Programmation Fonctionnelle

Dans ce cours nous allons présenter le style de programmation fonctionnelle.
Le λ-calcul enrichi (pour des raisons d'efficacités) par quelques fonctions usuelles prédéfinies (+, -, *, IF, ...) est alors vu comme un langage pivot (un genre d'assembleur pour les langages fonctionnels de haut niveau).
La machine à réduction (l'interpréteur du λ-calcul enrichi) implémente quelques règles de réduction simples (β-réduction, règles relatives aux fonctions prédéfinies, ...) permettant d'évaluer ainsi les expressions en entrée.
La récursivité dans les langages de haut niveau est mise en œuvre à l'aide des ces réductions (combinateurs à point fixe), ce qui est beaucoup plus simple que les mécanismes utilisés dans les langages procéduraux (gestion de la pile des zones des données).
Nous présenterons aussi le noyau fonctionnel du langage Lisp et verrons comment manipuler des fonctions d'ordre supérieur.
Dans la dernière partie du chapitre, nous montrerons comment utiliser le théorème du point fixe dans les preuves de programmes fonctionnels.

1. Introduction

La programmation fonctionnelle se réduit à l'utilisation d'expressions fonctionnelles pour décrire les différents traitements. La principale construction est l'application de fonctions (Programmation Applicative). Ce style de programmation fait partie du paradigme 'non procédural'.

Exemple d'expression fonctionnelle:
f(g(x,a, f(h(a),x)),b) avec f,g et h des fonctions, x une variable, a et b des constantes.

Caractéristiques des langages purement fonctionnels:

On retrouve principalement (les 2 premiers points ci-dessous) ce qui caractérise les langages non procéduraux :
- Absence d'affectation: pas d'instruction qui modifient les variables
- Absence de contrôle du séquencement : pas de GOTO, EXIT, Boucle, ...
- Calcul basé uniquement sur les fonctions.

Schéma de programmes fonctionnels

Le corps d'une fonction se résume à une seule expression fonctionnelle.
Expression = cste / var / Appel de fonction ayant comme arguments des expressions.

Exemple: Fact(n) : Si (Egal(n,0) , 1 , Mult(n , Fact (Pred (n))))

La construction Si ... Alors ... Sinon est considérée comme une fonction (non stricte) à 3 arguments :
Si (exp1 , exp2 , exp3) = Si (exp1) Alors retourner exp2 Sinon retourner exp3 Fsi
exp1 est évaluée, si sa valeur est vrai, exp2 est évaluée sinon c'est exp3 qui sera évaluée.
Le résultat de l'expression 'Si' est celui de la dernière expression évaluée.

On peut donc associer une réduction prédéfinie permettant d'évaluer un 'Si… alors … sinon' :
Si(vrai, e2, e3) → e2 et Si(faux, e2, e3) → e3

Une fonction est dite non stricte, si son évaluation ne nécessite pas l'évaluation de tous ses arguments.
Parmi les langages fonctionnels existant on peut citer :
Lisp, ISWIM, FP (langage sans variable, uniquement des combinateurs), Hope, ML, Miranda, …
La plupart ne sont pas purement fonctionnels, mais disposent d'un noyau purement fonctionnel.

2. Le λ-calcul

C'est une notation particulière pour exprimer des fonctions.
D'un point de vu informatique, si on enrichit le λ-calcul par les opérations prédéfinies usuelles (+, -, *, Si. alors...sinon, ...), il peut alors être considéré comme un langage de programmation purement fonctionnel de bas niveau. C'est ce que nous allons voir dans le reste de ce chapitre.
Pour chacune de ces opérations 'prédéfinies' ainsi que les types de données manipulées (comme les entiers par exemple), il existe des représentations en λ-calcul pur.

Les expressions du λ-calcul (λ-expressions) sont généralement écrites en préfixé.

Exemple :

(+ 3 4) est une λ-expression qui peut être réduite en une λ-expression plus simple : 7

(+ (* 5 6) (* 8 3)) est aussi une λ-expression, qui contient deux radicaux (expressions pouvant être réduites). Donc elle peut se réduire en (+ 30 (* 8 3)) ou bien en (+ (* 5 6) 24). Chacune de ses deux nouvelles expressions contient un radical, donc peut se réduire en une expression plus simple, ...etc.

Application de fonction - « curryfication »

(f x) application de la fonction f à l'argument x (en notation usuelle : f(x))
((g x) y) application de la fonction g à l'argument x, le résultat est une fonction qu'on applique sur l'argument y (en notation usuelle : g(x,y)).

Exemple :

((+ 3) 4) : la fonction '+' appliquée à l'argument 3, le résultat est une fonction qu'on applique à 4.
L'expression (+ 3) désigne donc une fonction qui rajoute 3 à son argument.

Définition de nouvelles fonctions (λ-abstraction)

En λ-calcul, on peut définir des fonctions par des λ-expressions particulières (appelée λ-abstraction).

Si E est une λ-expression quelconque, alors : λx. E est une λ-expression qui définie une fonction ayant x comme paramètre formel et E comme corps.

