Programmation fonctionnelle

- Historique
  - (1941) Lambda-calcul: inventé par Alonzo Church pour étudier la calculabilité
  - (1959) Lisp: premier langage symbolique pour l'IA
    - syntaxe simple et uniforme, préfixe parenthésé
    - récursion, listes, "garbage collection"
    - interactif, système de développement intégré, interprété et compilé
    - typage dynamique, polymorphisme
    - portée dynamique
  - (1975) Scheme: Lisp épuré et minimal, portée lexicale
  - (1987) ML: typage statique, syntaxe infixe
  - (1990) Haskell: fonctionnel pur

Transparence référentielle (1)

- Principe à la base de la programmation fonctionnelle
- Principe selon lequel le résultat du programme ne change pas si on remplace une expression par une expression de **valeur égale**
- Ce principe
  - Facilite l'analyse d'un programme (par exemple pour démontrer qu'il fait le calcul désiré)
  - Facilite les transformations de programme (par exemple pour en améliorer les performances ou le compiler)
    - Exemple: une règle qui dit "$X + X = 2 \times X$" permet de remplacer `f(y)+f(y)` par `2*f(y)`
- La programmation impérative viole ce principe à cause des effets de bords (affectation, E/S, ...)
- Programmation fonctionnelle = sans affectation

Transparence référentielle (2)

- Conséquences
  - La valeur d'une expression dépend seulement de la valeur de ses sous-expressions (p.e. la valeur de $E_1 + E_2$ dépend seulement de la valeur de E_1 et de E_2, et la valeur de E_2 n'est pas influencée par E_1, et vice versa)
  - L'ordre d'exécution a beaucoup moins d'importance qu'en programmation impérative (E_1 peut être évaluée avant, après ou pendant l'évaluation de E_2)
  - Le modèle de calcul est proche des mathématiques, ce qui permet d'appliquer les mêmes techniques de preuves et de raisonnement (p.e. preuve par induction)

Transparence référentielle (3)

- Déf: L'**évaluation** d'une expression c'est le processus qui permet de trouver sa valeur
- Les langages fonctionnels définissent ce processus d'évaluation comme l'application itérée de **règles de réduction** sur l'expression à évaluer jusqu'à l'obtention d'une **forme normale** (une expression qui ne peut plus être réduite)
- Chaque règle de réduction spécifie l'égalité de deux expressions
- Exemple de règles de réduction simplifiées pour Scheme:

  1. `(+` X Y`)` $= N$ si $N = X + Y$

  2. `(*` X Y`)` $= N$ si $N = X \times Y$

  Donc, `(+ (* 2 3) (* 4 5))` = `(+ (* 2 3) 20)` = `(+ 6 20)` = `26`

Scheme: Introduction

- Scheme est surtout utilisé interactivement avec une boucle "read-eval-print"

```
% gsi
Gambit Version 3.0

> 123
123
> (+ 1 (* 2 3) 4)
11
> (expt 2 100)
1267650600228229401496703205376
> (/ (expt 8 33) (expt 2 100))
1/2
> (odd? 5)
#t
> (+ 1 "deux")
*** ERROR IN (stdin)@6.1 -- NUMBER expected
(+ 1 "deux")
1> (+ 1 2)
3
1> (exit)
%
```

- Ceci est beaucoup plus rapide que éditer-compiler-exécuter car il est possible de corriger le programme pendant son exécution (sans avoir à le redémarrer)

Scheme: Syntaxe

- Scheme comme tous les langages fonctionnels est un langage basé sur les expressions (la catégorie syntaxique principale est l'**expression**)

- À part les expressions triviales (constante et variable) les expressions ont toujours la forme parenthésée suivante:

$$(\ \langle \textit{opération} \rangle \quad \langle \textit{arguments} \rangle \ )$$

- Si ⟨*opération*⟩ est un mot clé (comme `if` et `lambda`) alors l'évaluation se fait suivant des règles de réduction spéciales (on parle de **"forme spéciale"**)

- Sinon, il sagit d'un appel de fonction: ⟨*opération*⟩ est la fonction à appeler et ⟨*arguments*⟩ sont les paramètres actuels **passés par valeur**

