Programmation fonctionnelle en langage Scheme

- Support PG104 -

Myriam Desainte-Catherine et David Renault

ENSEIRB-MATMECA, département d'informatique

January 21, 2020

Objectifs pédagogiques

Objectif général

Découverte de la programmation fonctionnelle pure à travers :

- Les formes qu'elle prend syntaxiquement (expressions, fonctions et listes récursives)
- Les méthodes qu'elle permet de déployer.

Compétences générales attendues

- Spécifier un calcul de façon fonctionnelle plutôt qu'impérative : programmer au moyen d'expressions plutôt que d'instructions
- Spécifier des calculs récursivement plutôt qu'itérativement
- Spécifier un calcul générique : abstraire un calcul au moyen de paramètres fonctionnels et de retours fonctionnels
- Comparer des solutions selon le style et la complexité

Ce support est accessible en version électronique mise à jour régulièrement aux adresses suivantes :
http://www.labri.fr/perso/myriam/Enseignement/Scheme/scheme.pdf
http://www.labri.fr/perso/renault/working/teaching/schemeprog/schemeprog.php

Documentation : https://docs.racket-lang.org/

Chapitre 1 - Introduction

Concepts et terminologie

Concepts fonctionnels

- **Écriture fonctionnelle** : programmation par applications de fonctions plutôt que par l'exécution de séquences d'instructions
- **Transparence référentielle** : chaque expression peut être remplacée par son résultat sans changer le comportement du programme – sans effets de bord[a]
- **Programmation fonctionnelle pure** : sans effets de bords, avec transparence référentielle.
- **Fonctions de première classe** : type fonction, constantes fonction, opérateurs sur les fonctions

[a] Un effet de bord est une modification de l'environnement (affectation ou E/S)

Autres concepts nouveaux

- **Typage dynamique** : Les variables sont typées au moment de l'exécution et non au moment de la compilation
- **Références** : ce sont des adresses sur des objets, elles sont utilisées chaque fois que les contenus ne sont pas utiles (passages de paramètres, retours de fonctions)
- **Garbage collector ou ramasse-miettes** : gestion dynamique et automatique de la mémoire. L'utilisateur ne s'occupe pas de désallouer la mémoire.

Historique

Les langages des années 1950

- FORTRAN (1954) : calcul scientifique, données et calcul numérique.
- Lisp (1958) : calcul symbolique, données et algorithmes complexes (IA), démonstrations automatiques, jeux etc.
- Algol (1958) : langage algorithmique et structuré, récursivité.

Les langages lisp

- 1958 : John Mac Carthy (MIT)
- 1986 : common lisp ANSI (portabilité, consistence, expressivité, efficacité, stabilité).
- Les enfants de lisp :
  - Logo (1968), langage visuel pédagogique
  - Smalltalk (1969, Palo Alto Research Center de Xerox) premier langage orienté objets
  - ML (1973, R. Milner, University of Edinburgh), preuves formelles, typage statique, puis CaML (1987 INRIA), projet coq, et Haskell purement fonctionnel, paresseux.
  - Scheme (1975, Steele et Sussman MIT) mieux défini sémantiquement, portée lexicale, fermeture, et continuations de première classe.
  - emacs-lisp (Stallman 1975, Gosling 1982), langage d'extension de GNU Emacs.
  - CLOS (1989, Common Lisp Object System), common lisp orienté objets.

Le λ-calcul

Théorie des fonctions d'Alonzo Church (1930), modèle universel de calcul, directeur de thèse d'Alan Turing (machines de Turing, théorie de la calculabilité).

Syntaxe – λ-termes

- **Variables** : x, y, ...
- **Applications** : si u et v sont des λ-termes uv est aussi un λ-terme. On peut alors voir u comme une fonction et v comme un argument, uv étant alors l'image de v par la fonction u.
- **Abstractions** : si x est une variable et u un λ-terme alors $\lambda x.u$ est un λ-terme. Intuitivement, $\lambda x.u$ est la fonction qui à x associe u.

Exemple

- Constante : $\lambda x.y$
- Identité : $\lambda x.x$
- Fonction renvoyant une fonction : $\lambda x.\lambda y.a$
- Application : xyz ou $((xy)z)$
- Fonctions à plusieurs arguments : $\lambda xy.a$

Remarques: les applications sont faites de gauche à droite en l'absence de parenthèses, une occurrence de variable est dite **muette** ou **liée** si elle apparaît dans le corps d'un λ-terme dont elle est paramètre, sinon elle est dite **libre**.

Le λ-calcul – la substitution

Cette opération permet de remplacer les occurrences d'une variable par un terme pour réaliser le calcul des λ-termes. On note **t[x := u]** la substitution dans un lambda terme t de toutes les occurrences d'une variable x par un terme u.

Exemple

Dans ces exemples, les symboles x, y, z, a sont des variables.

- Dans une application : $xyz[y := a] = xaz$
- Dans une abstraction (cas normal) : $\lambda x.xy[y := a] = \lambda x.xa$
- Capture de variable libre : $\lambda x.xy[y := ax] = \lambda z.zax$(et non $\lambda x.xax$), renommage de la variable liée
- Substitution inopérante (sur variable liée): $\lambda x.xy[x := z] = \lambda z.zy = \lambda x.xy$

Définition

- **Variable**: si t est une variable alors $t[x := u] = u$ si $x = t$ et t sinon
- **Application**: si $t = vw$ alors $t[x := u] = v[x := u]w[x := u]$ si v et w sont des termes.
- **Abstraction**: si $t = \lambda y.v$ alors $t[x := u] = \lambda y.(v[x := u])$ si $x \neq y$ et y n'est pas une variable libre de u. Si y est une variable libre de u, on renomme y avant de substituer. Si $x = y$ le résultat est t.

Le λ-calcul – la β-réduction

On appelle **rédex** un terme de la forme $(\lambda x.u)v$. On définit alors la β-réduction

$$(\lambda x.u)v \longrightarrow u[x := v]$$

- La réduction du terme $(\lambda x.u)v$ est la valeur de la fonction $\lambda x.u$ appliquée à la variable v.
- u est l'image de x par la fonction $(\lambda x.u)$,
- L'image de v est obtenue en substituant dans u, x par v.

Exemple

- $(\lambda x.xy)a$ donne $xy[x := a] = ay$
- $(\lambda x.y)a$ donne $y[x := a] = y$

Remarque Les termes sont des arbres avec des noeuds binaires (applications), des noeuds unaires (les λ-abstractions) et des feuilles (les variables). Les réductions permettent de modifier l'arbre, cependant l'arbre n'est pas forcément plus petit après l'opération. Par exemple, si l'on réduit

$$(\lambda x.xxx)(\lambda x.xxx)$$

on obtient

$$(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)$$

Le λ-calcul – la normalisation

Un lambda-terme t est dit en forme normale si aucune β-réduction ne peut lui être appliquée, c'est-à-dire si t ne contient aucun rédex.

Remarques

- On peut simuler la normalisation des λ-termes à l'aide d'une machine de Turing, et simuler une machine de Turing par des λ-termes.
- Différentes stratégies de réduction sont définies dans le λ-calcul : stratégie applicative (par valeur, dans les langages lisp et scheme), stratégie paresseuse (par nom, dans Haskell).
- La normalisation est un calcul confluent. Soient t, u1 et u2 des lambda-termes tels que $t \longrightarrow u1$ et $t \longrightarrow u2$. Alors il existe un λ-terme v tel que $u1 \longrightarrow v$ et $u2 \longrightarrow v$. Conséquence : l'ordre d'évaluation des arguments d'une fonction n'a pas d'influence sur le résultat.

Exemple

Les symboles x, y, z, a sont des variables. Soit le terme $(\lambda x.y)((\lambda z.zz)a)$

- Stratégie applicative : $(\lambda x.y)\ \boxed{((\lambda z.zz)a)} \longrightarrow (\lambda x.y)\ \boxed{aa} \longrightarrow y$
- Stratégie paresseuse : $\boxed{(\lambda x.y)((\lambda z.zz)a)} \longrightarrow \boxed{y}$

Lien avec la syntaxe lisp

La syntaxe lisp est complètement basée sur le λ-calcul. Les parenthèses servent à délimiter les termes et les applications.

- Variables : x, et constantes de types numériques, symbolique, fonctionnel etc.
- Abstractions fonctionnelles : $\lambda x.y$ s'écrit (**lambda**(x) y)
- Application : uv s'écrit (u v)
  - Cas d'une abstraction fonctionnelle : ((**lambda**(x) y) a)
  - Cas d'une fonction nommée f (variable) ; fx s'écrit (f x)

Exemple

- Application d'une abstraction fonctionnelle
  - `((lambda (x) (x y)) a)` $\longrightarrow$ `(a y)`
  - `((lambda (x y) (x z y)) a b)` $\longrightarrow$ `(a z b)`
- Application d'une fonction nommée
  - Soit f la fonction `(lambda (x) (x y))`
    `(f a) = ((lambda (x) (x y)) a)` $\longrightarrow$ `(a y)`
  - Soit f la fonction `(lambda (x) (+ x 1))`, avec `+` correspondant à l'opération d'addition :
    `(f 2) = ((lambda (x) (+ x 1)) 2)` $\longrightarrow$ `(+ 2 1)` $\longrightarrow$ 3

Développement incrémental

Boucle Read Eval Print : REPL

1. Read : Lecture d'une expression
2. Eval : calcul (réduction) de l'expression
3. Print : affichage du résultat (forme normale)
4. Affichage du prompt $>$ et retour à 1

- **Top-level** : niveau de la REPL, l'imbrication des expressions induit plusieurs niveaux. Par exemple pour l'expression (+ 3 4 (* 1 2) 3)
  $\longrightarrow$ évaluation de ($*$ 1 2)
  $\longrightarrow$ résultat 2
  $\longrightarrow$ évaluation de (+ 3 4 3 3)
  $\longrightarrow$ résultat 13
- Notation

```
> (+ 3 4 (* 1 2) 3)
12
```

Définition et évaluation des expressions

Expressions symboliques

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

- Objet (nb, chaîne, symbole, etc.)
- Expression composée (sexpr sexpr ... sexpr) : liste de sexpr. Utilisé à la fois pour le code et les données :
  - Notation de l'application d'une fonction à ses arguments.
  - Notation des listes[a] : '(e1 e2 ... en)

[a] Pour éviter l'application et fabriquer une liste, il faut la faire précéder d'une quote

Evaluation

- Objets auto-évaluants : objet lui-même (nombres, booléens, caractères, chaînes de caractères).
- Symboles : valeur associée (identificateurs)
- Expression symbolique composée : application – évaluation de l'objet en position fonctionnelle (la première), évaluation des arguments[a], puis application de la fonction aux arguments et renvoi du résultat.

[a] dans un ordre non spécifié

Chapitre 2 - Types et constructions de base du langage

Résumé des constructions syntaxiques du langage

Types

- Simples : Boolean, Number, Character, String, Symbol
- Conteneurs : Pair, List, Vector
- Fonctions : Procedure

Variables et liaisons

- Locales : **let**, **let∗**, **letrec**, **letrec∗**, **let−values**, **let∗−values**
- Globales ou locales : **define**

Formes

Expression ou formes **define**, **let**, **lambda**, **if**, **cond**, **set!**, etc.