Exemple :
(λx. (+ x 1)) est une expression qui définie la fonction qui incrémente son argument.

Syntaxe générale d'une λ-expression

<λ-exp> ::= <Cste> / <Var> / <λ-exp> <λ-exp> / λ<Var> . <λ-exp>

Une λ-expression est une constante (a, b, c, ...) ou une variable (x, y, z, ...) ou une application d'un λ-expression sur une autre λ-expression ou alors c'est une définition d'une nouvelle fonction anonyme (λ-abstraction).

Conventions d'écriture des λ-expressions

Cas d'une suite d'applications : ((((A B) C) D) E)
On peut omettre les parenthèses internes distribuées à gauche, ce qui donne : (A B C D E)

Exemple : ((+ 3) 4) = (+ 3 4)

Cas d'une suite d'abstractions : (λx.(λy.(λz.(...E))))
On peut omettre les parenthèses internes distribuées à droite, ce qui donne : (λx.λy.λz. ...E)

Exemple : (λx.(λy.(λz.(+ x (- y z))))) = (λx.λy.λz. + x (- y z))

Variables libres et variables liées

Dans une λ-expression, toute occurrence d'une variable qui se trouve dans le champ d'application d'un λ est dite liée. Les occurrences qui ne sont pas liées, sont dites libres.
Exemple : dans l'expression (λx. (+ x y) 4) x, x a 2 occurrences et y a une seule occurrence.

| | |____ une occurrence libre de x
| |_____ une occurrence libre de y
|___une occurrence liée de x

Substitution

[N/x] M : remplacer dans l'expression M, toutes les occurrences libres de x par l'expression N.
Formellement, la substitution [N/x] M est définie par :

(i) [N/x] x = N
(ii) [N/x] y = y
(iii)[N/x] (A B) = ([N/x] A [N/x] B)
(iv)[N/x] (λx. E) = (λx. E)
(v) [N/x] (λy. E) = (λy. [N/x] E) si y n'apparaît pas libre dans N
= (λz. [N/x] [z/y] E) sinon (avec z qui n'apparaît libre ni dans E ni dans N).

Exemple : [y/x](λy. x) = (λz. x) et non (λy. y)

Règles de conversion

Une conversion est soit une réduction (d'une expression complexe vers une expression simple), soit une abstraction (d'une expression simple vers une expression plus complexe).

E1 → E2 veut dire qu'il existe une réduction entre E1 et E2
E1 ← E2 veut dire qu'il existe une abstraction entre E1 et E2
E1 ↔ E2 veut dire qu'il existe une conversion entre E1 et E2 (un genre d'équivalence)

Bêta conversion

Cette règle exprime l'équivalence entre une application de la forme ((λx. E) F) et son expression réduite [F/x] E.

((λx. E) F) $\leftrightarrow_{(\beta)}$ [F/x] E β-conversion
((λx. E) F) $\rightarrow_{(\beta)}$ [F/x] E β-réduction
((λx. E) F) $\leftarrow_{(\beta)}$ [F/x] E β-abstraction

Remarque : (λx. (λy.E)) = (λx. λy.E) = (λxy.E)

Par exemple l'expression ((λx. + x 1) 4) est bêta-convertible en (+ 4 1) :
((λx. + x 1) 4) $\leftrightarrow_{(\beta)}$ (+ 4 1)

Voici quelques exemples de réductions Bêta et celles d'opérations prédéfinies (+, -, ...) :
(λx. + x x) 4 $\rightarrow_{(\beta)}$ (+ 4 4) $\rightarrow_{(+)}$ 8
(λx. 3) 4 $\rightarrow_{(\beta)}$ 3
(λx. (λy. - y x)) 4 5 $\rightarrow_{(\beta)}$ (λy. - y 4) 5 $\rightarrow_{(\beta)}$ (- 5 4) $\rightarrow_{(-)}$ 1
(λx. x 3) (λy. + y 1) $\rightarrow_{(\beta)}$ (λy. + y 1) 3 $\rightarrow_{(\beta)}$ (+ 3 1) $\rightarrow_{(+)}$ 4
(λx. (λx. + (- x 1)) x 3) 9 $\rightarrow_{(\beta)}$ ((λx. + (- x 1)) 9 3) $\rightarrow_{(\beta)}$ (+ (- 9 1)) 3 $\rightarrow_{(-)}$ (+ 8) 3 $\rightarrow_{(+)}$ 11
(λx. λy. + x ((λx. - x 3) y)) 5 6 $\rightarrow_{(\beta)}$ (λy. + 5 ((λx. - x 3) y)) 6 $\rightarrow_{(\beta)}$ (+ 5 ((λx. - x 3) 6))
$\rightarrow_{(\beta)}$ (+ 5 ((- 6 3))) $\rightarrow_{(-)}$ (+ 5 3) $\rightarrow_{(+)}$ 8

Alpha conversion

Cette règle exprime l'équivalence entre expressions qui ne diffèrent que par le nom du paramètre formel :
(λx. E) $\leftrightarrow_{(\alpha)}$ (λy. [y/x]E) α-conversion

Par exemple :
(λx. + x 1) $\leftrightarrow_{(\alpha)}$ (λy. + y 1)

Un terme X est dit 'congruent' à un terme Y, ssi Y est le résultat de l'application d'un ou de plusieurs changements de noms de variables liées.