- On utilise la notation "E `=>` V" pour indiquer que l'évaluation de l'expression E donne V

- Exemple: `(+ (* 2 3) (* 4 5)) => 26`

Scheme: Formes spéciales de base (1)

- `(if` ⟨*expr*$_1$⟩ ⟨*expr*$_2$⟩ ⟨*expr*$_3$⟩`)`

  – Sémantique opérationnelle

    1. évaluer ⟨*expr*$_1$⟩

    2. si le résultat n'est pas `#f` retourner la valeur de ⟨*expr*$_2$⟩ sinon retourner la valeur de ⟨*expr*$_3$⟩

  – Sémantique par règles de réduction

    1. `(if #f` Y Z`)` $= Z$

    2. `(if` X Y Z`)` $= Y$ si X est une forme normale différente de `#f`

- Exemple: `(if (< 1 2) 1 2)`

- Évaluation: `(if (< 1 2) 1 2)` = `(if #t 1 2)` = `1`

Scheme: Formes spéciales de base (2)

- `(let (` $\Big\{$ `((`⟨*ident*$_1$⟩ ⟨*expr*$_1$⟩`)` $\Big\}$ `)` ⟨*expr*$_0$⟩`)`

  – Sémantique opérationnelle

    1. évaluer ⟨*expr*$_1$⟩,. . .

    2. créer les variables locales ⟨*ident*$_1$⟩,. . . dont la valeur initiale est donnée par ⟨*expr*$_1$⟩,. . . et dont la portée est ⟨*expr*$_0$⟩

    3. retourner la valeur de ⟨*expr*$_0$⟩

  – Sémantique par règles de réduction

    1. `(let ((`V_1 E_1`)...)` E_0`)` $= E_0$ avec toutes les références à V_1 remplacées par E_1,. . . (il est important de respecter les règles de portée lexicale et renommer au besoin les variables qui causent un conflit de nom)

- Le `let` permet d'attacher des noms à des valeurs à l'intérieur d'un bloc

- Exemple 1:

```
(let ((x (+ 1 2))
      (y (* 3 4)))
  (if (< x y) x y))
```

Scheme: Formes spéciales de base (3)

- Exemple 1, évaluation 1 (**ordre applicatif**)

```
  (let ((x (+ 1 2))
        (y (* 3 4)))
    (if (< x y) x y))

= (let ((x 3)
        (y (* 3 4)))
    (if (< x y) x y))

= (let ((x 3)
        (y 12))
    (if (< x y) x y))

= (if (< 3 12) 3 12)

= (if #t 3 12)

= 3
```

- Exemple 1, évaluation 2 (**ordre normal**)

```
  (let ((x (+ 1 2))
        (y (* 3 4)))
    (if (< x y) x y))

= (if (< (+ 1 2) (* 3 4)) (+ 1 2) (* 3 4)))

= (if (< 3 (* 3 4)) (+ 1 2) (* 3 4)))

= (if (< 3 12) (+ 1 2) (* 3 4)))

= (if #t (+ 1 2) (* 3 4)))

= (+ 1 2)

= 3
```

- La forme normale obtenue est la même quel que soit l'ordre de réduction

Scheme: Formes spéciales de base (4)

- Exemple 2 avec évaluation en ordre applicatif

```
  (let ((x 1))
    (+ (let ((x (+ x 1)) (y (+ x 2))) (* x y))
       x))

= (+ (let ((x (+ 1 1)) (y (+ 1 2))) (* x y))
     1)

= (+ (let ((x 2) (y (+ 1 2))) (* x y))
     1)

= (+ (let ((x 2) (y 3)) (* x y))
     1)

= (+ (* 2 3)
     1)

= (+ 6
     1)

= 7
```

- Dans le corps du `let` il faut seulement remplacer les variables qui sont **libres**, c'est-à-dire qui ne sont pas l'objet d'un `let` dans le corps du `let` (cela est nécessaire pour obtenir la **portée lexicale**)

Scheme: Formes spéciales de base (5)

- Lors de la réduction d'un `let` il ne faut pas qu'une des expressions substituées pour les variables contienne des variables libres qui ne seront plus libres après la réduction

