison correspondant à l'occurrence d'un symbole dans une expression située dans le corps d'une fonction, la **stratégie dynamique** consiste à rechercher sa liaison dans l'**environnement dynamique**, c'est-à-dire l'environnement d'application de la fonction. Cette stratégie correspond par exemple à LaTeX, et beaucoup de lisp dont emacs-lisp.

3.1.4 Variables lexicales et spéciales

En Common Lisp, on distingue deux sortes de variables. Pour les introduire, reprenons le troisième exemple du langage C en lisp. (Les deux premiers correspondent au même mécanisme).

```
(defparameter i 0)
(defun f(x)
  (* x i))

(defun g()
  (let ((i 2))
    (f 3)))
```

Le résultat de l'application de g : `(g)` est 6. La variable `i` est dite spéciale.

Par contre, dans l'exemple suivant, la variable `i` est déclarée différemment (localement au moyen d'un `let`, et sa liaison est recherchée selon une stratégie lexicale.

```
(let ((i 0))
  (defun f(x)
    (* x i)))

(defun g()
  (let ((i 2))
    (f 3)))
```

Le résultat de l'application de g : `(g)` est 0. La variable `i` est dite lexicale.
- Les **variables spéciales** sont recherchées selon une stratégie dynamique. Ces variables sont construites au moyen des formes **defvar**, **defparameter**. Elles peuvent être modifiées par les formes **let** et **let***.

– Les **variables lexicales** sont recherchées selon une stratégie lexicale. Elles sont construites au moyen des formes **let** et **let***, dans le cas où leur nom diffère de ceux des variables spéciales.

L'intérêt des variables spéciales est principalement de paramétrer des fonctions par un autre moyen que l'ajout de paramètres ou l'utilisation de variables globales. En effet, le nombre de paramètres d'une fonction ne peut croître au delà d'une certaine limite sans causer des désagréments. Et l'utilisation de variables globales posent le problème de la réinitialisation en cas d'erreur de la fonction.

Par exemple, en LaTeX.

Exemple. Une définition de variable spéciale ne change pas le statut des variables lexicales définies précédemment et portant le même nom.

```
* (let ((b 2))
  (defun f(x) (+ b x)))
F
* (f 4)
6
* (defvar b 0)
B
* (let ((b 10))
  (defun h() b))
H
* (h)
0
```

3.2 Récursivité

3.2.1 Exemple

Soit la fonction factorielle

$fact(0) = 1$ et $fact(n) = n * fact(n-1)$ si $n > 0$.

```
(defun fact(n)
  (if (zerop n)
      1
      (* n (fact (- n 1)))))
```

```
* (trace fact)
(FACT)
* (fact 4)
  0: (FACT 4)
    1: (FACT 3)
      2: (FACT 2)
        3: (FACT 1)
          4: (FACT 0)
          4: FACT returned 1
        3: FACT returned 1
      2: FACT returned 2
    1: FACT returned 6
  0: FACT returned 24
24
```

Le processus de calcul est typiquement récursif car toutes les multiplications sont laissées en suspend en attendant le retour des appels récursifs. Il est donc nécessaire de conserver les valeurs locales pour pouvoir effectuer les multiplications. Ces valeurs sont stockées dans une pile, qui grossit en fonction des appels et décroît en fonction des retours.

Ici, la liaison $n \longrightarrow valeur$ est empilée à chaque appel et dépilée à chaque retour.

3.2.2 Récursivité terminale

Si nous écrivions ce programme en C, nous utiliserions : une boucle itérative au lieu d'une boucle récursive et deux variables au lieu d'une. Nous allons comparer les écritures.

```
int
fact(int n)
{
  for (int r=1; n>0; n--)
    r*=n;
  return r;
}
```

La même calcul au moyen de deux variable peut s'écrire en lisp, en gardant l'écriture récursive.

```
(defun fact-iter(n r)
  (if (zerop n)
      r
    (fact-iter (- n 1) (* r n))))


* (trace fact-iter)
(FACT-ITER)
* (fact-iter 4 1)
  0: (FACT-ITER 4 1)
    1: (FACT-ITER 3 4)
      2: (FACT-ITER 2 12)
        3: (FACT-ITER 1 24)
          4: (FACT-ITER 0 24)
          4: FACT-ITER returned 24
        3: FACT-ITER returned 24
      2: FACT-ITER returned 24
    1: FACT-ITER returned 24
  0: FACT-ITER returned 24
24
```

On constate que le résultat final est calculé au plus bas niveau de la récursivité. Aucun calcul n'est donc nécessaire au retour des appels, donc aucun besoin de sauvegarder les valeurs des environnements locaux. Ce processus de calcul est dit itératif. On dit que la récursivité est terminale.

Définitions

- Un appel récursif est dit terminal si aucun calcul n'est effectué entre son retour et le retour de la fonction appelante.
- Une fonction est dite récursive terminale si tous les appels récursifs qu'elle effectue sont terminaux.

La plupart des langages sont développés de telle sorte que l'éxecution de toute fonction récursive consomme un espace mémoire croissant linéairement ave cle nombre d'appels, même si le processus de calcul décrit est en fait itératif. Dans ces langages, la seule façon d'engendrer unprocessus de calcul itératif est d'utiliser les structures de contrôle itératives.

Dans les langages lisp, et en particulier en Common Lisp et en Scheme, la récursivité terminale est gérée dans la plupart des implémentations. Une pile est utilisée seulement quand cela est nécessaire.

Pour rendre une fonction récursive terminale, il suffit bien souvent de lui ajouter un argument dans lequel on accumulera le résultat. Cela est possible seulement si la récursion est linéaire et si l'opération est associative. Des exemples avec des opérations non associatives seront abordés plus loin.

Chapitre 4

Listes

Dans ce chapitre on montre que l'on manipule sans le savoir depuis le début du cours des listes. La syntaxe des programmes est similaire à la syntaxe des données.

L'ensemble des expressions symboliques est partitionné en deux sous-ensembles qui sont les objets auto-évaluants et les expressions composées. Du point de vue des structures de données, on a une partition en deux sous-ensembles qui sont les atomes et les paires pointées. Les listes non vides sont incluses dans les paires pointées alors que la liste vide est un atome (nil ou()).