Expressions

constante, (f a_1 a_2 ... a_n)

Procédures

lambda

Macros

define−syntax

Continuations

call-with-current-continuation

Bibliothèques

library, import, export

Les types

Les booléens

- Constantes : #t et #f
- Toute valeur différente de #f est vraie
- L'objet #t est utilisé comme valeur vrai, quand aucune autre valeur ne paraît plus pertinente.
- Prédicat boolean?

Opérations booléennes

- **and** : stop au premier argument évalué à faux
- **or** : stop au premier argument évalué à vrai
- **not**
- nand, nor, xor, implies

Les opérateurs *and*, *or*, *nand*, *nor* et *xor* admettent n arguments, $n \geq 0$ et sont des formes spéciales.

Exemple

```
> (and)
#t
> (or)
#f
```

Les types

Tour des types numériques

- Number
- Complex
- Real
- Rational
- Integer

Exactitude

- Prédicat : `exact?`

Les nombres

Les entiers

- Taille non limitée (seulement par la mémoire)
- Prédicat : integer?

Exemple

```
> (integer? 1)
#t
> (integer? 2.3)
#f
> (integer? 4.0)
#t
> (exact? 4)
#t
```

Les nombres

Les rationnels

- 523/123
- Accesseurs : **numerator**, **denominator**
- Prédicats : **rational**?

Les réels

- 23.2e10
- Prédicat : real ?

Exemple

```
> (real? 1)
#t
> (exact? 4.0)
#f
```

Les nombres

Les complexes

- 3+2i
- Contructeurs : make−polar, make−rectangular
- Accesseurs : real−part, imag−part, magnitude, angle
- Prédicat : **complex**?

Exemple

```
> (sqrt -1)
0+1i
> (complex? -1)
#t
> (real? 1+2i)
#f
```

Prédicats numériques

- Nombres : zero?, positive ?, negative?
- Entiers : even?, odd?
- Comparaisons : = < <= > >= sur les réels.
- Égalités et inégalités sur les complexes.
- Nombre d'opérandes supérieur à 2.

Exemple

```
> (= 1 2 3)
#f
> (= 1 1 1)
#t
```

Opérations numériques

Arithmétiques	$+$, $-$, $*$, $/$	*n* arguments réels, $n \geq 0$
Unaires	add1, sub1	incrémentation, décrémentation
	max et **min**	*n* arguments réels, $n > 0$
Exponentiation	**sqr**, **sqrt**, **log**	
	exp	exponentielle naturelle
	expt	base arbitraire, exposant
Modulaires	modulo	
	quotient, remainder	
	quotient/remainder	renvoie 2 valeurs
	gcd, **lcm**, **abs**	
Arrondis	**floor**, **ceiling**,	
	truncate, **round**	
Trigonométrie	**sin**, **cos**, **tan**,	
	asin, **acos**, **atan**	

Les caractères et les chaînes de caractères

Constantes

- Caractère : #\a
- Chaîne : "de␣caracteres"

Prédicats

- Type :**char**?, **string**?
- Comparaisons : **char**=?, **char**<?, **char**>?,
  string=?, **string**<?, **string**>?.

Fonctions

- Constructeurs : make−string, **string**
- Accesseurs : string−ref
- Longueur : string−length
- Conversion : number−>**string**, string−>number

Les symboles

Constantes

Ce sont à la fois des noms d'identificateurs de variables et de fonctions, et des données symboliques.

- Suite de caractères alphabétiques et numériques
- Plus les caractères suivants :

  ! $ % & * + - . / : < = > %? ˜
- Échappements pour les délimiteurs[a] : |*symbol*|

---

[a] ( ) [ ] " , ' ‘ ;

Prédicats

- Type : symbol?
- Égalité : **eq**?

Conversions

- symbol−>**string**, string−>symbol

Les expressions conditionnelles

La forme **if**

(**if** ⟨*condition*⟩ ⟨*alors*⟩ ⟨*sinon*⟩)

- ⟨*condition*⟩, ⟨*alors*⟩ et ⟨*sinon*⟩ sont des expressions
- Si ⟨*condition*⟩ vaut vrai, le résultat est la valeur de l'expression ⟨*alors*⟩
- Sinon, le résultat est la valeur de l'expression ⟨*sinon*⟩

Exemple

```
> (if 1 2 3)
2
> (if (= 1 2) 3 4)
4
> (if (= 1 2) 3)
; error -> if: missing an "else" expression
```

La forme **when**

(**when** $\langle condition \rangle$ $\langle e_1 \rangle$ ... $\langle e_n \rangle$)

Cette forme évalue les expressions $\langle e_i \rangle$ et renvoie le résultat de la dernière quand l'expression $\langle condition \rangle$ vaut vrai.

La forme **unless**

(**unless** $\langle condition \rangle$ $\langle e_1 \rangle$ ... $\langle e_n \rangle$)

Même chose mais quand l'expression $\langle condition \rangle$ vaut faux.

La forme **cond**

$(\textbf{cond} \ \langle c_1 \rangle \ \ldots \ \langle c_n \rangle)$

- Les $\langle c_i \rangle$ sont des clauses : $[\langle \textit{condition} \rangle \ \langle e_1 \rangle \ \ldots \ \langle e_n \rangle \ ]$
- $\langle \textit{condition} \rangle$, $\langle e_1 \rangle$, ... $\langle e_n \rangle$ sont des expressions
- Évaluation des conditions des clauses dans l'ordre de $\langle c_1 \rangle$ à $\langle c_n \rangle$
- Soit c_i=[c e_1 ... e_n] la première clause dont la condition c vaut vrai, les e_i sont évaluées dans l'ordre et le résultat est celui de e_n.

Exemple

```
(cond [(number? x) "X est un nombre"]
      [(symbol? x) "X est un symbole"]
      [else (...)])
```

Les crochets définissant les clauses peuvent être remplacés par des parenthèses, conformément à la norme R6RS du langage Scheme

Définitions

- Un symbole est un identificateur, c'est-à-dire un nom symbolique.
- Une liaison est une entité, c'est-à-dire un objet nommé résidant dans la mémoire, donc l'association d'un symbole avec un emplacement mémoire contenant une valeur.

Exemple

```
int
g(int i)
{
  return i;
}
```

```
int
f(int n)
{
  int i=1;
  return i+n;
}
```

Dans un programme un même symbole peut apparaître dans plusieurs liaisons. De même, en C, un identificateur peut aussi servir à nommer plusieurs entités. Plusieurs stratégies de recherche ont été implémentées dans les langages de programmation.

Environnements global et locaux

L'environnement est formé de **liaisons** *symbole* $\longrightarrow$ *valeur*. Les symboles ne sont pas typés (non déclarés), mais leurs valeurs le sont. Il s'agit d'un **typage dynamique**.

Environnement global : la forme **define** au top-level

- Variables : (**define** $\langle v \rangle$ $\langle e \rangle$)
- Fonctions : (**define** ($\langle f \rangle$ $\langle p_1 \rangle$ $\langle p_2 \rangle$ ... $\langle p_n \rangle$) $\langle e_1 \rangle$ $\langle e_2 \rangle$ ... $\langle e_n \rangle$)
- Résultat non spécifié par la norme

Une définition établit une liaison entre une variable et un objet résultant de l'évaluation de l'expression, cet objet pouvant être une fonction.

Exemple

```
> (define a 0)
> (define (f x) x)
> (define g (lambda(x) (* 2 x)))
```

```
> a
0
> (f 1)
1
```

```
> (g 1)
```

1 2 error

Environnements locaux - Forme **let**

Ils sont fabriqués avec les formes **let**, **let** $*$, **letrec**, et par des définitions au moyen de la forme **define** dans le corps des fonctions.

La forme **let**

```
(let (⟨l1⟩
      ⟨l2⟩
      ...
      ⟨ln⟩)
  ⟨e⟩)
```

- $\langle l_i \rangle$ est une liaison : $(\langle s_i \rangle\ \langle o_i \rangle)$
- $\langle s_i \rangle$ est un symbole (id. de variable)
- $\langle o_i \rangle$ une valeur d'initialisation
- $\langle e \rangle$ est une expression
- Résultat de l'évaluation de l'expression $\langle e \rangle$ dans l'environnement créé

L'évaluation des valeurs d'initialisation est effectuée en premier puis les variables locales sont créées. Ce qui implique que les valeurs des variables locales définies dans un let ne sont pas utilisées dans l'évaluation des expressions d'initialisation.

Environnements locaux - Forme **let**$*$

La forme **let**$*$

```
(let* (⟨l1⟩
       ⟨l2⟩
       ...
       ⟨ln⟩)
  ⟨e⟩)
```

- $\langle l_i \rangle$ est une liaison : $(\langle s_i \rangle\ \langle o_i \rangle)$
- $\langle s_i \rangle$ est un symbole (id. de variable)
- $\langle o_i \rangle$ une valeur d'initialisation
- $\langle e \rangle$ est une expression
- Résultat de l'évaluation de l'expression $\langle e \rangle$ dans l'environnement créé

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

Exemple

```
> (let ((a 2)
        (b 3)
        (c 0))
    (- (* b b)
       (* 4 a c)))
9
```

```
> (let* ((a 2)
         (b 3)
         (c a))
    (- (* b b)
       (* 4 a c)))
-7
```

Environnements locaux - Forme **letrec**

La forme **letrec**

```
(letrec (⟨l1⟩
         ⟨l2⟩
          ...
         ⟨ln⟩)
  ⟨e⟩)
```

- $\langle l_i \rangle$ est une liaison : $(\langle s_i \rangle\ \langle o_i \rangle)$
- $\langle s_i \rangle$ est un symbole (id. de variable)
- $\langle o_i \rangle$ une lambda-expression
- $\langle e \rangle$ est une expression
- Résultat de l'évaluation de l'expression $\langle e \rangle$ dans l'environnement créé

L'évaluation des expressions d'initialisation permet les définitions récursives.

Exemple

```
> (letrec ((fact
            (lambda (n)
              (if (zero? n)
                  1
                  (* n (fact (sub1 n)))))))
    (fact 2))
> 2
```

Stratégies de recherche d'une liaison

Pour chercher la liaison correspondant à l'occurrence d'un symbole dans une expression, la recherche commence par l'environnement dans lequel apparaît l'expression. Si l'occurrence apparaît dans le corps d'une fonction et qu'aucune liaison ne correspond en local (cas d'une variable libre), deux stratégies existent.

Stratégie lexicale – Lexical scope

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 et au langage Scheme.

Stratégie dynamique – Dynamic scope

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.
Common-Lisp implémente les deux stratégies.