La congruence est une relation d'équivalence car l'alpha-conversion est réflexive, symétrique et transitive.

Êta conversion

C'est une simplification de certaines expressions de la forme (λx. E x) en E, à condition que x n'apparaît pas libre dans E :
(λx. E x) $\leftrightarrow_{(\eta)}$ E

Par exemple :
(λx. + 1 x) $\leftrightarrow_{(\eta)}$ (+ 1)

Formes Normales

Une expression E est en forme normale, si elle ne contient aucun radical (elle n'est pas réductible).

Une expression E possède une forme normale, si elle est réductible directement ou indirectement (par transitivité) vers une expression en forme normale.

Par exemple, l'expression (z v) représente la forme normale de l'expression (λx. (λy. y x) z) v

Il existe des expressions réductibles mais qui ne possèdent pas de formes normales, par exemple :
(λx. x x y) (λx. x x y) → (λx. x x y) (λx. x x y) y → (λx. x x y) (λx. x x y) y y → …

En programmation fonctionnelle, cela représente des programmes qui boucles à l'infini.

Ordres de réduction

Quand une expression contient plusieurs radicaux (sous-expressions réductibles), il se pose alors le problème du choix du radical à réduire en premier. Il existe plusieurs stratégies de réductions dont certaines risques de ne pas mener vers la forme normale pour des expressions qui pourtant en possède une.
Ordre normal : on commence à réduire d'abord le radical le plus externe, le plus à gauche ('appel par nom' dans les langages de programmation).
Appel par valeur : on commence à réduire d'abord le radical le plus interne, le plus à gauche.

Par exemple, avec l'appel par nom, quand on veut évaluer un appel de fonction F avec comme argument les expressions exp_1, exp_2,... exp_n, l'interpréteur commence d'abord par évaluer le corps de la fonction F, ensuite, s'il a besoin d'un argument (exp_i) durant l'évaluation du corps de F, celui-ci (exp_i) sera alors évalué.
Par contre dans l'appel par valeur, l'interpréteur commence toujours par évaluer tous les arguments d'abord (exp_1, exp_2, … exp_n) avant d'évaluer le corps de la fonction F.

L'appel par valeur est généralement plus efficace, car un argument n'est évalué qu'une seule fois, contrairement à l'appel par nom, où un même argument risque d'être évalué plusieurs fois s'il est utilisé à plusieurs endroit du corps de la fonction appelée.

L'appel par nom permet par contre de définir des fonctions non strictes, où on n'est pas obligé de connaître la valeur de tous les arguments pour pouvoir évaluer un appel.

Par exemple, pour la fonction définie (en notation usuelle) par :
f(x,y) = 1 si (x = 0)
f(x,y) = x+y sinon.
L'expression f(a, b/a), dans le cas où a=0 et b=3, pourra être évaluée correctement avec l'appel par nom (le résultat sera 1) mais pas avec l'appel par valeur. Ce dernier va tenter d'évaluer d'abord les deux arguments (0 et 3/0) avant l'expression globale.

Théorème de Church-Rosser I

Si U → X et U → Y alors il existe une expression Z telle que X → Z et Y → Z

On dit que l'expression A se réduit en l'expression B (A → B) ssi B est le résultat d'une série (éventuellement vide) de changements de nom de variables (alpha-conversion) et de bêta (ou Êta) réductions (contractions). La relation → est alors réflexive et transitive.

En conséquence on a le corollaire suivant :
Aucune λ-expression ne peut avoir deux formes normales distinctes (c-a-d qui ne soient alpha-convertibles).

En d'autres termes, si une expression possède une forme normale, alors celle-ci est unique.

Théorème de Church-Rosser II

Si E1 → E2 et E2 est en forme normale, alors il existe une suite de réductions de E1 à E2 selon l'ordre normal (appel par nom).

Ainsi pour mettre en œuvre un interpréteur de langage fonctionnel, en supposant que la forme intermédiaire d'un programme est une expression en λ-calcul, il suffit alors d'implémenter les règles de réductions (la bêta-réduction et les règles prédéfinies : +, - , si..alors..sinon , …) et d'utiliser une stratégie d'évaluation (appel par nom, par valeur, …) qui permet de trouver la forme normale. Cet interpréteur est appelé Machine à réductions.

Même l'évaluation des programmes récursifs peut s'exprimer très simplement sous forme de réductions, comme on va le voir dans section suivante.

λ-calcul et fonctions récursives

A travers un exemple, on va montrer comment le λ-calcul avec ses règles de réduction élémentaires, est suffisant pour évaluer (interpréter) des appels de fonctions récursives.

Prenons comme exemple, un programme récursif pour calculer n!