4.1 Symboles et citations

4.1.1 La forme quote

```
* (quote Pierre)
Pierre
* 'Pierre
Pierre
```

4.1.2 Le prédicat de type symbolp

```
* (symbolp Pierre)
Error in KERNEL::UNBOUND-SYMBOL-ERROR-HANDLER:  the variable
  PIERRE is unbound.
* (defparameter Pierre 0)
Pierre
```

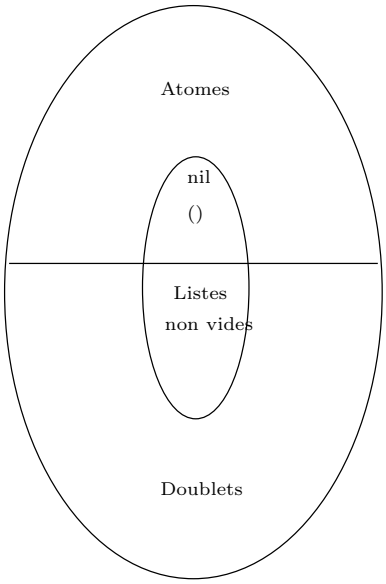

FIG. 4.1 – Les objets lisp

```
* (symbolp Pierre)
nil
* (symbolp 'Pierre)
T
```

4.1.3 Egalité eq

```
* (eq 'pierre 'Pierre)
T
```

4.2 Les paires pointées

4.2.1 L'opération cons

Opération non associative, non commutative.

```
* (cons (cons (cons 1 2) 3) 4)
(((1 . 2) . 3) . 4)
* (cons 1 (cons 2 (cons 3 4)))
```

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

4.2.2 Les sélecteurs car et cdr

Attention (car '()) donne (), de même cdr. Il faut impérativement tester le paramètre avant l'application pour différencier les cas.

4.2.3 Exemples

Exemples de structures complexes : (defun f (x y) (* x y))
Abbréviations. Ajouter nthcdr avec pour arguments : numéro du cdr, liste.

4.2.4 Prédicat consp

4.2.5 Affichage d'une structure complexe

La règle est la suivante :
- (a . (pp)) $\longrightarrow$ (a pp) si pp est une paire pointée.
- (a . ()) $\longrightarrow$ (a)

4.3 Les listes

4.3.1 Définition

On définit les listes récursivement. Une liste est soit
- la liste vide notée **()**, équivalente à **nil**.
- la forme **(cons E L)**, où **E** est une expression symbolique et **L** est une liste.

4.3.2 Prédicats listp et endp

Attention : **endp** ne marche que pour les listes. Utiliser **null** dans les autres cas.

4.3.3 Prédicats d'égalité

eq, equal

4.3.4 Fonction de construction list

4.3.5 Listes et évaluation

```
* (list 1 2 3)
(1 2 3)
* (1 . (2 3))
(1 2 3)
* '(1 2 3)
* (deparameter f '(defun f (x) x))
(defun f (x) x))
* (cons '(1 2) 3)
((1 2) . 3)
```

4.3.6 Fonctions prédéfinies

- **list-length**
- **copy-list** : discours sur la copie et le partage.
- **append** (sans effets de bord), nconc sera vu plus tard car elle introduit un effet de bord. Ces fonctions sont n-aires. Les arguments ne sont pas forcément des listes. C'est la dernière paire pointée de l'argument n qui est remplacée par la première de l'argument $n+1$.

  ```
  * (append '(1 . (2 . 3)) '(1))
  (1 2 1)
  * (append '() '())
  ```

- **reverse**
- **subseq** a trois arguments : liste, index du premier dans la sous-liste (index à partir de zéro), index du premier qui n'est pas dans la sous-liste.
- **first**, ... **tenth**, **nth** : index et liste
- **last**, **but-last**.
- **member** (member-if, member-if-not : plus tard) : élément, liste (test eq par défaut, mots clef :test pour changer).

Remarque. Il faut bien savoir différencier les actions de **cons**, **list** et **append**.

```
* (cons '(a b) '(1 2 3))
```

```
((A B ) 1 2 3)
* (list '(a b) '(1 2 3))
((A B ) (1 2 3))
* (append '(a b) '(1 2 3)))
(A B 1 2 3)
```

4.4 Programmation récursive sur les listes plates

On utilise le schéma récursif de la définition des listes.

4.4.1 Premier exemple

```
(defun somme (l)
  (if (endp l)
      0
    (+ (car l) (somme (cdr l)))))
```

4.4.2 Autres exemples

copie, concat, iota, reverse : amener l'écriture de reverse par récursion terminale de iota ou copie.

Récursivite terminale. L'opération *cons* n'étant pas associative, l'ordre de construction des paires pointées est important. Il s'avère nécessaire d'inverser la liste résultat.

4.5 Les listes d'association : a-listes

4.5.1 Définition

C'est une liste de paires pointées. Le car de chaque paire est généralement une clef et elles servent à représenter des tables (indexées), des dictionnaires, des environnements.

4.5.2 Fonctions

- La fonction **assoc** admet deux paramètres : une clef et une a-liste. Elle parcourt la liste et renvoie la première paire pointée dont le car est égal à la clef, et *nil* sinon.
- La fonction **acons** permet d'ajouter au début d'une aliste une paire pointée : `(acons 'Juliette "juliette@wanadoo.fr" *carnet*)`

```
*(defparameter *carnet*
  '((Pierre "pierre@labri.fr")
    (Régis  "dupond@aol.com")
    (Laure  "Durand@laposte.net")))
*CARNET*
* (assoc 'Laure *carnet*)
(LAURE "Durand@laposte.net")
* (cdr (assoc 'Laure *carnet*))
("Durand@laposte.net")
* (cadr (assoc 'Laure *carnet*))
"Durand@laposte.net"
* (acons 'Juliette "juliette@wanadoo.fr" *carnet*)
((JULIETTE . "juliette@wanadoo.fr") (PIERRE "casteran@labri.fr")
 (|RéGIS| "dupond@aol.com") (LAURE "Durand@laposte.net"))
```

Remarque. Comme pour la fonction member, il existe **assoc-if** et **assoc-if-not** que l'on présentera plus tard, à cause de l'argument fonctionnel.

4.6 Listes propres

A voir en TD, probablement.

4.7 Structures quelconques

Les fonctions agissant sur les structures quelconques sont basées sur la définition récursive suivante. Une structure complexe est

- Soit un atome
- Soit une paire pointée admettant pour *car* et pour *cdr* une structure complexe.

On obtient une récursivite double.

Il est souvent nécessaire de différencier l'atome *nil* des autres, car il correspond aussi à la liste vide. Voici un exemple.

```
(defun applatir(l)
  (cond ((null l) l)
        ((atom l) (list l))
        (t (append (applatir (car l))
                   (applatir (cdr l))))))
```

Si on ne différencie pas *nil* des autres atomes, on introduit des listes vides à la fin de chaque sous-liste, puisqu'on l'inclut à nouveau dans une liste afin de le concaténer.