Portées lexicales et dynamiques

Racket : lexical

```
> (define i 0)
> (define (f x) (* x i))
> (f 3)
 0
> (let ((i 2)) (f 3))
 0
> (let ((j 0))
    (let ((g (lambda(x)
               (* x j))))
      (let ((j 3))
        (g 3))))
 0
```

emacs-lisp : dynamique

```
> (defvar i 0)
> (defun f(x) (* x i))
> (f 3)
 0
> (let ((i 2)) (f 3))
6
> (let ((j 0))
    (flet ((g (x)
             (* x j)))
      (let ((j 2))
        (g 3))))
6
```

Common Lisp : dynamique

```
> (defvar i 0); variable speciale
> (defun f(x) (* x i))
> (f 3)
 0
> (let ((i 2)) (f 3))
6
```

Common Lisp : lexical

```
> (let ((j 0)); variable lexicale
    (flet ((g (x)
             (* x j)))
      (let ((j 2))
        (g 3))))
0
```

Portée dynamique en LaTeX

- Style book : saute 2 pages avant la table of contents
- Style report : saute 1 page avant la table of contents
- On souhaite ne sauter qu'une page dans le style book
- Commande `cleardoublepage` saute 2 pages
- Commande `clearpage` saute 1 page
- Commande `renewcommand` redéfinit une commande

Exemple

```
%% creation d'un bloc avec environnement local
\begingroup
\renewcommand{\cleardoublepage}{\clearpage}
\tableofcontents
\endgroup          %% sortie du bloc,
...
\cleardoublepage   %% \cleardoublepage restauree
```

Portée et durée de vie en Scheme

Portée lexicale

La portée d'une liaison est la partie du code source dans laquelle il est possible de l'utiliser.

- Les liaisons globales ont une portée égale à tout le programme.
- Les liaisons locales ont une portée limitée à la forme de définition let.

Durée de vie

La durée de vie d'un objet correspond à la période de l'exécution d'un programme comprise entre la création de cet objet et sa destruction.

- Les objets définis globalement ont une durée durée de vie égale à celle du programme.
- Les objets définis localement ont une durée de vie potentiellement égale à celle du programme.

Paradigme fonctionnel et environnements

Équivalence des formes Let et Lambda

La forme **let** équivaut à l'application d'une fonction construite avec la forme **lambda**. Les symboles définis correspondent aux paramètres formels de la fonction, et les expressions associées aux symboles définis correspondent aux arguments de l'application.

```
(let ((j 0))
  (* x j))
```

```
((lambda(j) (* x j)) 0)
```

Questions

La forme **let** est-elle fonctionnelle? oui ou non

La forme **let** $*$ est-elle fonctionnelle? oui ou non

Environnements et paradigme fonctionnel

L'organisation des environnements et des stratégies de recherche des liaisons sont des mécanismes qui ont été ajoutés au lambda-calcul pour en faire un langage de programmation opérationnel. Mais quels sont les liens entre ces mécanismes et le paradigme fonctionnel?

La portée lexicale correspond-elle au paradigme fonctionnel?

- Caractéristiques de la programmation fonctionnelle
  - Fondement provenant du λ-calcul
  - Programmation par applications de fonctions
  - Transparence référentielle
- Reformulation de la question
  - La portée lexicale implique-t-elle la transparence référentielle?
  - La portée lexicale implique-t-elle des constructions qui peuvent être simulés par des applications de fonctions?

Portée lexicale / dynamique – Conclusion

Portée lexicale et pureté fonctionnelle

- La portée lexicale respecte la transparence référentielle en associant à chaque fonction l'environnement lexical définissant leurs variables libres
- La portée lexicale correspond au calcul purement fonctionnel et les constructions d'environnements locaux peuvent être simulées par des applications de fonctions.

Portée dynamique et extensibilité

- La portée dynamique apporte davantage d'extensibilité que la portée lexicale.

Extensibilité – cas lexical (RMS - emacs-lisp)

- Fonction A — Extension utilisateur
  - Définit la variable FOO
  - Appelle la fonction B – avec paramètre FOO
- Fonction B — Système standard
  - Appelle la fonction C – avec paramètre FOO
- Fonction C — Extension utilisateur
  - Utilise la variable FOO

La solution consistant à utiliser un paramètre pour faire passer la variable FOO depuis la fonction A vers la fonction C revient à modifier l'appel de la fonction C dans la fonction B et en conséquence, cet appel étant générique, tous les appels aux autres commandes.

Extensibilité – cas dynamique (RMS - emacs-lisp)

- Fonction A — Extension utilisateur
  - Définit la variable FOO
  - Appelle la fonction B – variable à portée dynamique
- Fonction B — Système standard
  - Appelle la fonction C
- Fonction C — Extension utilisateur
  - Utilise la variable FOO

La solution consistant à utiliser une variable à portée dynamique dans la fonction A n'implique aucun changements pour la fonction B ni pour les autres commandes existantes.

Portée dynamique – Richard Stallman

Some language designers believe that dynamic binding should be avoided, and explicit argument passing should be used instead. Imagine that function A binds the variable FOO, and calls the function B, which calls the function C, and C uses the value of FOO. Supposedly A should pass the value as an argument to B, which should pass it as an argument to C.

This cannot be done in an extensible system, however, because the author of the system cannot know what all the parameters will be. Imagine that the functions A and C are part of a user extension, while B is part of the standard system. The variable FOO does not exist in the standard system; it is part of the extension. To use explicit argument passing would require adding a new argument to B, which means rewriting B and everything that calls B. In the most common case, B is the editor command dispatcher loop, which is called from an awful number of places.

What's worse, C must also be passed an additional argument. B doesn't refer to C by name (C did not exist when B was written). It probably finds a pointer to C in the command dispatch table. This means that the same call which sometimes calls C might equally well call any editor command definition. So all the editing commands must be rewritten to accept and ignore the additional argument. By now, none of the original system is left!

Spécification récursive sous forme d'équations fonctionnelles

$fact(0) = 1$
$fact(n) = n * fact(n - 1)$

Programme Scheme récursif

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```

Récursivité

Evaluation

$>$ (fact 4)
$\longrightarrow$ (* 4 (fact 3))
$\longrightarrow$ (* 4 (* 3 (fact 2)))
$\longrightarrow$ (* 4 (* 3 (* 2 (fact 1))))
$\longrightarrow$ (* 4 (* 3 (* 2 (* 1 (fact 0)))))
$\longrightarrow$ (* 4 (* 3 (* 2 (* 1 1))))
$\longrightarrow$ (* 4 (* 3 (* 2 1)))
$\longrightarrow$ (* 4 (* 3 2))
$\longrightarrow$ (* 4 6)
$\longrightarrow$ 24

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```

Pour évaluer l'application de cette fonction, une pile est nécessaire pour stocker les valeurs successives de n qui sont utilisées dans le calcul lors du retour des appels récursifs.

Récursivité et pile d'appels

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```

Evaluation

$>$ (fact 4)

$\longrightarrow$ (* n (fact (sub1 n)))

$\longrightarrow$ (* n (fact (sub1 n)))

$\longrightarrow$ (* n (fact (sub1 n)))

$\longrightarrow$ (* n (fact (sub1 n)))

Pile d'appels

n $\longrightarrow$ 4

n $\longrightarrow$ 3

n $\longrightarrow$ 2

n $\longrightarrow$ 1

Résultats

- 24
- 6
- 2
- 1

Est-il nécessaire d'empiler toutes les valeurs de n pour calculer la factorielle?

Oui | *Non* | *Je ne sais pas*

Récursivité et itération

Version impérative itérative

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

Version fonctionnelle

```
(define (fact-t n r)
  (if (zero? n)
      r
     (fact-t (sub1 n)
             (* n r))))
```

Spécification fonctionnelle

$$fact\text{-}t(0, r) = r$$
$$fact\text{-}t(n, r) = fact\text{-}t(n-1, n*r)$$

Récursivité terminale

Evaluation

$>$ (fact-t 4 1)
$\longrightarrow$ (fact-t 3 4)
$\longrightarrow$ (fact-t 2 12)
$\longrightarrow$ (fact-t 1 24)
$\longrightarrow$ (fact-t 0 24)
$\longrightarrow$ 24

Version fonctionnelle

```
(define (fact-t n r)
  (if (zero? n)
      r
      (fact-t (sub1 n)
              (* n r))))
```

Les valeurs successives de n sont utilisées dans les calculs qui sont effectués avant les appels récursifs. Il est inutile de les conserver dans une pile.

Les appels récursifs sont dits terminaux car aucun calcul n'est effectué après leur retour.

Récursivité terminale – Ordre des calculs

Pour rendre une fonction récursive terminale, on déplace le calcul effectué après l'appel récursif pour le faire avant. Ceci modifie l'ordre des calculs.

Quelles propriétés doivent être vérifiées par les opérateurs concernés pour que le résultat n'en soit pas affecté?

Commutativité | Associativité

Factorielle

- Version non récursive terminale
- (* 4 (* 3 (* 2 (* 1 1))))

- Version récursive terminale
- (* 1 (* 2 (* 3 (* 4 1))))

Exemple

> (- 4 (- 3 (- 2 (- 1 1))))
- ??

> (- 1 (- 2 (- 3 (- 4 1))))
- ??

> (- 1 2 3 4 1)
- -9

> (- (- (- (- 1 2) 3) 4) 1)
- -9

Récursivité terminale – Pb de paramètres

Notre fonction fact-t demande un paramètre supplémentaire qui doit être initialisé à 1. Pour garantir son bon fonctionnement, il faut définir deux fonctions, l'une faisant l'appel initial et l'autre le calcul récursif terminal. Il y a deux solutions possibles.

Deux fonctions globales

```
(define (factorielle n)
   (fact-t n 1))
```

```
(define (fact-t n r)
  (if (zero? n)
      r
      (fact-t (sub1 n)
              (* n r))))
```

Une fonction globale avec une fonction locale

```
(define (factorielle n)
  (letrec ((fact-t
            (lambda (n r)
               (if (zero? n)
                   r
                   (fact-t (sub1 n)
                           (* n r))))))
    (fact-t n 1)))
```

Quelle version préférez-vous? la 1ère La 2ième

Récursivité et continuation fonctionnelle

Notion de continuation

- Une continuation représente le futur d'un calcul, c'est-à dire le calcul à faire après le calcul d'une expression
- Une continuation est modélisée par une fonction qui s'applique au résultat d'une expression
- Elle représente un Goto fonctionnel

Comment construire la continuation d'un appel récursif?