On pourrait l'écrire en lambda-calcul comme suit :
Fac = (λn. SI (= n 0) 1 (* n (Fac (- n 1)))) (I)
En réalisant une Bêta-abstraction sur le membre droit de l'équation, on aura alors :
Fac = (λf.(λn. SI (= n 0) 1 (* n (f (- n 1))))) Fac (II)
Posons H = (λf.(λn. SI (= n 0) 1 (* n (f (- n 1)))))
L'équation (II) sera alors de la forme : Fac = H Fac (III)

C'est une équation à point fixe dont la solution coïncide avec la fonction factorielle.

Ainsi, à tout programme récursif (purement fonctionnel), correspond une équation à point fixe (de la forme f = H f) dont la solution n'est autre que la fonction calculée par le programme H.

Supposons, pour le moment, qu'il existe une expression Y, permettant de trouver la solution d'une équation de la forme f = H f, quelque soit le programme H.

Dans l'exemple ci-dessus, (Y H) représentera alors la fonction factorielle.

Donc si on veut calculer, par exemple la valeur 3!, la machine à réduction devra alors chercher la forme normale de l'expression (Y H) 3.

Comme (Y H) est un point fixe (c-a-d une solution de l'équation f = H f), on a alors le droit d'écrire : (Y H) = H (Y H). Donc pour calculer (Y H), 3 il suffit de chercher la forme normale de (H (Y H)) 3.

On peut rajouter une nouvelle règle de réduction qui modélise cette dernière substitution :
Y H $\rightarrow_{(Y)}$ H (Y H) (la **règle de Y** est alors rajoutée à la machine à réduction)

Pour l'exemple précédent (calcul de 3!), voici alors le déroulement de la machine à réductions, jusqu'à trouver la forme normale (le résultat de l'expression initiale) : (H (Y H)) 3

= ((λf.(λn. SI (= n 0) 1 (* n (f (- n 1))))) (Y H)) 3	on remplace H par sa valeur
→ (λn. SI (= n 0) 1 (* n ((Y H) (- n 1)))) 3	bêta-reduction sur f
→ SI (= 3 0) 1 (* 3 ((Y H) (- 3 1)))	bêta-reduction sur n
→ (* 3 ((Y H) (- 3 1)))	règles du = et du SI
→ (* 3 ((Y H) 2))	règle du -
→ (* 3 (H (Y H) 2))	règle de Y
= (* 3 ((λf.(λn. SI (= n 0) 1 (* n (f (- n 1))))) (Y H) 2))	on remplace H par sa valeur
→ (* 3 (λn. SI (= n 0) 1 (* n ((Y H) (- n 1)))) 2))	bêta-reduction sur f
→ (* 3 (SI (= 2 0) 1 (* 2 ((Y H) (- 2 1)))))	bêta-reduction sur n
→ (* 3 (* 2 (Y H) (- 2 1)))	règles du = et du SI
→ (* 3 (* 2 (Y H) 1)))	règle du -
→ (* 3 (* 2 (H (Y H) 1)))	règle de Y
= (* 3 (* 2 ((λf.(λn. SI (= n 0) 1 (* n (f (- n 1))))) (Y H) 1)))	on remplace H par sa valeur
→ (* 3 (* 2 (λn. SI (= n 0) 1 (* n ((Y H) (- n 1)))) 1)))	bêta-reduction sur f
→ (* 3 (* 2 (SI (= 1 0) 1 (* 1 ((Y H) (- 1 1))))))	bêta-reduction sur n
→ (* 3 (* 2 (* 1 (Y H) (- 1 1))))	règles du = et du SI
→ (* 3 (* 2 (* 1 (Y H) 0))))	règle du -
→ (* 3 (* 2 (* 1 (H (Y H) 0))))	règle de Y
= (* 3 (* 2 (* 1 ((λf.(λn. SI (= n 0) 1 (* n (f (- n 1))))) (Y H) 0))))	on remplace H par sa valeur
→ (* 3 (* 2 (* 1 (λn. SI (= n 0) 1 (* n ((Y H) (- n 1)))) 0))))	bêta-reduction sur f
→ (* 3 (* 2 (* 1 (SI (= 0 0) 1 (* 0 ((Y H) (- 0 1)))))))	bêta-reduction sur n
→ (* 3 (* 2 (* 1 1)))	règles du = et du SI
→ (* 3 (* 2 1))	règle de *
→ (* 3 2)	règle de *
→ 6	règle de *

(arrêt de l'évaluation – forme normale trouvée : 6)

L'expression **Y** qui a servi de base pour cette démarche existe bien, c'est l'un des combinateurs de point fixe connus (un combinateur est une λ-expression ne contenant pas de variables libres).

Par exemple on peut prendre Y = (λh. (λx. h (x x)) (λx. h (x x))) qui permet de vérifier la réduction : Y H → H (Y H).

Y H = (λh. (λx. h (x x)) (λx. h (x x))) H
→ (λx. H (x x)) (λx. H (x x))
→ H ((λx. H (x x)) (λx. H (x x)))
= H (Y H)

3. Le langage LISP

L'un des premiers langages informatiques (comme Fortran) mais orienté vers la manipulation symbolique et basé sur un paradigme fonctionnel. Il a été (et est toujours) très utilisé en intelligence artificielle (IA).
Beaucoup de versions (dialectes) existent aujourd'hui (comonLisp, Scheme, …) et beaucoup de constructions procédurales (affectation, séquentielle, boucles...) sont aussi incluse dans le langage pour le rendre plus accessible aux programmeurs habitués avec le style impératif. Cependant il garde un noyau (formé par quelques primitives) purement fonctionnel que nous allons présenter dans ce qui suit.