Exercice de TD. Rendre la fonction récursive terminale sur un de ses deux appels.

Chapitre 5

Fonctions

5.1 Paramètres et retours de fonctions

5.1.1 Liste de paramètres

Paramètres optionnels

- `&optional` : les paramètres apparaissant après ce mot dans la définition de la fonction sont facultatifs.
- Un paramètre facultatif peut être initialisé par défaut, pour le cas où il est utilisé même quand l'appelant ne l'a pas spécifié. Il est alors remplacé par une liste à deux éléments : son nom et sa valeur (expression).

```
* (defun f(a b &optional c d)
      (list a b c d))
F
* (f 1 2)
(1 2 NIL NIL)
* (f 1 2 3 4)
(1 2 3 4)
* (defun g(a b &optional (c 0) d)
      (list a b c d))
G
* (g 1 2)
(1 2 0 NIL)
```

- Il est possible de savoir si la valeur d'un paramètre a été donnée lors de l'application ou non en ajoutant un paramètre supplémentaire dans

la liste d'initialisation. Sa valeur est booléenne : vrai si le paramètre a été donné, faux sinon.

```
* (defun g(a b &optional (c 0 c-supplied-p) d)
      (list a b c c-supplied-p d)
G
* (g 1 2)
(1 2 0 NIL NIL)
* (g 1 2 3 4)
(1 2 3 T 4)
```

– `&rest` : après ce mot les paramètres sont facultatifs et sont mis dans une liste. Cela est utile pour écrire des fonctions à nombre d'arguments variable. Ces paramètres-là ne peuvent pas être initialisés par défaut.

```
* (defun f(a b &optional (c 0) &rest d)
     (list a b c d))
F
* (f 1 2 3 4 5 6 7)
(1 2 3 (4 5 6 7))
* (g 1 2)
(1 2 0 NIL)
```

Paramètres mots clefs

Après le mot `&rest`, des mots-clefs peuvent apparaître dansla définition de la fonction. Ils sont précédés du mot `&key`. Ils peuvent être initialisés comme les paramètres facultatifs. Les mots-clefs non attendus dans une fonction sont ignorés seulement si le mot `&allow-other-keys` apparaît à la fin.

```
* (defun f(a &rest r &key (b 0 b-supplied-p) &allow-other-keys)
     (list a b b-supplied-p r))
F
* (f 1 :y 2 :c 3)
(1 0 NIL (:Y 2 :C 3))
* (f 1 :c 2 :y 3  :b 0)
(1 0 T (:C 2 :Y 3 :B 0))
```

5.1.2 Résultat multiple

La forme `(values v1 v2 ... vn)` permet d'associer plusieurs valeurs afin de les retourner en résultat d'une fonction.

```
(defun sumdiff(x y)
   (values (+ x y) (- x y)))
```

Par défaut, seule la première valeur est récupérée. Pour accéder aux deux valeurs on utilise la forme `multiple-value-bind`

```
(multiple-value-bind (a b)
   (sumdiff 3 5)
  (* a b))
```

Le nombre maximal de valeurs est indiqué dans la variable `multiple-value-limit`.

Remarque. Plusieurs fonctions de bibliothèque utilisent ce mécanisme. Par exemple, `floor`, `ceiling`, `truncate` et `round` admettent en arguments un nombre n et un diviseur d (par défaut à 1). Elles renvoient un quotient q et un reste r tels que : $qd + r = n$, où n est divisé par q et le résultat est converti en entier.

5.2 Les fonctions anonymes

Une λ-expression est une expression dont le résultat est une fonction anonyme. La notation lisp des fonctions anonymes est héritée des travaux d'Alonzo Church (1940), mathématicien logicien, sur la théorie des fonctions, appelée le λ-calcul. Il a développé le λ-calcul pour fournir une base rigoureuse à l'étude de la notion de fonction et d'application de fonction. Le λ-calcul est devenu un outil pour l'étude de la sémantique des langages de programmation.

5.2.1 Notation

L'expression $\lambda x.10x + 2$ a pour résultat la fonction (anonyme) qui au paramètre x associe associe l'expression $10x + 2$.

Pour définir des fonctons à plusieurs paramètres, on peut utiliser une des deux notations :

– $\lambda xy.10xy + 2$
– $\lambda x\lambda y.10xy + 2$

On passe de la première notation à la deuxième par une opération appelée **curryfication**, qui conssite à enlever un argument à une fonction en le "retardant" de façon à le faire passer à son résultat.

En lisp, on écrit

```
(lambda(p1 p2 ... pn) e1 e2 ... ek)
```

Par exemple

```
* (lambda(x) (+ (* 10 x) 2))
#<Interpreted Function (LAMBDA (X) (+ (* 10 X) 2)) 48012579>
```

Exercice. Une expression dont le résultat est une fonction à trois arguments numériques et qui renvoie le plus grand.

5.2.2 Application de λ et expressions *let*

Les λ-expressions correspondent aux fonctions lisp. L'application se fait par valeur.

Une application directe d'une λ-expression est possible de la façon suivante :

```
* ((lambda(x) (+ (* 10 x) 2)) 1)
12
```

Propriété. L'application directe d'une λ-expression est équivalente à une expression *let*. n En effet, les paramètres formels de la λ-expression correspondent aux variables locales du let, les paramètres effectifs (d'application) correspondent aux valeurs d'initialisation du let, et les expressions constituant le corps de la $\lambda - expression$ correspondent aux expressions du corps du let.

```
* (let ((x 1))
    (+(* 10 x) 2))
12
```

5.2.3 Autres opérations sur les λ-expressions

Il est possible de faire passer une λ-expression en argument d'une fonction et de retourner une λ-expression comme résultat d'une fonction.

1. Passage en argument

```
* (member 2 '(1 2 3 4 5) :test (lambda(x y) (= (* 2 x) y)))
(4 5)
```

2. Retour de fonction

```
* (lambda(x) (lambda(y) (= (* 2 x) y)))
#<Interpreted Function (LAMBDA (X) (LAMBDA (Y) (= # Y)))
  48053879>
```

3. Applications

```
* ((lambda(x) (lambda(y) (= (* 2 x) y))) 1)
#<Interpreted Function "(LAMBDA (X) (LAMBDA # #)) 1"
4809D501>
* ((lambda(x) (lambda(y) (= (* 2 x) y))) 1) 2)
In: ((LAMBDA (X) (LAMBDA # #)) 1) 2
  (((LAMBDA # #) 1) 2)
Error: Illegal function call.
* (funcall ((lambda(x) (lambda(y) (= (* 2 x) y))) 1) 2)
T
```

Remarque 1. Seule l'application directe d'une λ-expression est acceptée par le compilateur. L'utilisation de **funcall** est requise dans les autres cas (voir plus loin).