- On effectue une abstraction de l'expression contenant le calcul récursif par une application : (k a)
- Le paramètre a de l'application est le résultat de l'appel récursif
- La fonction k est la continuation

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)) )))
```

Calcul Paramètre

((lambda(x) (* n x)) (fact (sub1 n)))

Récursivité et continuation fonctionnelle – Exemple

Continuations des appels récursifs de la fonction fact

$>$ (fact 4) ———- (lambda(x) x) – **identity**

$\longrightarrow$ (∗ 4 (fact 3)) ——— (lambda(x) (∗ 4 x))

$\longrightarrow$ (∗ 4 (∗ 3 (fact 2))) ——– (lambda(x) (∗ 4 (∗ 3 x)))

$\longrightarrow$ (∗ 4 (∗ 3 (∗ 2 (fact 1)))) ——- (lambda(x) (∗ 4 (∗ 3 (∗ 2 x))))

$\longrightarrow$ (∗ 4 (∗ 3 (∗ 2 (∗ 1 (fact 0))))) —— (lambda(x) (∗ 4 (∗ 3 (∗ 2 (∗ 1 x)))))

$\longrightarrow$ (∗ 4 (∗ 3 (∗ 2 (∗ 1 1)))) —— Application de la continuation à 1

$\longrightarrow$ (∗ 4 (∗ 3 (∗ 2 1)))

$\longrightarrow$ (∗ 4 (∗ 3 2))

$\longrightarrow$ (∗ 4 6)

$\longrightarrow$ 24

Récursivité et continuation fonctionnelle – Utilisation

On ajoute un paramètre pour représenter la continuation, qui doit être initialisé à l'identité

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```

```
(define (fact-c n k)
  (if (zero? n)
      (k 1)
      (fact-c (sub1 n)
              (lambda (x) (* n (k x))))))
```

> (fact-c 3 identity)

⟶ (fact-c 2 (lambda(x) (* 3 (**identity** x))))

⟶ (fact-c 1 (lambda(x) (* 2 ((**lambda**(x) (∗3 (**identity** x))) x))))

⟶ (fact-c 0 (lambda(x) (* 1 ((**lambda**(x) (∗2 ((**lambda**(x) (∗3 (**identity** x))) x))) x))))

⟶ ((lambda(x) (* 1 ((lambda(x) (* 2 ((lambda(x) (* 3 (identity x))) x))) x))) 1)

- ```
  ((lambda(x) (* 1
                 ((lambda(x) (* 2
                                ((lambda(x) (* 3
                                               (identity x)))
                                 x)))
                  x)))
   1)
  ```

Cette fonction est-elle récursive terminale? oui ou non
  ```

Récursivité terminale et continuation fonctionnelle

À votre avis, quand une fonction est déjà récursive terminale, la continuation a-t-elle une propriété particulière? oui non

La fonction fact-t

```
(define (fact-t n r)
  (if (zero? n)
      r
     (fact-t (sub1 n)
             (* n r))))
```

```
(define (fact-t-c n r k)
  (if (zero? n)
      (k r)
      (fact-t (sub1 n)
              (* n r)
              k)))
```

$>$ (fact-t 4 1)

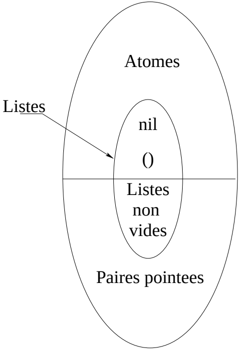
$\longrightarrow$ (fact-t 3 4) —— identity

$\longrightarrow$ (fact-t 2 12) —— identity

$\longrightarrow$ (fact-t 1 24) —— identity

$\longrightarrow$ (fact-t 0 24) —— identity

$\longrightarrow$ 24


Atomes
Listes
nil
()
Listes
non
vides
Paires pointees

Les objets

Les atomes

- Numériques,
- Booléens,
- Symboles,
- Chaînes de caractères
- Liste vide : ()
- Prédicat : **null** ?

Les paires pointées

- Constructeur : **cons**
- Accesseurs : car, cdr
  (argument paire pointée uniquement)
- Prédicats : pair ?, **cons**?
- Abbréviation : cddr, caadr, ..., cadddr, ..., list−ref

Forme quote et symboles

```
> (quote Pierre)
'Pierre
>'Pierre
'Pierre
> (define a 'Pierre)
> a
'Pierre
> Pierre
Error : Pierre undefined;
> '(define a 'Pierre)
'(define a 'Pierre)
```

Exemples

```
> (cons 1 2)
'(1 . 2)
> (cons (cons (cons 1 2) 3) 4)
'(((1 . 2) . 3) . 4)
> (cons 1 (cons 2 (cons 3 4)))
'(1 2 3 . 4)
> (car (cons 1 2))
1
> (cdr (cons 3 (cons 1 4)))
'(1 . 4)
```

Affichage des paires pointées

- (a . (pp)) $\longrightarrow$ (a pp) si pp est une paire pointée.
- (a . ()) $\longrightarrow$ (a)

Exemple

```
> '(1 . (2 3))
'(1 2 3)
> (define f '(define (f x) x))
> f
'(define (f x) x)
> (cons '(1 2) 3)
'((1 2) . 3)
```

Les listes

Définition récursive des listes Scheme

- Liste vide : '() ou **null**
- Une paire pointée dont le car est un élément de la liste, et le cdr est une liste;

Autres types de structures

- Liste impropre : une liste qui ne se termine pas par la liste vide.
- Liste circulaire : une chaîne de cons sans fin;

Ces autres types de structures ne sont pas des listes

Définitions formelles

Atome

atome ::= number | symbol | string | ()

Paire pointée

paire-pointée ::= (objet . objet)

Objet

objet ::= atome | paire-pointée

Liste

liste ::= () | (objet . liste)

Liste impropre

liste-impropre ::= () | paire-pointée

Fonctions de base sur les listes

- Prédicats **list** ?, empty? et **null**?
- Prédicats d'égalité : **eq**?, **equal**?
- Fonction de construction : **list** , (voir aussi **list** ∗), make−list
- Fonctions prédéfinies : **length**, list−ref , list−tail , **append**, **reverse**, **member**, **remove**, **first** , ... **tenth**, **nth**, **rest**, **last** , last−pair, take, drop, split−at, take−right, drop−right, split−at−right, flatten , **remove**∗, remove−duplicates, range, shuffle , permutations, remv, remq, memv, memq.
- Fonctions de a-listes : assq, **assoc**

Exemple

```
> (define s1 '(1 2 3))
> (define s2 '(1 2 3))
> (define s3 s1)
> (eq? s1 s2)
#f
```

Exemple

```
> (equal? s1 s2)
#t
> (eq? s1 s3)
#t
> (equal? s1 s3)
#t
```

Construction de listes

- quote : aucun élément de la liste n'est évalué

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

```
> '(1 (* 1 2) 3)
'(1 (* 1 2) 3)
```

- **list** : tous les éléments sont évalués

```
> (list 1 2 3)
'(1 2 3)
```

```
> (list 1 (* 1 2) 3)
'(1 2 3)
```

- **cons** : tous les arguments sont évalués (pour rappel)

```
> (cons (* 1 2) '(1 2 3))
'(2 1 2 3)
```

- **list** ∗ : (**list** ∗ 1 2 3) ⟶ '(1 2 . 3)
- make−list: (make−list 3 1) ⟶ '(1 1 1)
- range : intervalle

```
> (range 10)
'(0 1 2 3 4 5 6 7 8 9)
> (range 10 20 2)
'(10 12 14 16 18)
```

Fonction **append** : Concaténation de listes

- Fonction n-aire
- Les arguments sont des listes, sauf le dernier qui est un objet quelconque
- La dernière paire pointée de l'argument n est remplacée par la première de l'argument $n+1$
- Sans effets de bord : recopie des paires pointées de toutes les listes en argument (avec partage de tous leurs éléments), sauf le dernier argument qui est partagé.

Exemple

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

Filtrage de listes : fonction **remove**

La fonction **remove** prend en arguments un élément et une liste et elle renvoie la liste privée de la 1ère occurrence de l'élément. Elle admet un 3ième argument optionnel, qui est le prédicat de test de l'égalité. Par défaut c'est **equal?** qui est utilisé.

Exemple avec le prédicat par défaut

```
> (remove 2 '(1 2 3 2 4))
'(1 3 2 4)
> (remove* '(1 2) '(1 2 3 2 4)) ;Enleve toutes les occurrences
'(3 4)
```

Exemple avec des prédicats passés en argument

```
> (remove 2 '(1 2 3 4) >)
'(2 3 4)
> (remove 2 '(1 2 3 4) (lambda (x y) (not (= x y))))
```

Devinez le résultat : '(2 3 4) ou '(1) ?

Voir aussi remove-dupliquates, remv, remq

Appartenance à une liste : fonction **member**

La fonction **member** prend en arguments un élément e et une liste l et elle renvoie **#f** si e n'appartient pas à l, et la liste l privée de ses éléments jusqu'à l'occurrence de e si celui-ci apparaît dans la liste. Le prédicat d'égalité utilisé est **equal?**.

Exemples

```
> (member 2 '(1 2 3))
'(2 3)
> (member 3 '(1 2 . 3))
'(1 2 . 3) : not a proper list
> (member 2 '(1 2 . 3))
'(2 . 3)
> (member 5 '(1 2 3 4))
#f
```

- Quel est le résultat de cette expression : `(member 2 '(3 2 1 2 4 5 2))` ?

Voir aussi memv, memq, memf

Appartenance à une liste : fonction **member**

Exemple

```
> (member 3 '(3 2 1 2 4 5 2) >)
'(2 1 2 4 5 2)
> (member 3 '(3 2 1 2 4 5 2) <)
'(4 5 2)
```

Devinez le résultat de l'expression suivante :

```
> (member 2 '(3 2 1 2 4 5 2) >)
```

Programmation récursive sur les listes

Définition récursive

- Soit liste vide
- Soit une paire pointée dont le *car* est un élément de la liste et le *cdr* la suite de la liste, soit la liste privée de son premier élément.

Spécification récursive d'un calcul

- Si la liste est vide, calculer le résultat correspondant.
- Sinon, exprimer le calcul en fonction de l'elément courant et du résultat d'un appel récursif sur la liste privée de son premier élément.

Somme des éléments d'une liste

Équations fonctionnelles :

- somme-liste(null) = 0
- somme-liste(l) = car(l) + somme-liste(cdr(l))

```
(define (somme-liste l)
   (if (empty? l)
       0
       (+ (car l)
          (somme-liste (cdr l)))))
```

Programmation récursive sur les listes : append et reverse

Concaténation de deux listes

concat(null, l2) = l2

concat(l1, l2) = cons(car(l1), concat(cdr(l1), l2))

```
(define (concat l1 l2)
  (if (null? l1)
      l2
      (cons (car l1)
            (concat (cdr l1) l2))))
```

Cette fonction est-elle linéaire? oui ou non

Inversion d'une liste

inversion(null, l) = l

inversion(l1, l1) = inversion(cdr(l1)), cons(car(l1), l2)))

```
(define (inversion l1 l2)
  (if (null? l1)
      l2
      (inversion (cdr l1)) (cons (car l1) l2)))
```

Cette fonction est-elle récursive terminale? oui ou non

Programmation récursive sur les listes : **cons** et récursion terminale

iota1 récursive

iota1(0) = '()
iota1(n) = cons(n, iota1(n-1))

```
(define (iota1 n)
  (if (zero? n)
      '()
      (cons n
            (iota1 (sub1 n)))))
```

iota2 récursive terminale

iota2(0, l) = l
iota2(n, l) = iota2(n-1, cons(n, l))

```
(define (iota2 n l)
    (if (zero? n)
      l
      (iota2 (sub1 n)
             (cons n l))))
```

En inversant l'ordre d'un **cons** *et d'un appel récursif, on inverse l'ordre de la liste résultat – l'opération cons n'est ni associative ni commutative*