Lisp se base principalement sur la manipulation de listes (d'où son nom : LISt Processor).

Les objets manipulés sont :

- les atomes (symboles et nombres)
- les listes (suite d'objet entre parenthèses)

Exemple :
(a b3 (1 2 (7a vvv)) 12) est une liste formée de 4 éléments, les 2 premiers sont des symboles, le 3e est une liste de 3 éléments et le 4e est un nombre.

La liste vide est représenté par une liste contenant 0 élément ( ) ou alors par un symbole particulier nil.

Un programme Lisp est une liste particulière appelée forme. Le premier élément de cette liste doit être le nom d'une fonction à évaluer. Les reste des éléments représentent ses arguments. C'est donc une notation préfixée.
L'interpréteur Lisp utilise la règle de l'appel par valeur, c'est à dire que pour évaluer une forme, l'interpréteur commence d'abord par évaluer les arguments en premier.

Exemple :
Pour la forme (* (+ 2 1) (- 5 3)), les sous-expressions (+ 2 1) et (- 5 3) sont évaluées avant l'expression globale (* 3 2).

Quelques fonctions et objets prédéfinis

Fonction identité

Si un argument n'est pas une fonction (et représente donc une données), on doit lui appliquer d'abord la fonction identité QUOTE qui retourne l'objet sans être évalué :
(uneFonction … (QUOTE obj) …)

Comme l'utilisation d'arguments non fonctionnel est généralement très répandue, la fonction identité peut être abrégée en 'obj :
(uneFonction … 'obj …) à la place de : (uneFonction … (QUOTE obj) …)

Les données numériques n'ont pas besoin d'être préfixées par la fonction QUOTE (ou ').

Valeurs de vérité

Les valeurs de vérité sont représentées par des objets particuliers :
le symbole T pour la valeur de vérité VRAI,
le symbole NIL ou la liste vide ( ) pour la valeur de vérité FAUX

Quelques fonctions booléennes

(CONSP obj) retourne vrai si obj est une liste non vide
(NULL obj) retourne vrai si obj est une liste vide
(SYMBOLP obj) retourne vrai si obj est un symbole
(NUMBERP obj) retourne vrai si obj est un nombre
(ATOM obj) retourne vrai si obj est un atome (un symbole ou un nombre)
(EQUAL obj1 obj2) retourne vrai si obj1 est égal à obj2

De plus les opérateurs relationnels et logique sont aussi prédéfinis :
(< ….), (> ….), (= ….), …

La conditionnelle

Le si.. alors... sinon … est une fonction (non stricte) à trois arguments :

(IF (condition...) (partie_alors....) (partie_sinon....))

L'expression condition est évaluée d'abord, si elle est vraie, la partie partie_alors est évaluée et sa valeur sera retournée par la fonction IF. Sinon c'est la partie partie_sinon qui est évaluée et sa valeur sera retournée.

Exemple :
(IF (< x 3) (+ x 1) 0) // si (x < 3) alors retourner x+1 sinon retourner 0

Manipulation de listes

(CAR ListeNonVide) retourne le premier élément de la liste

(CDR ListeNonVide) retourne la liste sans son premier élément (c'est la queue ou reste de la liste).

Exemples (remarquer l'utilisation de ' pour empêcher l'interprétation des arguments):
(CAR '(a b c)) → a
(CDR '(a b c)) → (b c)
(CAR '((ab) c (d 23 ( ) e) f)) → (a b)
(CDR '(a)) → ( )

Pour construire une nouvelle liste, on utilise la fonction :
(CONS obj1 Liste) retourne une liste dont le CAR est obj1 et le CDR est Liste

Exemples :
(CONS 'a '(b c)) → (a b c)
(CONS 'a '( )) → (a)
(CONS '(a b) '(c d)) → ((a b) c d)
(CONS '( ) '(1 2 3)) → (( ) 1 2 3)
(CONS (+ 3 2) '(a b)) → (5 a b)
(CONS '(+ 3 2) '(a b)) → ((+ 3 2) a b)

On remarque que :
(CONS (CAR X) (CDR X)) → X
(CAR (CONS X Y)) → X
(CDR (CONS X Y)) → Y

Voici une implémentation possible des fonctions de manipulation de listes en λ-calcul :
CONS := λx.λy.λf. f x y
CAR := λp. p (λx.λy. x)
CDR := λp. p (λx.λy. y)
NIL := λx. λx.λy. x

Définition de nouvelles fonctions en Lisp

Pour définir une nouvelle fonction dans l'environnement Lisp, on peut utiliser la fonction :

(DE <nomFonc> (<listeParamètres...>) (<corpsFonc...>))

Par exemple :
(DE inc (x) (+ x 1)) // incrémente son paramètre
(DE elem2 (L) (CAR (CDR L))) // retourne le 2e élément de la liste L

La récursivité et la conditionnelle sont les principales constructions permettant de définir des fonctions complexes.