Remarque 2. Apparemment, print n'affiche que le premier niveau des λ-expressions. Cela ne dépend pas de `*print-level*`

5.3 Fonctions et espaces de noms

5.3.1 Représentation des symboles

Un symbole est représenté par plusieurs champs : nom, valeur, fonction, plist, package.

```
* (defun f(x) x)
F
* (inspect 'f)
F is a symbol.
```

```
0. Value: Unbound
1. Function: #<Interpreted Function F 480F7771>
2. Plist: NIL
3. Package: #<The COMMON-LISP-USER package, 20/21 internal, 0/9 external>
> 1
#<Interpreted Function F 480F7771> is a funcallable-instance.
0. %NAME: F
1. ARGLIST: (X)
2. LAMBDA: (LAMBDA (X) (BLOCK F X))
3. DEFINITION: NIL
4. GCS: 0
5. CONVERTED-ONCE: NIL
6. CLOSURE: NIL
```

5.3.2 Évaluation des symboles

Par défaut, un symbole est considéré comme une variable, et son évaluation donne lieu à un accés à sa valeur (premier champ). Pour accéder à la valeur fonctionnelle, on utilise la forme **function** (abbréviation #'), ou **symbol-function** qui ne sont pas équivalentes. Nous utiliserons principalement *function*.

- **function** tient compte des définitions locales et n'évalue pas son argument ;
- **symbol-function** ne tient pas compte des définitions locales et évalue son argument ;

```
* f
Error in KERNEL::UNBOUND-SYMBOL-ERROR-HANDLER:  the variable
  F is unbound...
* (function f)
#<Interpreted Function F 480F7771>
* #'f
#<Interpreted Function F 480F7771>
```

5.3.3 Espace des noms des fonctions

Les fonctions définies au moyen de **defun**, de **flet** et **labels** ont leur valeur fonctionnelle dans le champ fonction de la structure du symbole défini. On dit qu'elles sont dans l'espace des noms de fonctions.

Application

La forme `(nom-fonction arg1 ...)` permet d'appliquer une telle fonction car la valeur associée au symbole `nom-fonction` est accédée dans le champ "fonction" de la structure du symbole associé à la fonction. On dit que ce symbole est en position foncitonnelle.

Passage en paramètre

Dans les autres situations, c'est le champ valeur des symboles qui est accédé lors d'une évaluation. En particulier, quand le symbole d'une fonction est passé en paramètre. Dans ces cas-là, il faut utiliser la forme *function* ou bien #' pour que ce soit bien le champ fonction qui soit utilisée.

```
* (defun true(x y) t)
TRUE
* (member 2 '(1 2 3) :test #'true)
(1 2 3)
* (member 2 '(1 2 3) :test true)
Error in KERNEL::UNBOUND-SYMBOL-ERROR-HANDLER:  the variable
  TRUE is unbound
```

5.3.4 Espace des noms de variables

Il est possible d'initialiser une variable
- avec la forme **defparameter** au moyen d'une valeur fonctionnelle, par exemple obtenue au moyen d'une λ-expression ;
- lors d'un passage de paramètres ;

Dans ces deux cas, d'après ce qui est dit précédemment, c'est le champ "valeur" du symbole qui est affecté et non le champ "fonction". Ces fonctions nécessitent alors une forme spéciale pour être appliquées.

Application

Une telle fonction ne peut pas être appliquée en la mettant simplement en position fonctionnelle, puisque la fonction se trouve dans le champ "valeur" et non le champ "fonction". Il faut utiliser la forme

```
(funcall nom-fonction paramètres)
```

Dans cette forme, la valeur associée au symbole `nom-fonction` est accédée dans le champ "valeur" de la structure.

1. Définition au moyen de `defparameter`

```
* (defparameter g (lambda(x) (- x)))
G
* (funcall g 1)
-1
```

2. Passage de paramètre : les valeurs sont stockées dans le champ "valeur" des symboles des paramètres.

```
* (defun of-zero (f)
       (funcall f 0))
OF-ZERO
* (of-zero #'f)
0
```

Remarque

En fait, cette forme marche aussi dans l'espace des fonctions, je suppose par souci de compatibilité ? ?

```
*(funcall #'+ 1 2 3)
6
```

5.3.5 Fonctions anonymes

Les fonctions anonymes ne sont pas stockées dans des structures, puisqu'aucun symbole ne leur correspond. Ces expressions peuvent être utilisées telles quelles, ou bien précédées de #'. La différence entre les deux formes m'échappe pour l'instant... peut-être encore un souci de compatibilité.

```
* (lambda(x) x)
#<Interpreted Function (LAMBDA (X) X) 48032FE1>
* #'(lambda(x) x)
#<Interpreted Function (LAMBDA (X) X) 48033FB9>
* #'(lambda(x) x)
#<Interpreted Function (LAMBDA (X) X) 48034F91>
```

Par contre, elles sont souvent utilisées pour retourner une fonction en résultat d'une fonction. On parle de **fonctionnelle**. Des exemples seront vus dans le chapitre des foncitonnelles.

Chapitre 6

Fonctionnelles

Une fonctionnelle (en anglais *higher-order function*) est une fonction admettant des arguments et/ou un résultat qui sont des fonctions.

6.1 Fonctions en arguments

En lisp, il est très courant de paramétrer des fonctions par des fonctions. Voici un exemple introductif, vu dans le chapitre précédent.

```
* (defun of-zero (f)
       (funcall f 0))
OF-ZERO
* (defun id (x) x))
ID
* (of-zero #'id)
0
```

Remarque. Il est important de remarquer que le paramètre de `of-zero` doit être une fonction à un paramètre numérique. Bien qu'il n'y ait pas de déclarations, le programmeur doit maîtriser les types de ses objets. Il s'agit de **typage dynamique**. C'est sa position fonctionnelle dans l'expression **(f 0)** qui nous indique le type de **f**.

6.1.1 Fonctions de bibliothèque

Quelques fonctions avec arguments fonctionnels :

– **member** (member-if, member-if-not : plus tard) : élément, liste (test eq par défaut, mots clef :test pour changer).

```
* (member '(1) '(2 (1) 3) :test #'equal)
((1) 3)
* ((member-if #'evenp '(1 2 3))
(2 3)
```

– **remove** : même principe.

```
* (remove '(1) '( 2 (1) 1 3 (1)) :test #'equal)
(2 1 3)
```

– **assoc** : même principe.