Les listes d'association : a-listes

Définition

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

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 au sens de *equal*? à la clef, et $\#f$ sinon.
- La fonction assq réalise le même travail avec *eq*? (assv pour *eqv*)

Les listes d'association : a-listes

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 au sens de *equal*? à la clef, et #*f* sinon.
- La fonction assq réalise le même travail avec *eq*? (assv pour *eqv*)

```
>(define *carnet*
  '((Pierre "pierre@labri.fr")
   (Laure "Durand@laposte.net")
   (Laura "laura@twinpeaks.fr")))
> (assoc 'Laure *carnet*)
'(LAURE "Durand@laposte.net")
> (cdr (assoc 'Laure *carnet*))
'("Durand@laposte.net")
> (cadr (assoc 'Laure *carnet*))
"Durand@laposte.net"
> (assoc 'hell-boy *carnet*)
#f
```

```
(define (add-alist1 a l)
   (cond ((assoc a l) l)
         (else (cons a l))))
```

Programmation récursive sur les objets scheme

Objet scheme – Expression symbolique

- Soit un atome
- Soit une paire pointée admettant pour *car* et pour *cdr* un objet scheme (un atome ou une paire pointée)

Spécification récursive d'un calcul

- Si l'objet est un atome, calculer le résultat correspondant.
- Sinon, exprimer le calcul en fonction du fils gauche (le *car*), et du fils droit (le *cdr*).

```
(define (what a)
   (cond
     ((not (pair? a)) 1)
     (else
       (+ (what (car a))
          (what (cdr a))))))
```

Programmation récursive sur les arbres

Arbre binaire

- Soit l'arbre vide (null)
- Soit une feuille (atome non null)
- Soit une paire pointée admettant au moins un arbre non vide pour *car* ou pour *cdr*.

Spécification récursive d'un calcul

- Si l'arbre est vide, calculer le résultat correspondant.
- Si l'arbre est une feuille, calculer le résultat correspondant.
- Sinon, exprimer le calcul en fonction du fils gauche (le *car*), et du fils droit (le *cdr*).

```
(define (yawhat a)
   (cond
     ((null? a) 0)
     ((not (pair? a)) 1)
     (else
       (+ (yawhat (car a))
          (yawhat (cdr a))))))
```

Exercice : longueur récursive de listes de listes

L'argument est supposé être toujours une liste, donc son cdr est aussi une liste

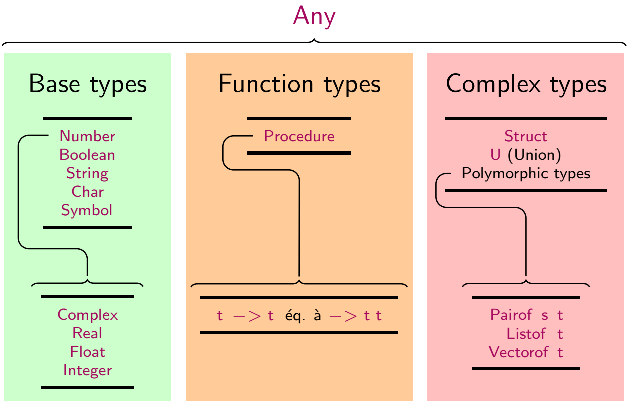
Equations récursives

- length*('()) = 0
- length*(cons(cons(a , r1), r2)) = length*(car(l)) + length*(r2)
- length*(cons(a , r)) = 1 + length*(r)

```
(define (yalength* l)
  (cond
    ((null? l) 0)
    (else
      (+ (if (pair?(car l))
             (yalength* (car l))
             1)
         (yalength* (cdr l))))))
```

```
(define (length* l)
  (match l
    ('() 0)
    ((cons (cons a r1) r2)
     (+ (length* (car l))
        (length* (cdr l))))
    ((cons a r)
     (add1 (length* (cdr l))))))
```

Classification des valeurs en ensembles appelés types de manière à garantir la correction de certains programmes.



Styles de typage

Styles de typage

- le typage dynamique : déterminé pendant l'exécution par le runtime, il ne nécessite aucune intervention du programmeur;
- le typage statique : fixé avant l'exécution par le compilateur, il est soit inféré automatiquement, soit indiqué par des annotations dans le code.

Racket définit en fait plusieurs langages :

- En Racket classique, le typage est dynamique;
- En Typed/Racket, le typage est dynamique mais autorise des annotations statiques vérifiées par le compilateur.

Comment typer ?

Afin d'annoter une valeur $\langle val \rangle$ par un type $\langle typ \rangle$, il suffit d'écrire *avant la définition* de $\langle val \rangle$:

- (: $\langle val \rangle$ $\langle typ \rangle$) Ex : `(: int_exm Integer)`

Les types primitifs contiennent en particulier : Number, Integer, Float, Char, String, . . .

Les types des fonctions sont écrits à l'aide d'une des syntaxes suivantes :

- ($\langle arg_1 \rangle$ $\langle arg_2 \rangle$ ... $\langle arg_n \rangle$ −> $\langle res \rangle$) Ex : `(Number Number -> Boolean)`
- (−> $\langle arg_1 \rangle$ $\langle arg_2 \rangle$ ... $\langle arg_n \rangle$ $\langle res \rangle$) Ex : `(-> Number Number Boolean)`

Exemple

```
#lang typed/racket                  ;; Choice of the language
(: num-fun (Number -> Number))      ;; Type annotation of num-fun
(define (num-fun n) (add1 n))       ;; Definition of num-fun
(:print-type num-fun)               ;; (-> Number Number)
```

Intérêts du typage

- Détection d'erreurs de type : passer une valeur de type String à une fonction Int −> Int est incohérent.

Exemple

```
(: num-fun (Number -> Number))
(define (num-fun n) (add1 n))
(num-fun "abc")     ;; -> Type Checker error : type mismatch
```

- Compatibilités de type : passer une valeur de type Integer à une fonction Number −> Number est cohérent car un Integer est aussi un Number.

Exemple

```
(: intg Integer)  ;; Type annotation of intg
(define intg 3)
(num-fun intg)    ;; -> 4
```

- Optimisations : le compilateur peut écrire du code dédié à des types particuliers (exemple : nombre flottants et instructions FPU).

Exemples

Exemple

```
(: greater-than (Number Number -> Boolean))
(define (greater-than x y)
  (> x y))          ;; Type error : Number cannot be compared
```

En effet, un Number peut aussi être un Complex, donc non comparable.

Exemple

```
(: plus (Number Number -> Number))
(define (plus x y) (+ x y))

(: greater-than (Real Real -> Boolean))
(define (greater-than x y) (> x y))

(greater-than (plus 3 4) 5)
;; Type error : Number cannot be compared
```

En effet, plus renvoie un Number, qui potentiellement n'est pas un Real.

Remarque : en réalité, le type de l'opérateur + est générique.

Définir ses propres types

Le code suivant définit un Struct représentant des points du plan :

```
(struct: point ([x : Real] [y : Real]))

(: distance (point point -> Real))
(define (distance p1 p2)
  (sqrt (+ (sqr (- (point-x p2) (point-x p1)))
           (sqr (- (point-y p2) (point-y p1))))))
```

Cette construction définit en même temps les fonctions suivantes :

- Un constructeur point permettant de construire des instances comme par exemple (point 3 4).
- Deux accesseurs point−x et point−y permettant d'accéder aux champs de la structure.

Types inductifs : les listes

Définition récursive des listes en Scheme (Rappel)

- Soit la liste vide : **null** ou '()
- Soit une paire (car, cdr) où car est un élément de la liste et cdr est une liste.

La définition de type pour les listes en Racket :

```
;; A List is either a Pair or Empty
(define-type (List a) (U (Pair a) Empty))
;; A Pair is a struct with car and cdr, and Empty is empty
(struct: (a) Pair ([car : a] [cdr : (List a)]))
(struct: Empty ())
```

Remarque : le type pour les listes est un type polymorphe.

```
(: a_list (List Integer))
(define a_list (Pair 1 (Pair 2 (Pair 3 (Empty)))))
```

Reconnaissance de motif

La forme match

```
(match t
   [⟨pat1⟩  res1]
   [⟨pat2⟩  res2]
   ...
   [⟨patn⟩  resn]
   [_        default]
)
```

La reconnaissance de motif ou pattern-matching :

- compare l'expression t à chacun des motifs $\langle pat_k \rangle$
- et renvoie le résultat associé au premier indice pour lequel t correspond.

Les motifs peuvent introduire des liaisons utilisées dans le résultat.

Exemple pour calculer la longueur d'une liste :

```
(define (list-length l)
  (match l
    [(Empty)     0]                ;; Match Empty struct
    [(Pair x xs)                   ;; Match Pair struct
       (add1 (list-length xs))]    ;; and binds x and xs
    ))
```

Reconnaissance de motif

La forme match peut être utilisée en dehors de typed/racket

Somme des éléments d'une liste

Équations fonctionnelles :

- somme-liste('()) = 0
- somme-liste(l) = car(l) + somme-liste(cdr(l))

- somme-liste('()) = 0
- somme-liste(cons(n, r)) = n + somme-liste(r))

Exemple

```
(define (somme-liste l)
  (match l
    ('() 0)
    ((cons n r) (+ n (somme-liste r)))))
```

Chapitre 7 - Les fonctionnelles

Higher-order functions

- Fonctions anonymes : forme **lambda**
- Fonctionnelles de la bibliothèque : arguments fonctions
- Fonctionnelles de la bibliothèque : fonctions en retour

La forme **lambda** : rappel et utilisation

(**lambda** ($\langle p_1 \rangle$ $\langle p_2 \rangle$... $\langle p_n \rangle$) $\langle e \rangle$)

- **Nommage de λ-expressions** : `(define f (lambda (x y) (+ (* 10 x) y))`
  Équivalent à : `(define (f x y) (+ (* 10 x) y))`
- **Application de λ-expressions** : mise en position fonctionnelle, stratégie applicative (par valeur) – Ex. `((lambda (x) (sub1 x)) 1)`
- **Passage de λ-expressions en paramètres** : juxtaposition
- **λ-expressions en retour de fonction** : imbrication

Application de λ-abstractions : juxtaposition

Rappels sur les λ-termes

- une variable : x, y, ...
- une application : u v où u et v sont des λ-termes
- une λ-abstraction : $\lambda x.u$

On appelle rédex un terme de la forme $(\lambda x.u)v$. On définit alors la bêta-réduction

$$(\lambda x.u)v \longrightarrow u[x := v]$$

Applications d'une λ-abstraction à une λ-abstraction

- Soit le terme :$(\lambda x.xy)(\lambda x.ux)$, on a la suite de réductions suivante : $xy[x := \lambda x.ux] \longrightarrow (\lambda x.ux)y \longrightarrow ux[x := y] \longrightarrow uy$
- Soit $(\lambda f.f0)(\lambda x.(*2x))$, on a la suite de réductions suivante : $f0[f := \lambda x.(*\ 2\ x)] \longrightarrow (\lambda x.(*\ 2\ x))0 \longrightarrow (*\ 2\ x)[x := 0] \longrightarrow (*\ 2\ 0) \longrightarrow 0$
- En scheme : `((lambda(f) (f 0)) (lambda (x) (* 2 x)))`, on a la suite de réductions suivante (stratégie applicative) :

```
((lambda(x) (* 2 x)) 0) -> (* 2 0) -> 0
```

λ-expressions en retour de fonction : imbrication

Exemples en λ-calcul

- Soit $\lambda x.(\lambda y.xy)f$, on a $\lambda y.xy[x := f] \longrightarrow \lambda y.fy$
- Soit $(\lambda x.(\lambda y.xy)f)z$, on a $(\lambda y.xy[x := f])z \longrightarrow (\lambda y.fy)z \longrightarrow fy[y := z] \longrightarrow fz$

Exemples en scheme

```
> ((lambda(x) (lambda (y) (= (* 2 x) y))) 1)
(lambda (y) (= (* 2 1) y))


> (((lambda(x) (lambda (y) (= (* 2 x) y))) 1) 2)
-> ((lambda (y) (= (* 2 1) y) 2)
-> (= (* 2 1) 2)
#t
```

Méthode : pour effectuer une réduction, repérer la fonction qui est en première position, puis les arguments e_1, e_1, etc.. Évaluer les arguments puis appliquer la fonction.