Exemples :

```
// longueur d'une liste
(DE long (L)
        (IF (CONSP L)  (+ 1  (long  (CDR L)))
                       0
        )
)

// appartenance d'un élément x à une liste L
(DE app (x L)
        (IF (CONSP L)  (IF (EQUAL (CAR L) x) T  (app x (CDR L)) )
                        NIL
        )
)

// une autre version plus compacte avec la fonction OR
(DE app (x L)
        (IF (CONSP L)  (OR (EQUAL (CAR L) x)  (app  x  (CDR L)))
                       NIL
        )
)

// concaténation de 2 listes
(DE concat (L1 L2)
        (IF (CONSP L1)  (CONS (CAR L1)  (concat  (CDR L1) L2) )
                        L2
        )
)

// inversion d'une liste
(DE  inv  (L)
        (IF  (CONSP  L)  (concat  (inv  (CDR  L1))  (CONS  (CAR  L1)  '() ))
                         NIL
        )
)
```

```
// somme des éléments d'une liste de nombres
(DE som (L)
        (IF  (CONSP  L)  (+  (CAR L)  (som  (CDR L)))     0 )
)
```

utilisation : (som '(1 2 3 4)) → 10

```
// somme de tous les éléments de type nombre dans une liste quelconque
(DE som_Imbriquee  (L)
        (IF  (CONSP  L)
                (+  (som_Imbriquee (CAR L))  (som_Imbriquee (CDR L)) )
                (IF (NUMBERP  L)  L  0 )
        )
)
```

utilisation : (som_Imbriquee '(3 abc (e f (g 4)) h (((1) T23z) 2 5))) → 15

```
// sous-liste d'une liste L donnée, entre les positions I et J
(DE  sousListe (I  J  L)
        (IF (>=  J  I)
                (IF (CONSP L)
                        (IF (> I 1)  (sousListe  (- I 1)  (- J 1)  (CDR L))
                                (IF (> J 1)
                                    (CONS  (CAR L)  (sousListe 1 (- J 1)  (CDR L)) )
                                    (CONS  (CAR L)  '() )
                                )
                        )
                        NIL
                )
                NIL
        )
)
```

voici un petit déroulement :

```
(sousListe  2  4  '(a b c d e) )                                          → (b c d)
        |
        |_____ (sousListe  1  3  '(b c d e) )                              → (b c d)
                        |
                        |_____ (CONS  'b  (sousListe  1  2  '(c d e)) )    → (b c d)
                                        |
                                        |_____ (CONS  'c  (sousListe 1 1 '(d e)) )  → (c d)
                                                        |
                                                        |_____ (CONS 'd '() )  → (d)
```

Fonctions d'ordre supérieur

En Lisp chaque symbole peut avoir deux types de valeurs en même temps. La valeur nominale qui représente la valeur qu'une variable possède durant l'exécution d'une fonction (en programmation purement fonctionnelle, cela correspond à la valeur du paramètre effectifs lors de l'appel à la fonction. La valeur fonctionnelle d'un symbole est le nom d'une (éventuelle) fonction représentée

par ce symbole.
Exemple :
Soient la définition des deux fonctions f et g suivantes :
(DE f (x) (+ x 1))
(DE g (f) (* (f f) 2))
Durant l'évaluation de l'appel : (g 3), (qui retourne la valeur 8), le symbole f possède une valeur nominale : 3 et une valeur fonctionnelle : la fonction f(x)=x+1.
Lors de l'évaluation de la sous-expressions (f f), l'interpréteur utilise la valeur fonctionnelle dans la première occurrence de f et la valeur nominale dans la deuxième occurrence de f.

En règle générale, si le symbole à évaluer se trouve au début d'une forme (le premier élément de la liste), c'est sa valeur fonctionnelle qui est considérée, sinon c'est sa valeur nominale qui sera prise en compte.

Pour définir des fonctions de fonctions, c'est-à-dire des fonctions ayant comme paramètre des fonctions, on devrait alors avoir la possibilité d'utiliser la valeur nominale comme nom de fonction.

Il existe une fonction Lisp prédéfinie qui permet d'appeler la fonction dont le nom est spécifié dans la valeur nominale d'un symbole :
(FUNCALL f param1 param2 …)
L'évaluation de cet appel, indique à l'interpréteur d'appeler la fonction dont le nom est la valeur nominale de f avec les paramètres param1, param2, …

Exemples :
1) soient r et s deux fonctions :

```
(DE r (x) (+ x 1) )
(DE s (x) (* x 2) )
```

on peut alors définir une fonction m ayant deux arguments (f et x) où f est le nom d'une fonction quelconque que l'on voudrait appliquer à un nombre x donné :

```
(DE m (f x) (FUNCALL f x)
```

Les appels suivants pourront être évalués correctement :

```
(m 'r 3) → 4
(m 's 3) → 6
```