6.1.2 Fonctionnelles sur les listes

Mapcar

Cette fonction prend une fonction et une liste en arguments.
– Cas d'une fonction unaire
(mapcar f '(a1 a2 ... an)) ⟶ (list (f a1) (f a2) ... (f an))
– Cas d'une fonction n-aire
(mapcar f '(a1 ...) '(b1 ...) ...) ⟶ (list (f a1 ...) (f a2 ...) ... (f an ...))

```
* (mapcar #'square '(1 2 3 4))
(1 4 9 16)
* (mapcar #'+ '(1 2 3) '(3 2 1) '(2 2 2))
(6 6 6)
* (mapcar #'cons '(1 2 3) '(a b c))
((1 . A) (2 . B) (3 . C))
* (mapcar #'car '((a1 a2 a3 a4) (b1 b2 b3 b4) (c1 c2 c3 c4)))
(A1 B1 C1)
* (mapcar #'cdr '((a1 a2 a3 a4) (b1 b2 b3 b4) (c1 c2 c3 c4)))
((A2 A3 A4) (B2 B3 B4) (C2 C3 C4))
```

Remarque 1. Si les listes n'ont pas le même nombre d'arguments, dans le cas d'opérations n-aires, c'est la plus courte qui donne la taille du résultat.

Remarque 2. Le premier argument doit impérativement être une fonction et non une macro. Les formes spéciales sont donc exclues : **or**, **and**. Il

est toutefois possible de définir une fonction ayant le même résultat. Mais son foncitonnement est évidemment différent, puisqu'elle évalue TOUS ses arguments.

```
* (defun ou(x y) (or x y))
OU
* (mapcar #'ou '(t t nil t) '(t nil nil t))
(T T NIL T)
```

Remarque 3. L'écriture de **ou** n-aire nécessite l'utilisation le la fonction **apply**. Voir ci-après.

Apply

Cette fonction réalise l'application d'une fonction à une liste d'arguments. Ce mécanisme est utile pour l'écriture de fonctions à nombre d'arguments variable (puisque les arguments sont justement dans une liste).

```
(apply fonction liste-arguments)
```

Exemple 1.

```
* (apply #'+ '(1 2 3))
6
```

Exemple 2. Moyenne des carrés d'un nombre quelconque d'arguments numériques.

```
(defun moyenne-carre (&rest l)
  (/ (apply #'+ (mapcar #'* l l)) (length l)))

* (moyenne-carre 1 2 3)
14/3
```

Exemple 3. La fonction **ou** n-aire

```
(defun ou (&rest l)
  (cond ((null l) t)
        ((car l) t)
        (t (apply #'ou (cdr l)))))
```

```
* (ou nil t nil (print nil))
NIL
T
```

Remarque. Même si la fonction a un nombre d'arguments variable, et même si elle ne parcourt pas toute la liste (elle s'arrête au premier argument valant vrai), les arguments sont tout de même tous évalués au moment de l'application.

Filtrage

Beaucoup de problèmes nécessitent de filtrer des listes, c'est-à-dire, de constituer une sous-liste à partir d'une liste **L** contenant un sous-ensemble des éléments de **L** ordonnés de la même façon.

La fonction suivante (non prédéfinie) est utile pour résoudre ce problème. Elle réalise la concaténation des valeurs de retour de chaque application par mapcar.

```
(defun append-map (f l)
  (apply #'append (mapcar f l)))
```

Exemple 1. Voici un exemple d'application de cette fonction.

```
* (append-map #'(lambda (x) (if (evenp x) (list x) '()))
              '(1 2 3 4 5))
(2 4)
```

Exemple 2. Voici une fonction permettant de généraliser le principe précedent en acceptant en arguments : le test **f** et la liste **l**.

```
(defun filtre (f l)
  (append-map #'(lambda (x)
                  (if (funcall f x)
                      (list x)
                    '()))
              l))
```

```
* (filtre #'evenp '(1 2 3 4 5 6))
(2 4 6)
```

6.2 Fonctions en retour de fonctions

En lisp, il est possible de créer des fonctions dynamiquement. Ce mécanisme est différent de celui des pointeurs de fonctions dans le langage C, car, en lisp, il est possible de créer de nouvelles fonctions, alors qu'en langage C, elles doivent être écrites au préalable.

6.2.1 Un premier exemple : la composition de fonctions

$$f : A \longrightarrow B$$

$$g : B \longrightarrow C$$

La composée de f par g est la fonction $h : A \longrightarrow C$ telle que $h(x) = g(f(x))$. Elle est notée $h = gof$.

```
(defun o(g f)
  (lambda(x)
    (funcall g (funcall f x))))
```

Exemple .

```
* (o #'carre #'1-)
#<Interpreted Function "LAMBDA (G F)" 4809E671>
* (funcall (o #'carre #'1-) 2)
1
* (defparameter carre-1 (o #'carre #'1-))
CARRE-1
* (funcall carre-1 2)
1
```

Remarque 1. Pour nommer une telle fonction, on est obligé d'utiliser le champ "valeur" d'un symbole en utilisant la forme `defparameter`. Ce qui nous oblige à utilser `funcall` pour l'application.

Remarque 2. C'est grâce à une représentation interne adéquate des fonctions que ce mécanisme marche : la **fermeture**. C'est un couple associant l'environnement lexical de la fonction avec son code. Exemple sous emacs-lisp (à faire en TD). Lors de l'application

```
(funcall (o #'carre #'1-) 2)
```

On obtient l'erreur suivante. Pourquoi ? ?

```
debug(error (void-variable f))
(funcall f (funcall g x))
(lambda (x) (funcall f (funcall g x)))(2)
funcall((lambda (x) (funcall f (funcall g x))) 2)
eval((funcall (o (function carre) (function 1-)) 2))
```

6.2.2 Un deuxième exemple : la curryfication

Soit une fonction

$$f \ : A \times B \longrightarrow C$$

la curryfication lui associe la fonction suivante ayant :

$$f^c \ : A \longrightarrow (B \longrightarrow C)$$

qui a pour résultat une fonction allant de B dans C et telle que $\forall x \in A, f(x,y) = (f^c(x))(y)$

Cette fonction s'écrit de la façon suivante en lisp :

```
(defun curry (f)
  (lambda(x)
    (lambda(y)
      (funcall f x y))))
```

Exemple. On va curryfier la fonction **filtre** vue dans la section précédente pour fixer le premier paramètre foncitonnel au test pair (evenp) et pour pouvoir ainsi l'appliquer à différentes listes.

```
* (defparameter filtre-f (curry #'filtre))
FILTRE-F
* (defparameter filter-evenp (funcall filtre-f #'evenp))
FILTER-EVENP
* (funcall filter-evenp '(1 2 3 4 5 6))
(2 4 6)
```

6.2.3 Exemples avec récursivité

Cette sous-section est destinée à bien différencier le code de la foinctionnelle de celui de la fonction résultat.