Fonctionnelles en Python

Fonctions anonymes

$\lambda x.x$ s'écrit `lambda x: x`

Exemple

```
delirius: python
Python 2.7.10 (default, May 27 2015, 18:11:38)
[GCC 5.1.1 20150422 (Red Hat 5.1.1-1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.

>>> sentence = 'All work and no play makes Jack a dull boy'
>>> words = sentence.split()
>>> print words
['All', 'work', 'and', 'no', 'play', 'makes', 'Jack', 'a', 'dull', 'boy']
>>> lengths = map( lambda word: len(word) , words)
>>> print lengths
[3, 4, 3, 2, 4, 5, 4, 1, 4, 3]
>>> f = lambda word: len(word)
>>> lengths = map(f, words)
```

Fonctionnelles en Javascript

Fonctions anonymes

```
function(message) {
  alert(message);
}
```

En scheme

```
(lambda (message)
   (alert message)))
```

Fonctionnelles

```
function ajouteur(nb) {
  return function (val) {
           return val + nb;
         }
 }
var ajoute10 = ajouteur(10);
ajoute10(1); //retourne 11
```

En scheme

```
(define (ajouteur nb)
  (lambda (val) (+ val nb)))

(define ajoute10 (ajouteur 10))

> (ajoute10 1)
11
```

Fonctionnelles en Common Lisp

Common Lisp

Il y a deux espaces de noms, un pour les variables et un pour les fonctions. Un symbole est représenté par une structure à plusieurs champs, l'un d'eux représente la valeur de variable et un autre la valeur de fonction. Lors de l'évaluation le système doit sélectionner une des deux valeurs. Son choix dépend du contexte : si le symbole est en position fonctionnelle il accède à sa valeur de fonction, en toute autre situation, il accède à sa valeur de variable.

- Quand on souhaite faire passer une fonction en paramètre, il faut indiquer au système d'utiliser la valeur de fonction et non la valeur de variable (#')
- Quand on veut appliquer une fonction renvoyée en résultat de l'application d'une fonction, il faut indiquer au système d'utiliser la valeur de variable et non la valeur de fonction (funcall)

```
> (defun ajouteur(nb) (lambda (val) (+ val nb)))
> (defparameter ajoute10 (ajouteur 10))
> (funcall ajoute10 1)
1
```

Fonctionnelles en C#

Fonctions anonymes

- Notation C# : `(x, y) => x+y`
- En scheme : `(lambda(x y) (+ x y))`

Fonctionnelles

```
double ajouteur (double nb) {
   return (val) ⇒ val+nb;
}
Myfunction ajoute10 = ajouteur (10);
double y = ajoute10(1);
```

Avec

```
double Myfunction (double x); signature fonctionnelle
```

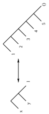
Remarques :

- C# est un langage typé statiquement, donc il faut déclarer les types des fonctions dynamiques.
- On peut aussi composer des fonctions en C#

Fonctions à nombre d'arguments variable : λ-expressions

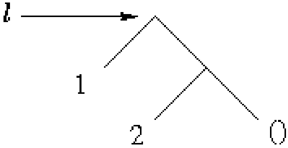
Notation pointée

```
> ((lambda (x y . l) (list x y l)) 1 2 3 4 5)
 '(1 2 (3 4 5))
```



Paramètres en liste

```
> ((lambda l l) 1 2)
 '(1 2)
(define g (lambda l l))
> (g 1 2 3 4)
 '(1 2 3 4)
```



Fonctions nommées avec nombre d'arguments variables

Notation pointée

```
> (define (f x y . l) (list x y l))
> (f 1 2 3 4 5)
'(1 2 (3 4 5))
```

Paramètres en liste

```
> (define (h . l) l)
> (h 1 2 3 4)
'(1 2 3 4)
```

Fonctions à plusieurs résultats

On utilise la forme `values` pour créer plusieurs résultats.

```
> (define (2r x y) (values (+ x y) (- x y)))
> (2r 1 2)
3
-1
```

Fonctionnelles de la bibliothèque : arguments fonctions

- **Appartenance** : (memf ⟨*proc*⟩ ⟨*lst*⟩)
- **Filtrage** : (filter ⟨*pred*⟩ ⟨*lst*⟩)
- **Liste d'associations** :
  (**assoc** ⟨*v*⟩ ⟨*lst*⟩)
  (**assoc** ⟨*v*⟩ ⟨*lst*⟩ ⟨*pred*⟩)
  (assf ⟨*proc*⟩ ⟨*lst*⟩)
- **Constructeur** : (build−list ⟨*n*⟩ ⟨*proc*⟩)
- **Itération sur des listes** : **map**, **apply**, andmap, ormap, foldl , foldr

Exemple

```
>(memf (lambda (arg) (> arg 9)) '(7 1 9 10 3))
'(10 3)

> (filter positive? '(1 -2 3 4 -5))
'(1 3 4)

> (assf (lambda (arg) (> arg 2)) (list (list 1 2) (list 3 4) (list 5 6)))
'(3 4)

> (build-list 10 list)
'((0) (1) (2) (3) (4) (5) (6) (7) (8) (9))
```

Itération : la forme **map**

Définiton

La forme **map** prend une fonction `f` et n listes en arguments ($n > 0$), où n est l'arité de la fonction `f`. Soit $l_1 = (\langle a_1 \rangle \ \langle a_2 \rangle ... \langle a_n \rangle)$, et $l_2 = (\langle b_1 \rangle \ \langle b_2 \rangle ... \langle b_n \rangle)$.

- Cas d'une fonction unaire :
  $(\texttt{map} \ \langle f \rangle \ \langle l_1 \rangle) \longrightarrow ((\langle f \rangle \ \langle a_1 \rangle)(\langle f \rangle \ \langle a_2 \rangle) \ldots (\langle f \rangle \ \langle a_n \rangle))$
- Cas d'une fonction n-aire :
  $(\texttt{map} \ \langle f \rangle \ \langle l_1 \rangle \ \langle l_2 \rangle \ \ldots) \longrightarrow ((\langle f \rangle \ \langle a_1 \rangle \ \langle b_1 \rangle ..)(\langle f \rangle \ \langle a_2 \rangle ..)..(\langle f \rangle \ \langle a_n \rangle ..))$

Exemple

```
> (map sub1 '(1 2 3 4 5))
 '(0 1 2 3 4)
> (map cons '(1 2 3 4) '(a b c d))
 '((1 . a) (2 . b) (3 . c) (4 . d))
> (map (lambda(x) (list (add1 x))) '(1 2 3))
 '((2) (3) (4))
```

Remarques : Les listes doivent avoir le même nombre d'arguments, le premier argument doit impérativement être une fonction et non une macro.

Itérations logiques : formes andmap et ormap

Forme andmap

Cette forme a la même signature que la forme **map**. Elle applique la fonction aux éléments de la liste dans l'ordre. Le résultat est celui de la dernière application, pas de mise en liste. S'arrête au premier résultat faux.

```
> (andmap positive? '(1 2 3))
#t
> (andmap + '(1 2 3) '(4 5 6))
9
```

Forme ormap

Comme la forme andmap mais renvoie le premier vrai.

```
> (ormap eq? '(a b c) '(a b c))
#t
> (ormap positive? '(1 2 a))
#t
> (ormap + '(1 2 3) '(4 5 6))
5
```

Itérations générales : formes foldl et foldr

Comme la forme **map**, les formes fold appliquent une fonction aux éléments d'une ou plusieurs listes. Alors que **map** combine les résultats obtenus dans une liste, les formes fold les combinent d'une façon déterminée par leur paramètre fonctionnel `f`. Elles appliquent `f` aux éléments des listes de gauche à droite ou bien de droite à gauche. L'argument `init` est utilisé pour terminer la combinaison récursive du résultat.

- **Calcul de gauche à droite** : `(foldl f init list)` = $(\mathbf{f}\ e_n(\mathbf{f}\ e_{n-1}(\mathbf{f}\ldots(\mathbf{f}\ e_1\ init))))$
  avec list = $(e_1\ e_2 \ldots e_n)$

```
> (foldl cons '() '(1 2 3))
(cons 3 (cons 2 (cons 1 '())))
'(3 2 1)
> (foldl * 1 '(1 2 3))
(* 3 (* 2 (* 1 1)))
6
> (foldl (lambda (x y) (+ (sqr x) y)) 0 '(1 2 3))
14
```

- **Calcul de droite à gauche** : `(foldr f init list)` = $(\mathbf{f}\ e_1(\mathbf{f}\ e_2(\mathbf{f}\ldots(\mathbf{f}\ e_n\ init))))$
  avec list = $(e_1\ e_2 \ldots e_n)$

```
>  (foldr cons '() '(1 2 3))
(cons 1 (cons 2 (cons 3 '())))
'(1 2 3)
>  (foldr cons '(1 2 3) '(3 4 5))
(cons 3 (cons 4 (cons 5 '(1 2 3))))
'(3 4 5 1 2 3)
```

Évaluation applicative avec **eval** et **apply**

Évaluation : (eval sexpr)

- **sexpr** : expression symbolique

- Si **sexpr** est autoévaluante, renvoyer **sexpr**
- Si **sexpr** est un symbole, alors
  - Rechercher une liaison définissant **sexpr** dans l'environnement courant et renvoyer la valeur associée.
- Si **sexpr** est une liste
  - Calculer (eval (car **sexpr**)). Soit **f** la fonction résultat.
  - Calculer (eval **ei**), pour tout élément **ei** de (cdr **sexpr**). Soit **v** la liste des résultats.
- Calculer (apply **f** **v**)

Application : (apply f v)

- **f** : fonction à appliquer
- **v** : liste des valeurs des arguments

- Soient **e** l'environnement lexical de **f**, **lf** la liste des paramètres formels, et **expr** le corps de la fonction.
- Construire l'environnement local **e-local** constitué des liaisons entre les paramètres formels de **lf** et les valeurs correspondantes dans **v**.
- Calculer : (eval **expr** (cons **e-local** **e**)).