2) Voici comment définir la fonction 'mapcar' (déjà prédéfinie), une fonction qui applique la fonction donnée en premier argument aux éléments de la liste donnée en deuxième argument :

```
(DE mapcar (f L)
        (IF (CONSP L)
                (CONS (FUNCALL f (CAR L)) (mapcar f (CDR L)) )
                NIL
        )
)
```

voici des exemples d'utilisation :

```
(mapcar 'r '(1 2 3 4) ) → (2 3 4 5)
(mapcar 's '(1 2 3 4) ) → (2 4 6 8)
```

4. Preuves de programmes fonctionnels

Pour prouver formellement qu'un programme fonctionnel calcule bien une fonction donnée, on peut

utiliser le système formel de Hoare étendue par la règle des appels. Il suffit alors de transformer chaque fonction récursive en une procédure récursive dans un langage impératif quelconque et rajouter des paramètres de sorties et des affectations en fin de procédures pour renvoyer les résultats.

Une autre approche, plus intéressante, consiste à utiliser la théorie du point fixe dans le domaine des fonctionnelles. C'est ce que l'on va voir ci-dessous.

Approche point fixe

Soit $\mathscr{F}(E,F)$ l'espace des fonctions partielles de E dans F, ordonné par la relation « moins définie que » entre fonctions (notée $\subseteq$) :
$f \subseteq g$ veut dire, $\forall x \in E$ tel que f(x) est définie, alors g(x) est aussi définie et g(x) = f(x).

Propriété : ($\mathscr{F}(E,F)$, $\subseteq$) est inductif et son plus petit élément (Ω) est la fonction nulle part définie ($\forall x \in E$, $\Omega(x)$ n'est pas définie et donc $\forall f \in \mathscr{F}(E,F)$, on aura toujours $\Omega \subseteq f$).

Un programme fonctionnel récursif p, dont le corps est défini par une expression fonctionnelle $\tau(p)$ est une équation à point fixe dans $\mathscr{F}(E,F)$:
$p = \tau(p)$
dont la plus petite solution coïncide exactement avec la fonction calculée par le programme.

D'après le théorème du point fixe, la plus petite solution f_0 est, dans ce cas, donnée par :
$f_0 = Sup(\tau^n(\Omega))_{n\geq 0}$

De cette manière, on a deux possibilités de faire une preuve de correction :

a- preuve totale : à partir d'un programme donné, on retrouve la fonction calculée exactement. On dit qu'un programme p calcule exactement une fonction f ssi f n'est définie qu'aux points x où le programme p(x) s'arrête et en plus la valeur retournée par p(x) est égale à f(x).
Pour cela il suffit de calculer la plus petite solution (le plus petit point fixe) de l'équation $p = \tau(p)$, qui est donnée par : $f_0 = Sup(\tau^n(\Omega))_{n\geq 0}$.

b- preuve partielle : à partir d'un programme et d'une fonction donnés, on vérifie si le programme calcule partiellement la fonction. On dit qu'un programme p calcule partiellement une fonction f ssi à chaque fois que le programme p s'arrête pour une donnée en entrée x, la fonction f est définie au point x, de plus, la valeur retournée par le programme (p(x)) est égale à f(x).
Pour ce faire, il suffit de vérifier que f est bien un point fixe de l'équation $p = \tau(p)$,
c-a-d que : $f = \tau(f)$.

Exemple de preuve totale :

Etant donné un programme **p**, on cherche à retrouver la fonction calculée exactement par **p**.
P = (λn. Si (= n 0) 1 (* n (**p** (- n 1)))
$p = \tau (p)$

- Commençons par calculer ($\tau^1(\Omega)$) :
il suffit de remplacer p par Ω dans le programme τ :

$\tau^1(\Omega)$ = (λn. Si (= n 0) 0 (* n (Ω (- n 1)))
cette fonction est définie en un seul point (n = 0) et elle retourne 1, sinon elle n'est pas définie car Ω(n-1) n'est pas définie et donc l'expression n*Ω(n-1) n'est pas définie aussi.

- Calculons maintenant ($\tau^2(\Omega) = \tau(\tau(\Omega))$) :
il suffit de remplacer p par $\tau(\Omega)$ dans le programme τ :
$\tau^2(\Omega)$ = (λn. Si (= n 0) 1 (* n ($\tau(\Omega)$ (- n 1)))
comme $\tau(\Omega)$ est une fonction qui n'est définie que lorsque son argument est égal à 0, l'expression :
(* n ($\tau(\Omega)$ (- n 1))
ne sera définie que lorsque n-1 = 0 (donc pour n=1) et sa valeur sera alors = 1*1 (donc 1)
donc $\tau^2(\Omega)$ est une fonction définie en 2 points, pour n=0, elle vaut 1 et pour n=1 elle vaut aussi 1. Ailleurs, elle n'est pas définie.