Exemple de fonctionnelle récursive : composition d'une liste de fonctions

Attention à bien mettre la récursivité dans la fonctionnelle et non dans le résultat.

```
(defun o (f &rest l)
  (if (null l)
      f
    (apply #'o (lambda(x) (funcall (car l) (funcall f x)))
             (cdr l))))
```

Développement de l'application suivante :

```
(funcall (o #'1+ #'carre #'1-) 2)
f = #'1+ - l = (#'carre #'1-)
f = (lambda(x) (funcall #'carre (funcall #'1+ x))) - l = (#'1-)
f = (lambda(x)
       (funcall #'1-
                (funcall (lambda(x) (funcall #'carre
                                             (funcall #'1+ x)))
                         x)))
```

Remarque. La liste de fonctions est appliquée à l'envers. Il est donc nécessaire d'inverser cette liste avant l'application.

```
(defun rond (&rest l)
  (labels ((o (f &rest l)
              (if (null l)
                  f
                (apply #'o
                       (lambda(x)
                          (funcall (car l) (funcall f x)))
                       (cdr l)))))
           (if (null l)
               #'(lambda (x) x)
```

```
            (apply #'o l))))
```

Exemple de mauvaises écritures.

– Tout le calcul est gelé. Il s'éxecutera entièrement au moment de l'application de la fonction résultat.

```
(defun bad (&rest l)
  (lambda(x)
    (if ... )))
```

– Une partie est gelée et s'éxecutera au moment de l'application de la fonction résultat.

```
(defun bad (&rest l)
  (if (null l)
      #'id
    (lambda (x)
      (funcall (car l) (funcall (apply #'bad (cdr l)) x)))))
```

Exercice. Iterer une fonction n fois.

Exemple de résultat récursif

À faire en TD. Soit la fonctionnelle généralisant le calcul de la racine carrée (voir feuille de td).

Chapitre 7

Structures impératives

Ce chapitre est simplement destiné à montrer que des structures impératives existent aussi dans le langage lisp. L'objectif n'est pas de présenter la totalité des structures, mais plutôt la cohabitation avec ce qui précède. Ces expressions rnvoient aussi un résultat comme les autres, mais sont plutôt utilisées pour les effets de bord qu'elles produisent. Les points soulevés sont : les références, le partage de la mémoire.

7.1 Modifications d'objets

7.1.1 setf

```
(setf place expression)
```

Retour : valeur de l'expression affectée.

7.1.2 Modification de paires pointées

```
(defparameter p (cons 1 2))
(setf (car p) 3)
```

7.2 Fonctions modifiant leurs paramètres

7.2.1 Modifications de paramètres

– Impossible de modifier l'adresse d'un paramètre, car c'est sa référence (sa valeur) qui est transmise et non son adresse.

```
*(defun incrementer (x)
     (setf x (1+ x)))
INCREMENTER
* (incrementer 1)
* 2
* (defparameter a 1)
A
* (incrementer a)
2
* a
1
```

– Les seules adresses modifiables sont celles qui sont accessibles à partir de la valeur des paramètres.

```
* (defun set-car (p e)
      (setf (car p) e))
SET-CAR
* (defparameter a (cons 1 2))
A
* (set-car a 'a)
(A . 2)
* a
(A . 2)
```

7.2.2 Nconc

Cette fonction réalise la concaténation de n listes sans recopier les $n-1$ premières.

```
* (defparameter l1 '(1 2 3))
L1
* (defparameter l2 '(a b))
L2
```

```
* (defparameter l (nconc l1 l2))
L
* l
(1 2 3 A B)
* l1
(1 2 3 A B)
* l2
(A B)
```

7.2.3 Fonctions d'application

- **mapc** : elle applique une fonciton à chaque élément d'une liste passée en paramètre. Le résultat est son paramètre. La fonction se doit donc de modifier les éléments de la liste.

  ```
  * (mapc (lambda(x) (setf x (1+ x))) '(1 2 3))
  (1 2 3)
  * (mapc (lambda(x) (setf (car x) (1+ (car x)))) '((1 . 2) (2
  . 3)))
  ((2 . 2) (3 . 3))
  ```

- **mapcan** (mapc + nconc), **mapl** (application aux sous-listes), **maplist** et **mapcon** (mapl + nconc) : à voir seuls.

7.2.4 Partage de la mémoire et affectation

Donner un exemple.

7.2.5 Listes circulaires

```
* (defparameter l '(1 2 3))
L
* (append l l)
(1 2 3 1 2 3)
* l
(1 2 3)
* (nconc l l)
(1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1
 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1
```

```
 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1
* l
(1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1
 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1
 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1
```

7.2.6 Retour sur les variables spéciales

C'est l'occasion de revenir sur les variables spéciales.

- `*print-circle*` : affichage des boucles
- `*print-length*` : affichage du préfixe de la liste de cette longueur.
- `*print-lines*` : affichage de ce nombre de lignes au maximum.

```
  * (defparameter l '(1 2))
  L
  * (setf *print-circle* t)  ; nécessaire pour l'affichage
  T                          ; du résultat de l'expression suivante
  * (setf (cdr l) l)
  #1=(1 . #1#)
* (let ((*print-circle* nil)
        (*print-length* 10))
     (print l))
(1 1 1 1 1 1 1 1 1 1 ...)
(1 . #1=(1 . #1#))
* l
(1 . #1=(1 . #1#))
* (let ((*print-circle* nil)
       (*print-lines* 10))
    (print l))
(1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
(1 . #1=(1 . #1#))
* (defparameter ll (cons l l))
LL
* ll
(#1=(1 . #2=(1 . #2#)) . #1#)
```

Modification des variables spéciales. Il est préférable de les modifier au moyen d'un let. En effet, en cas de sortie sur erreur dans le bloc du let, les liaisons du let sont détruites et la valeur intiale de la variable est restaurée. Ce que ne permet pas l'écriture suivante :

```
* (setf *old-p-value* *p*)
* (setf *p* new-value)
* (appel ...)
* (setf *p* *old-p-value*)
```

7.3 Tableaux et structures

7.3.1 Tableaux

- Dimension : arbitraire
- Création : **make-array** liste, mots-clefs :element-type, :initial-element.
- Accès aux éléments : **aref** tableau, indices.