Application : la forme **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.

$$(\texttt{apply}\ \langle f \rangle \langle l \rangle) = (\langle f \rangle\ \langle e_1 \rangle\ \langle e_2 \rangle ... \langle e_n \rangle)$$

avec $\langle l \rangle = (\langle e_1 \rangle\ \langle e_2 \rangle ... \langle e_n \rangle)$

Exemple

```
> (apply + '(1 2 3)); -> (+ 1 2 3)
6
> (apply + '()); -> (+)
0
> (apply + 1 2 '(3 4)); -> (+ 1 2 3 4)
10
```

Cas d'utilisation : fonctions à nb d'arguments variable

```
(define (moyenne-carre . l)
  (if (null? l)
      0
      (/ (apply + (map sqr l)) (length l))))
> (moyenne-carre)
0
```

```
(define (iota n . l)
  (if (zero? n)
      (cons n l)
      (apply iota (sub1 n) (cons n l))))
> (iota 4)
'(0 1 2 3 4)
```

Exemple de fonction n-aire avec apply

La fonction ou n-aire

```
(define (ou . l)
   (cond ((null? l) #f)
         ((car l))
         (else (apply ou (cdr l)))))
```

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.

Fonctions en retour de fonctions : composition

En scheme, il est possible de manipuler et de créer des fonctions dynamiquement au moyen d'expressions.

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 $h = gof$.

Opérations de composition

- **Fonctions unaires** : (compose1 $\langle proc_1 \rangle$ $\langle proc_2 \rangle$ ... $\langle proc_n \rangle$)
- **Arité quelconque** : (compose $\langle proc_1 \rangle$ $\langle proc_2 \rangle$ ...$\langle proc_n \rangle$)
  Le nombre de résultats de $\langle proc_i \rangle$ doit correspondre à l'arité de $\langle proc_{i-1} \rangle$

Exemple

```
> ((compose1 sqrt add1) 1) ; (sqrt (add1 1))
1.414213562
> (define (2r x y) (values (+ x y) (- x y)))
> ((compose list 2r) 1 2) ; (list (2r 1 2))
'(3 -1)
```

Fonctions en retour de fonctions : curryfication

La fonction curry curryfie son argument. Soit une fonction

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

la curryfication lui associe la fonction suivante :

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

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

Haskell B. Curry

Définition de fonctions currifiées

```
> (define *2 ((curry *) 2))
> (*2 3)
6
> (define *curried (curry *))
> (*curried 2)
#<procedure:curried>
> ((*curried 2) 3)
6
```

Construction de fonctions curryfiées à la volée

```
> (((curry list) 1) 2)
'(1 2)
> (((curry list) 1 2) 2)
'(1 2 2)
```

Cas d'utilisation de la curryfication

On va curryfier la fonction filter de la bibliothèque pour la spécialiser sur le prédicat even?

```
> (filter even? '(1 2 3 4))
'(2 4)
> (define filter-even ((curry filter) even?))
> (filter-even '(1 2 3 4))
'(2 4)
```

Cela peut permettre de l'utiliser dans une fonctionnelle de liste comme map par exemple.

Exemple

```
> (map filter-even '((1 2 3 4) (3 6 2 4 23 1)))
'((2 4) (6 2 4))
```

Programmation de fonctionnelles : composition

Composition de deux fonctions

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

Exemple

```
> (o sqr sub1)
#<procedure>
> ((o sqr sub1) 2)
1
> (define sqr1- (o sqr sub1))
> (sqr1- 2)
1
```

Remarque : écriture équivalente

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

Programmation de fonctionnelles : curryfication

Curryfication d'une fonction binaire

$$curry1 \ : (A \times B \longrightarrow C) \longrightarrow (A \longrightarrow (B \longrightarrow C))$$

telle que pour $f \in (A \times B \longrightarrow C)$ soit $f^c = curry1(f) \in (A \longrightarrow (B \longrightarrow C))$, on a $f(x, y) = (f^c(x))(y)$

```
(define (curry1 f) (lambda (x) (lambda (y) (f x y))))
```

On utilise la fonction map avec deux arguments, c'est-à-dire avec un argument fonctionnel unaire pour premier argument et une liste pour deuxième argument.

```
> (curry1 map)
#<procedure>
> ((curry1 map) list)
#<procedure>
> (define map-list ((curry1 map) list))
> (map-list '(1 2 3 4))
'((1) (2) (3) (4))
```

Remarque : écriture équivalente

```
(define curry1 (lambda (f) (lambda (x) (lambda (y) (f x y)))))
```

Notion de fermeture (closure)

Définitions

- L'environnement lexical d'une fonction est l'environnement dans lequel elle est définie.
- Une fermeture (closure en anglais) est la représentation d'une fonction sous forme d'un couple associant l'environnement lexical et le code de la fonction.
- En Scheme les fonctions sont représentées par des fermetures pour conserver leur environnement de définition contenant des références éventuelles (ce n'est pas le cas par exemple du langage emacs-lisp).
- Les fermetures peuvent être utilisées pour représenter des états, par modification de l'environnement (voir chapitre suivant).

Par exemple pour l'application suivante de la fonction de composition :

```
> (define carre-1 (o sqr sub1))
```

La fermeture représentant carre-1 est la suivante :

$$[((f \;.\; \mathrm{sub1})\; (g \;.\; \mathrm{sqr}))\; (\mathrm{lambda}(x)\; (g\; (f\; x)))]$$

Les liaisons définissant les variables libres f et g de carre-1 dans l'environnement lexical de la fermeture permettent de conserver les valeurs qui ont été données lors de l'application de la fonctionnelle.

Emacs-lisp

- Emacs-lisp n'a pas de fermetures, les fonctions ne sont représentées que par leur code (sans l'environnement lexical). Soient les fonctions :

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

- Lors de l'application :

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

- On obtient :

```
Debugger entered--Lisp error: (void-variable g)
  (funcall g (funcall f x))
  (lambda (x) (funcall g (funcall f x)))(2)
  funcall((lambda (x) (funcall g (funcall f x))) 2)
  eval((funcall (o (function carre) (function 1-)) 2) nil)
```

Programmation de fonctionnelles récursives

Première ébauche d'écriture d'une fonction de composition d'une liste de fonctions : ici, tout le calcul est gelé par la λ-expression. Il s'éxecutera entièrement au moment de l'application de la fonction résultat.

Exemple

```
(define (bad1 . l)
  (lambda(x)
    (if (null? l)
        x
        ((car l) ((apply bad1 (cdr l)) x)))))

> (define add2 (bad1 add1 add1))
(lambda(x) ; le resultat est la lambda-expression
  (if (null? '(add1 add1))
      x
      (add1 ((bad1 add1) x))))
```

Solution récursive terminale

Pour être sûr de mettre l'appel récursif dans la fonctionnelle, il faut rendre celle-ci récursive terminale.

```
(define (o f . l)
  (if (null? l)
      f
      (apply o (lambda(x) (f ((car l) x)))
               (cdr l))))
```

Lors de l'application de la fonctionnelle **o**, tous les calculs s'effectuent, sauf ceux qui nécessitent de connaître l'argument de la fonction résultat.

Conclusion

Règles d'écriture

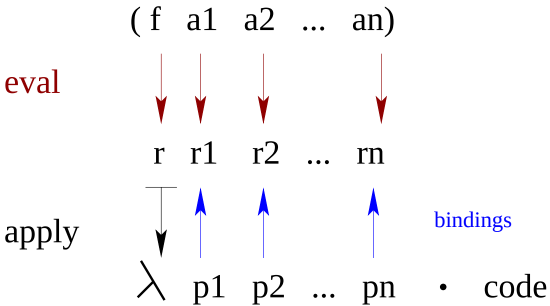
- L'appel récursif doit être effectué par la fonctionnelle plutôt que par la fonction résultat. De cette façon, on effectue la boucle une seule fois au moment de la construction, sinon, la boucle est effectuée à chaque application de la fonction résultat.
- Pour ne pas geler l'appel récursif de la fonctionnelle, il faut qu'il soit extérieur à toute fermeture (λ-expression), ce qui implique de constuire les fermetures en arguments plutôt qu'en valeurs de retour d'appels récursifs, et donc de rendre les fonctionnelles récursives terminales.

Références

Une référence est un objet correspondant à une adresse mémoire et dont l'indirection est faite automatiquement dans toute situation où une valeur est requise. L'adresse associée à une référence n'est pas directement manipulable en tant que telle (il n'existe pas d'opérations pour le programmeur sur les références)

- Un symbole est lié à une référence, correspondant à un atome ou une paire pointée.
- L'évaluation d'un symbole renvoie une référence vers sa valeur.
- La référence est utilisée partout où la valeur n'est pas requise.
- On trouve des références dans d'autres langages : Java, C++.

Passage d'arguments



Passage d'arguments

Soit f une fonction, soient p1, p2, ... pn ses paramètres formels. Soit l'application :

$$(f\ a1\ a2\ ...\ an)$$

Soient r1, r2, ..., rn les références vers les résultats des évaluations respectives des arguments a1, a2, ..., an.
Lors de l'application, un environnement local est construit. Il est constitué des liaisons entre les paramètres formels pi de la fonction f et les références ri des arguments de l'application.

$$((p1\ .\ r1)(p2\ .\ r2)...(pn\ .\ rn))$$

Les références r1, r2, ..., rn sont utilisées comme des valeurs à travers les symboles p1, p2, ..., pn, les indirections étant effectuées automatiquement. Ainsi, il est impossible de modifier un paramètre pi, car la modification reste locale à cet environnement.

L'affectation

La forme **set!**

$$(\textbf{set!}\ \langle id \rangle\ \langle e \rangle)$$

- La référence associée à l'identificateur $\langle id \rangle$ est remplacée par la référence du résultat de l'évaluation de l'expression $\langle e \rangle$.
- La valeur de retour de l'affectation est la valeur $\# < void >$ que la fonction `read` n'affiche pas. La procédure `void` rend ce même résultat en prenant un nombre quelconque d'arguments.

Modification de paires pointées

On ne peut pas modifier les paires pointées de base dans la norme scheme. En Racket, il faut utiliser le paquetage `mpair`

Exemple

```
> (define mp (mcons 1 2))
> (set-mcar! mp 2)
> mp
(mcons 2 2)
```

Modification de paramètres

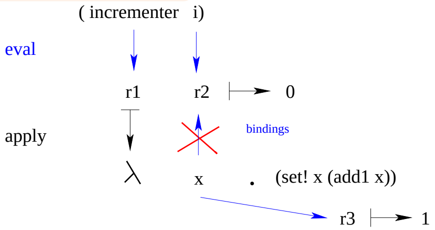
On se donne la session suivante :

```
(define (incrementer x)
  (set! x (add1 x)))
> (incrementer 2)
> (define i 0)
> (incrementer i)
```

Quel est le résultat de cette expression?