- Continuons l'évaluation de $\tau^3(\Omega)$ (= $\tau(\tau^2(\Omega))$) :
il suffit alors de remplacer p par $\tau^2(\Omega)$ dans τ :
$\tau^3(\Omega)$ = (λn. Si (= n 0) 1 (* n ($\tau^2(\Omega)$ (- n 1)))
comme $\tau^2(\Omega)$ est une fonction qui n'est définie qu'en deux points : lorsque son argument est égal à 0 ou alors à 1 et sa valeur est égale à 1 dans les 2 cas, l'expression : (* n ($\tau^2(\Omega)$ (- n 1)) sera elle aussi définie qu'en 2 points, lorsque son argument (n-1) est égal à 0 ou alors à 1 (donc pour n=1 ou n=2) et sa valeur sera alors :
n*$\tau^2(\Omega)$(n-1) = 1 * 1 (si n=1) et
n*$\tau^2(\Omega)$(n-1) = 2 * 1 (si n=2)
donc $\tau^3(\Omega)$ est une fonction définie en 3 points, pour n=0, elle vaut 1, pour n=1 elle vaut aussi 1 et pour n=2 sa valeur est 2. Ailleurs elle n'est pas définie.

On peut remarquer que $\tau^m(\Omega)$ calcule n! pour tout n<m, sinon elle n'est pas définie. On montre alors par récurrence que cette hypothèse est correcte (on aurait pu calculer $\tau^4(\Omega)$, $\tau^5(\Omega)$, … jusqu'à deviner une hypothèse adéquate à démontrer).

Notre hypothèse de récurrence est donc : $\tau^m(\Omega)(n) = n!$ pour tout n<m,
montrons alors que : $\tau^{m+1}(\Omega)(n) = n!$ pour tout n<(m+1)

D'après l'équation à point fixe, on peut écrire :
$\tau^{m+1}(\Omega)$ = (λn. Si (= n 0) 1 (* n ($\tau^m(\Omega)$ (- n 1))) car $\tau^{m+1}(\Omega) = \tau(\tau^m(\Omega))$).
Or d'après l'hypothèse de récurrence on a :
(* n ($\tau^m(\Omega)$ (- n 1)) = (* n (n-1)!) pour tout (n-1)<m, c-a-d n<(m+1), sinon c'est indéfini, donc,
(* n ($\tau^m(\Omega)$ (- n 1)) = n! pour tout n<(m+1) et donc $\tau^{m+1}(\Omega)(n) = 1$ si n=0 et n! si 0<n<m+1,
c-a-d $\tau^{m+1}(\Omega)(n) = n!$ pour tout n<(m+1).
On conclue alors que la propriété est vrai pour tout n.

Ainsi le plus petit point fixe $\tau^\infty(\Omega)(n)$ est la fonction n! et c'est donc la fonction calculée exactement par le programme p.

Exemple de preuve partielle :

Etant donné un programme **q**, on cherche à savoir s'il calcule partiellement une fonction g donnée :
q = (λxy. Si (= x y) (+ y 1) (**q** x (**q** (- x 1) (+ y 1))))

q = τ (q)

g(x,y) = x+1 pour tout couple d'entiers (x,y)

Il suffit alors juste de vérifier si g est une solution à l'équation :
g = τ(g)

Calculons τ(g) :
τ(g) = (λxy. Si (= x y) (+ y 1) (g x (g (- x 1) (+ y 1))))

Si x=y, τ(g) vaut y+1 qui est égale aussi à x+1 et donc à g aussi

Si x ≠ y, τ(g) vaut : g(x, g(x-1,y+1)), vérifions alors que c'est égal à g lorsque x ≠ y :
g(x-1,y+1) = x (d'après la définition de g : g(x,y) = x+1)
et donc g(x, g(x-1,y+1)) = g(x,x) = x+1
donc τ(g) = x+1 = g(x,y) lorsque x=y ou x ≠ y
donc le programme **q** calcule bien partiellement la fonction g.

Montrons que le même programme **q** calcule aussi partiellement la fonction h, définie par :
h(x,y) = x+1 si x ≥ y et h(x,y) = y-1 si x <y

Il suffit donc de montrer que h est solution de l'équation à point fixe, c-a-d :
h = τ(h)

τ(h) = (λxy. Si (= x y) (+ y 1) (h x (h (- x 1) (+ y 1))))

Dans la partie alors (x=y), τ(h) retourne y+1 qui est donc aussi égale à x+1
et dans les mêmes conditions (x=y), la fonction h(x,y) vaut aussi x+1

Dans la partie sinon (x ≠ y), τ(h) retourne h(x, h(x-1,y+1)),
on remarque que h(x-1,y+1) = x si x-1≥ y+1 et h(x-1,y+1) = y si x-1 < y+1,
donc :
h(x, h(x-1,y+1)) = h(x,x) = x+1 si x-1 ≥ y+1 (c-a-d x ≥ y+2) et,
h(x, h(x-1,y+1)) = h(x,y) si x-1 < y+1 (c-a-d x < y+2)

donc :
τ(h) = x+1 = h(x,y) si x≥ y+2 ou x = y et,
τ(h) = h(x,y) si x < y+2 et x ≠ y
donc τ(h) = h(x,y) pour tout x et pour tout y.
Donc le programme **q** calcule (aussi) partiellement la fonction h.
(pourtant g ≠ h !)

Pour trouver la fonction calculer exactement par le programme **q**, il suffit de calculer le plus petit point fixe de l'équation q = τ(q) (à faire comme exercice).