Exemples. Le Robert p134 et p136 tableaux de caractères.

Remarque. La notation utilisée par **print** n'est pas normalisée pour une utilisation avec **read**.

7.3.2 Structures

- Création : (**defstruct** nom champ1 champ2 ...)
- Accès aux éléments : (**nom-champ1** objet)

7.3.3 Tables de hachage

- Création : (**make-hash-table** :test eq). Le test d'égalité doit être un des tests primitifs de lisp.
- Accès aux éléments : (**gethash** clef table) permet d'accéder et de modifier avec **setf**.
- Destruction : (**rem-hash** clef table)

7.4 Structures de contrôle

7.4.1 Blocs

Certaines expressions pouvant effectuer des effets de bord, il devient possible de les mettre en séquence. Les formes **defun**, **lambda**, **let**, **flet** et **labels** acceptent une séquence d'expressions dans leur corps.

En d'autres situations, en particulier dans une forme conditionnelle, on peut utiliser la forme **progn** qui permet de mettre en séquence des expressions.

Dans tous ces cas, une séquence d'expressions est évaluée de la façon suivante

- Chaque expression est évaluée dans son ordre d'apparition.
- Le résultat de l'évaluation de la séquence est celui de la dernière.
- Les valeurs des expressions précédentes sont perdues.

Remarque. Il est particulièrement important que l'ordre d'évaluation des expressions soit connu, puisque chacune d'elles effectue un effet de bord.

Remarque. La forme **prog1** diffère de **progn** par le fait qu'elle renvoie le résutat de sa première expression. Elle doit au moins en comporter une.

7.4.2 Conditionnel

La forme **case** ressemble à la forme **cond**.

```
(case objet clause1 clause2 ... clausen)
```

Chaque clause est constituée d'une liste d'objets, ou d'un seul objet, suivie d'une séquence d'expressions. Le premier paramètre est évalué et comparé aux éléments des listes des clauses, dans l'ordre d'apparition des clauses.

Aussitôt qu'une liste le contient, les expressions de la clause concernée sont évaluées dans l'ordre. Le résultat est celui de la dernière expression évaluée. Si aucune liste ne contient le premier paramètre, le résultat est NIL.

La dernière clause peut comporter **otherwise** ou **t**, pour indiquer le complémentaire des autres clauses. Cette clause est facultative.

Le prédicat d'égalité utilisé est **eql**.

7.4.3 Itération

Les formes **do** (do*, dotimes, dolist) sont les plus anciennes. Aujourd'hui, ce sont les boucles **loop** qui sont utilisées par les programmeurs. Une boucle **loop** a la forme suivante.

Formes do

```
(do (varclause1
     varclause2
     ...
     varclausen)
     (terminaison)
  expr1
  expr2
  ...
  exprn)
```

Avec
- **varclause = (var [init] [iter])**
- **terminaison = (test finexpr1 finexpr2 ... finexprk)**

Exemple.

```
(do ((i 0 (1+ i)))
    ((> i 5) (print "end"))
    (print i))
```

- do* : séquentiel comme le let*
- dotimes : itérations un nombre prédéterminé de fois.
- dolist : itérations sur les éléments d'une liste.

Formes loop

– Le prologue contient les initialisations des variables et les expressions à évaluer avant l'éxecution de la boucle.
– Le corps comprend les expressions à évaluer à chaque tour de boucle.
– L'épilogue comprend les expressions à évaluer aprés les itérations.

Ces formes sont très puissantes. Nous en présentons un sous-ensemble seulement.

– Répétition

```
(loop repeat n
      do (print i))
```

– Itération sur des listes : par défaut le passage au suivant se fait avec la fonction **cdr**.

```
(loop for i in '(1 2 3 4) [by ~'cdr]
      do (print i))
```

– Itération sur des vecteurs

```
(loop for i across #(5 2 6 4)
      do (print i))
```

– Boucle pour

```
(loop for i from 1 to 3
      do (print i))
```

– Boucle infinie

```
(loop do (print 1) (print 3))
```

7.5 Fermetures et affectation

Au moyen de l'affectation, on peut utiliser une fermeture pour représenter un état, évoluant à chaque appel.

7.5.1 Générateurs

– Générateur d'entiers

```
(let ((i 0))
  (defun gen-entier ()
     (setf i (1+ i))))
```

Exemple de session

```
* (gen-entier)
1
* (gen-entier)
2
* (gen-entier)
3
```

– Suite de Fibonacci

```
(let* ((u0 0)
       (u1 1)
       (u2 (+ u0 u1)))
  (defun gen-fib ()
    (setf u0 u1)
    (setf u1 u2)
    (setf u2 (+ u0 u1))
    u0))
```

Exemple de session

```
* (gen-fib)
1
* (gen-fib)
1
* (gen-fib)
2
* (gen-fib)
5
```

7.5.2 Mémo-fonctions

La technique des mémo-fonctions est utilisée pour optimiser le calcul des fonctions, en mémorisant des résultats intermédiaires. Par exemple, le calcul de la série de fibonacci.

```
(let ((memo-table '((1 . 1) (2 . 1))))
  (defun memo-fib (n)
    (cond ((cdr (assoc n memo-table)))
          (t (let ((new-value (+ (memo-fib (1- n))
                                 (memo-fib (- n 2)))))
               (acons n new-value memo-table)
```

```
                new-value)))))
```

Exemple de session

```
* (memo-fib 5)
5
* (memo-fib 8)
21
```

Remarque. En Scheme, cette technique est plus compliquée à mettre en oeuvre, puisqu'elle nécessite d'utiliser une fonctionnelle pour construire la fermeture.

Chapitre 8

Macroexpansions

8.1 Rappels sur l'évaluation et l'application

8.1.1 Évaluation applicative

Cette forme d'évaluation est utilisée pour toutes les fonctions construites avec des **lambda**, **defun**, **flet** et **labels**. C'est celle qui est mise en oeuvre dans la plupart des langages de programmation, en particulier impératifs (C, Java).