0 ou **1**

```
> i
```



Listes mutables circulaires

Rappel sur la concaténation

```
> (define l '(1 2 3))
> (append l l)
'(1 2 3 1 2 3)
> l
'(1 2 3)
```

Fonction rendant une liste mutable circulaire

```
(define (cirlist ml)
   (letrec ((aux (lambda(l)
                   (if (null? (mcdr l))
                       (set-mcdr! l ml)
                       (aux (mcdr l))))))
     (if (null? ml)
         ml
         (aux ml))))
> (define ml (mcons 1 (mcons 2 '())))
> (cirlist ml)
> ml
#0=(mcons 1 (mcons 2 #0#))
```

Blocs d'expressions

Certaines expressions pouvant effectuer des effets de bord, il devient possible de les mettre en séquence. Contrairement aux formes let et lambda, certaines formes, telles le if nécessitent d'utiliser une forme spéciale de mise en séquence.

La forme begin

(begin $\langle e_1 \rangle \langle e_2 \rangle ... \langle e_n \rangle$)

- Chaque expression ei est évaluée selon son ordre d'apparition.
- Le résultat de l'évaluation de la séquence est celui de la dernière.
- Les valeurs des évaluations des expressions précédentes sont perdues.
- Il existe une forme **begin0** qui renvoie le résultat de la première expression de la séquence.

```
(let ((tmp 0))
  (if (< x y)
      (begin (set! tmp x)
             (set! x y)
             (set! y tmp)
             x)
      y))
```

Fermetures et affectations : en Common Lisp

On peut utiliser les fermetures pour modéliser des états.

Générateurs en Common Lisp

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

Exemple

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

Fermetures et affectations : et en Scheme?

- Quel est le résultat de la session suivante? 1 ou Erreur

```
(let ((x 1))
  (define (f)
    x)
   (f))
1
> (f)
```

- Quel est le résultat de la session suivante? 1 ou 0 ou erreur

```
(define (make-f)
  (let ((x 1))
    (lambda () x)))
> (define e (make-f))
> (define x 0)
> (e)
```

Fermetures et affectations : générateurs en Scheme

Exemple

```
(define (make-int-gen n)
  (let ((i n))
    (lambda ()
      (set! i (add1 i))
      i)))
> (define int-gen0 (make-int-gen -1))
> (int-gen0)
0
> (int-gen0)
1
> (int-gen0)
2
```

Les mémo-fonctions (memo functions, memoization)

La technique des mémo-fonctions est utilisée pour optimiser le calcul des fonctions, en mémorisant des résultats d'appels coûteux.

Suite de Fibonacci

```
(define (make-memo-fib) ; Creation de la fermeture
  ; Initialisation de la table dans l'environnement lexical
  (let ((memo-table '((1 . 1) (2 . 1))))
    (define (memo-fib n) ; definition de la fonction
      ; Recherche dans la table
      (let ((computed-value (assoc n memo-table)))
        (if computed-value
            (cdr computed-value) ; la valeur est trouvee
            ;; La valeur est calculee et stockee
            (let ((new-value (+ (memo-fib (sub1 n)) ; calcul
                                (memo-fib (- n 2)))))
              (set! memo-table ; stockage
                    (cons (cons n new-value)
                          memo-table))
              new-value)))) ; retour de la valeur
    memo-fib)) ; retour de la fonction
```

Les mémo-fonctions : utilisation

Comme pour les générateurs, il faut créer la fermeture par une première application de la fonctionnelle.

Exemple

```
> (define memo-fib (make-memo-fib))
> (memo-fib 5)
5
> (memo-fib 8)
21
>
> (time (memo-fib 100))
cpu time: 6 real time: 7 gc time: 0
354224848179261915075
```

Évaluation applicative : (eval o env)

Cette forme d'évaluation est utilisée pour toutes les fonctions construites avec des **lambda**, **define**, **let** et **letrec**. C'est celle qui est mise en oeuvre dans la plupart des langages de programmation, en particulier impératifs (C, Java). Soit **env** l'environnement courant.

- Si l'objet **o** est autoévaluant, renvoyer **o**
- Si **o** est un symbole, alors
  - Rechercher une liaison définissant **o** dans **env**, renvoyer la référence associée.
- Si **o** est une liste
  - Calculer (eval (car **o**) **env**). Soit **f** la fermeture résultat.
  - Calculer (eval **a** **env**), pour tout élément **a** de (cdr **o**). Soit **v** la liste des résultats.
- Calculer (**apply f v**)

Rappels sur l'évaluation et l'application – 2

Application : (apply **f** **v**)

Avec :

- **f** : fermeture de la fonction à appliquer
- **v** : liste des valeurs des arguments

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

Rappels sur l'évaluation et l'application – 3

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 donc les boucles.

Remplacement textuel

Il y a deux niveaux de substitutions : l'appel d'une macro est substitué par la définition de la macro dans laquelle les paramètres formels ont été substitués par les arguments donnés lors de l'appel. Toutes ces substitutions sont textuelles. Ainsi, la structure syntaxique n'est pas prise en compte. En C, les macro-fonctions fonctionnent de cette façon. Pour éviter certains pièges syntaxiques, il faut respecter des règles d'écriture des macros (paramètres entre parenthèses, corps entre parenthèses).

Remarque : Quelques problèmes liés à la non prise en compte de la syntaxe

#define CARRE(x) x*x

3*CARRE(x+1) $\longrightarrow$ $\boxed{3*x+1*x+1}$ ou $\boxed{3*(x+1)*(x+1)}$

#define CARRE(x) ((x)*(x))

CARRE(x++) $\longrightarrow$ $\boxed{((x++) * (x))}$ ou $\boxed{((x++) * (x++))}$

Macroexpansions par transformation de source

En lisp et en scheme, les macroexpansions fonctionnent par transformation de source, en tenant compte de la syntaxe. Elles permettent d'écrire ces formes dites spéciales, dont l'évaluation n'est pas applicative. Les arguments sont évalués sur demande (en lisp) ou par nécessité (en scheme).

Définition en scheme

```
(define-syntax-rule ⟨pattern⟩ ⟨template⟩)
```

- ⟨*pattern*⟩ : (⟨*nom*⟩-⟨*macro*⟩ ⟨p_1⟩ ...)
- ⟨p_i⟩ : variables de la macro
- ⟨*template*⟩ : expressions
- Remplacement des variables dans le template
- Le résultat est une forme
- Évaluation de la forme dans l'environnement d'appel

Macroexpansions en scheme

Exemple

```
(define-syntax-rule (ifnot test then else)
  (if (not test)
      then
      else))
```

- > (define x 1)
- > (expand-once #'(ifnot (= 1 2) 0 x)) ⟶ #<syntax:8:17 (if (not (= 1 2)) 0 x)>
- > (syntax->datum (expand-once #'(ifnot (= 1 2) 0 x))) ⟶ '(if (not (= 1 2)) 0 x)
- > (ifnot (= 1 2) 0 x) ⟶ 0

Remarque : On constate que les arguments ne sont pas évalués lors du remplacement. Une modification de paramètres est alors possible dans la macro.

Macroexpansions

Mécanisme de citation

- **Quote** : '
- **Backquote** : `
- **Virgule** : ,
- **Arobase** : @

Exemple

> (define l '(1 2 3))

> `(1 ,l) $\longrightarrow$ (1 (1 2 3))

> `(+ ,@l) $\longrightarrow$ (+ 1 2 3)

Macroexpansions en lisp

Syntaxe

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

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.

Exemple

```
(defmacro ifn (test e1 e2)
   `(if (not ,test) ,e1 ,e2))
> (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.

Problèmes d'évaluations multiples : en scheme, lisp et C

```
(define-syntax-rule (cube x)
   (* x x x))
```

- Problème d'utilisation

```
> (define a 1)
> (define-syntax-rule (++! x)
     (begin (set! x (add1 x))
            x))
> (cube (++! a)); -> (* (++! a) (++! a) (++! a))
24
```

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

```
(define-syntax-rule (cube x)
   (let ((tmp x))
     (* tmp tmp tmp)))

> (cube (++! a)); ->    (let ((tmp (++! a))
                ;          (* tmp tmp tmp)))

125
```

Question : évaluations multiples

```
(define-syntax-rule (for i start end body)
  (letrec ((loop (lambda (i)
                   (if (> i end) (void)
                       (begin body (loop (+ i 1)))))))
    (loop start)))
```

- Soit la session suivante, quel est le résultat de l'évaluation de a?
  0 ou 1 ou 2 ou 3

```
> (define a 0)
> (for k 1 2 (set! a (+ a k)))
> a
```

i $\longrightarrow$ k   start $\longrightarrow$ 1   end $\longrightarrow$ 2
body $\longrightarrow$ (set! a (+ a k))

- Soit la session suivante, combien de fois l'expression (∗ 1 2) est-elle évaluée dans la macro? 1 ou 2 ou 3 ou 4

```
> (define a 0)
> (for i 1 (* 1 2) (set! a (+ a i)))
```

Question : évaluations multiples

```
(define-syntax-rule (for i start end body)
   (let ((end-value end))
     (letrec ((loop (lambda (i)
                  (if (> i end-value) (void)
                      (begin body (loop (+ i 1)))))))
        (loop start))))
```

```
> (define a 0)
> (for i 1 (begin (print a) 2)  (set! a (+ a i)))
0
```

- Quel est le résultat de l'expression suivante Stack Overflow ou i: Undefined

```
> (for i 1 (+ 1 i) (set! a (+ a i)))
```

Problèmes de captures de noms : en lisp et C

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

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

- $*j* \longrightarrow 5$ – tmp $\longrightarrow 3$

> (echanger tmp *j*)

> (macroexpand-1 '(echanger tmp *j*))

```
(LET ((TMP TMP))
  (SETF TMP *J*)
  (SETF *J* TMP))
```

- Que vaut *j* : 3 ou 5 ?

Renommage des variables locales

Pour éviter la capture de nom, il faut employer des noms de temporaires qu'aucun utilisateur ne pourra imaginer. En Common-lisp (et en scheme aussi), la forme **gensym** permet d'engendrer des noms de symboles nouveaux à chaque appel. En C il faut créer une macro qui fabrique un nom par concaténation avec une partie aléatoire.

```
> (gensym)
#:G879
```

- Exemple

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

- Macroexpansion

```
> (macroexpand-1 '(echanger tmp *l*))
(LET ((#:G893 TMP))
  (SETF TMP *l*)
  (SETF *l* #:G893))
```

Utilisation des macros

- Les macros ne sont pas des fonctions, elles ne sont donc pas utilisables avec les fonctionnelles (map, apply, fold).
- Les macros permettent de créer de nouvelles formes syntaxiques, conditionnelles ou modifiant leurs paramètres, car leur application ne s'effectue pas de façon applicative.
- Pour éviter l'évaluation multiple, il faut lier le paramètre destiné à apparaître plusieurs fois dans le corps de la macro, avec une variable temporaire (nommée par la forme **gensym**).
- En common-lisp et en C, pour éviter la capture de variables, il faut créer des variables temporaires nommées par la forme **gensym** en common-lisp ou par une macro C.
- Ces difficultés impliquent que les fonctions sont conseillées chaque fois que leur utilisation est possible.