`(eval` **o ed el** `espace).`

Avec
- **o** : forme à évaluer
- **ed** : environnement dynamique
- **el** : environnement lexical
- espace : valeurs ou fonctions

Précisions sur l'algorithme de l'évaluation
- Si l'objet **o** est autoévaluant, renvoyer **o**
- Si **o** est un symbole, alors
  - Si espace de valeurs alors si **o** est spécial, rechercher une liaison dans **ed**, sinon dans **el**
  - Si espace de fonctions alors rechercher une liaison fonctionnelle dans **el**
- Si **o** est une liste

  - Appeler (eval **a ed el** valeurs), pour tout élément **a** de (cdr **o**). Soit **v** la liste des résultats.
  - Appeler (eval (car **o**) **ed el** fonction). Soit **f'** la fermeture résultat.
- Appeler (**apply f'**) **v el ed**)

```
(apply  f  v  el  ed)
```

Avec
- **f** : fermeture de la fonction à appliquer
- **v** : liste des valeurs des arguments
- **ed** : environnement dynamique
- **el** : environnement lexical

Précisions sur l'algorithme d'application
- Soient **e** l'environnement lexical de **f**, **lf** la liste des paramètres formels, et **c** le corps de la fermeture.
- Construire l'environnement lexical local **el-local** constitué des liaisons entre les paramètres formels non spéciaux de **lf** et les valeurs correspondantes dans **v**.
- Construire l'environnement dynamique local **ed-local** constitué des liaisons entre les paramètres formels spéciaux de **lf** et les valeurs correspondantes dans **v**.
- Pour la suite d'expressions **expr** du corps **c** de **f**, faire (eval **expr** (cons **ed-local ed**) (cons **el-local e**) valeurs).
- Renvoyer le résultat de l'évaluation de la dernière expression de **c**.

8.1.2 Évaluation paresseuse

L'évaluation paresseuse ou par nécessité consiste à retarder l'évaluation des paramètres jusqu'au moment de leur utilisation. Éventuellement, certains paramètres ne sont pas évalués dans certains cas. Ce mécanisme est nécessaire pour implémenter les conditionnelles et les boucles.

En lisp, les macroexpansions permettent d'écrire ces formes dites spéciales.

8.1.3 Remplacement textuel

En C, les macro-fonctions fonctionnent selon un mécanisme de remplacement textuel. Le corps de la macro est une suite de caractères. Lors de la substitution, chaque occurrence de paramètre est remplacée par la chaîne

de caractère donnée à l'appel. Ainsi, la structure syntaxique n'est pas prise en compte. Pour éviter ces pièges syntaxiques, il faut respecter des règles d'écriture des macros (paramètres entre parenthèses, corps entre parenthèses).

8.2 Écriture de macroexpansions en lisp

8.2.1 Citation

- Rappel : quote
- Backquote : ‘
- Virgule : ,
- Arobase : @

8.2.2 Définition

```
(defmacro nom-macro (p1 p2 ... pn)
   corps)
```

Exemple

```
(defmacro ifn (test e1 e2)
   ‘(if (not ,test) ,e1 ,e2))
```

8.2.3 Macroexpansion

```
(macroexpand-1 'appel)
```

Exemple 1.

```
* (macroexpand-1 '(ifn (= 1 2) (+ 2 1) (* 2 2)))
(IF (NOT (= 1 2)) (+ 2 1) (* 2 2))
```

Remarque. On constate que les arguments n'ont pas été évalués.

Exemple 2 : nombre d'arguments variable.

```
(defmacro while (test &body body)
  ‘(do ()
       ((not ,test))
```

```
      ,@body))

* (macroexpand-1 '(while (< *i* 3) (print *i*) (incf *i*)))
(DO () ((NOT (< *I* 3))) (PRINT *I*) (INCF *I*))
* (while (< *i* 3) (print *i*) (incf *i*))
1
2
```

8.2.4 Fonctionnement

- Constitution de l'environnement local : des liaisons sont établies entre les paramètres formels et les paramètres d'appel **non évalués**.
- Macroexpansion : le corps de la macro est évalué dans cet environnement, augmenté de l'environnement lexical de la macro.
- Le résultat de la macroexpansion est évalué dans l'environnement d'appel.

8.3 Problèmes d'utilisation des macros

8.3.1 Captures de noms

```
(defmacro echanger (x y)
  `(let ((tmp ,x))
     (setf ,x ,y)
     (setf ,y tmp)))

* (defparameter *i* 3)
* (defparameter *j* 5)
* (echanger *i* *j*)
* *i*
5
* *j*
3
```

Un problème de capture de nom survient quand un des paramètres a le même nom que la variable temporaire.

```
* (defparameter tmp 3)
* (defparameter *j* 5)
```

```
* (echanger tmp *j*)
* (macroexpand-1 '(echanger tmp *j*))
(LET ((TMP TMP))
  (SETF TMP *J*)
  (SETF *J* TMP))
* *j*
5
* tmp
5
```

Remarque. Ce problème est général aux macroexpansions et se retrouve aussi dans le langage C.

```
#define ECHANGER(x,y) {int tmp=x; x=y; y=tmp;}

int i=3;
int tmp=5;

ECHANGER(tmp,i);
```

8.3.2 Renommage

Pour éviter la capture de nom, il faut employer des noms de variables temporaires qu'aucun utilisateur ne pourra imaginer. La forme **gensym** permet d'engendrer des noms de symboles nouveaux à chaque appel.

```
* (gensym)
#:G879
```

Exemple.

```
(defmacro echanger (x y)
  (let ((tmp (gensym)))
    `(let ((,tmp ,x))
       (setf ,x ,y)
       (setf ,y ,tmp))))
```

```
* (macroexpand-1 '(echanger tmp *I*))
(LET ((#:G893 TMP))
```

```
(SETF TMP *I*)
(SETF *I* #:G893))
```

8.3.3 Évaluations multiples

Ce problème est aussi rencontré dans le langage C.

```
(defmacro carre (x)
  `(* ,x ,x))

* (defparameter x 0)
* (macroexpand-1 '(carre (incf x)))
(* (INCF X) (INCF X))
```

Pour remédier à ce problème, il faut créer des variables temporaires destinées à recevoir les valeurs des expressions fournies dans les paramètres.

```
(defmacro carre (x)
  (let ((tmp (gensym)))
    `(let ((,tmp ,x))
       (* ,tmp ,tmp))))
```

8.4 Conclusion

- Les macros ne sont pas des fonctions, elles ne sont donc pas utilisables avec les fonctionnelles (mapcar, apply).
- Elles peuvent être utilisées pour écrire de nouvelles formes spéciales.
- Pour éviter la capoture de variables, kil faut utiliser la forme **gensym** chaque fois qu'une variabloe temporaire est nécessaire.
- Pour éviter l'évaljuation multiple, il faut lier le paramètre destiné à apparaître plusieurs fois dans le corps de la macro, avec une variable temporaire.
- Ces difficultés impliquent que les fonctions sont conseillées chaque fois que leur utilisation est possible.

Table des figures