- Pour contourner ce problème l'évaluateur renomme au besoin les variables qui causent un conflit

```
      INCORRECT         |       CORRECT
                        |
  (let ((x 0))          |   (let ((x 0))
    (let ((y (+ x 1)))  |     (let ((y (+ x 1)))
      (let ((x 99))     |       (let ((x 99))
        (+ x y))))      |         (+ x y))))
                        |
= (let ((x 0))          | = (let ((x 0))
    (let ((x 99))       |     (let ((y (+ x 1)))
      (+ x (+ x 1))))   |       (let ((x2 99))    ;;; x -> x2
                        |         (+ x2 y))))
= (let ((x 0))          |
    (+ 99 (+ 99 1)))    | = (let ((x 0))
                        |     (let ((x2 99))
= (+ 99 (+ 99 1))       |       (+ x2 (+ x 1))))
                        |
                        | = (let ((x 0))
                        |     (+ 99 (+ x 1)))
                        |
                        | = (+ 99 (+ 0 1))
```


Scheme: Formes spéciales de base (6)

- `(lambda (` $\{ \langle \mathit{ident}_1 \rangle \}$ `)` $\langle \mathit{expr}_0 \rangle$`)`
  - Sémantique opérationnelle
    - cette "lambda-expression" **crée** une fonction anonyme ("fermeture") et la retourne
    - les paramètres formels de la fonction sont $\langle \mathit{ident}_1 \rangle$,... (notez l'absence de type)
    - lorsque cette fonction est appelée, il y a création des variables locales $\langle \mathit{ident}_1 \rangle$,... dont la valeur initiale est donnée par les paramètres actuels et dont la portée est $\langle \mathit{expr}_0 \rangle$, ensuite la valeur de $\langle \mathit{expr}_0 \rangle$ est retournée comme résultat de l'appel
  - Sémantique par règles de réduction
    1. `((lambda (`V_1`...)` E_0`)` E_1`...)` $=$ `(let ((`V_1 E_1`)...)` E_0`)`

- Exemple:

```
(let ((f (lambda (x) (* x x))))
  (f (f 10)))
```

Scheme: Formes spéciales de base (7)

- Évaluation en ordre applicatif

```
  (let ((f (lambda (x) (* x x))))
    (f (f 10)))

= ((lambda (x) (* x x)) ((lambda (x) (* x x)) 10))

= (let ((x ((lambda (x) (* x x)) 10))) (* x x))

= (let ((x (let ((x 10)) (* x x)))) (* x x))

= (let ((x (* 10 10))) (* x x))

= (let ((x 100)) (* x x))

= (* 100 100)

= 10000
```


Scheme: Environnements (1)

- Déf: l'**environnement** d'évaluation d'une expression c'est l'ensemble des variables qui sont accessibles et leur valeur

- Déf: la **liaison** d'une variable c'est la valeur qui lui est associée dans l'environnement

- Exemple:

```
(let ((x (+ 1 2))) ; liaison de x à 3
  (* x x))
```

- L'environnement contient des variables **locales** (qui sont déclarées par `let` et `lambda`) et **globales**

- Les variables globales sont soit **prédéfinies** (i.e. elles sont liées à une valeur standard) ou sont définies explicitement avec la forme spéciale `define`

Scheme: Environnements (2)

- Les variables prédéfinies (telles que `+`, `*`, `sqrt`) sont liées aux **fonctions primitives** de Scheme
  - valeur de `+` => fn. d'addition (arité variable)
  - valeur de `sqrt` => fn. racine carrée
  - valeur de `not` => fn. négation Booléene

- L'appel de fonction a la syntaxe

$$(\ \langle expr_0 \rangle \ \ \{ \ \langle expr_1 \rangle \ \} \ )$$

- Les paramètres actuels **et** $\langle expr_0 \rangle$ sont évalués avec les règles normales d'évaluation

- Exemple 1 avec évaluation:

```
  (+ 1 2)
= (#<procedure +> 1 2)
= 3
```


Scheme: Environnements (3)

- Exemple 2 avec évaluation:

```
  (let ((x 1) (y 2)) ((if (< x y) * +) x y))
= ((if (< 1 2) * +) 1 2)
= ((if #t * +) 1 2)
= (* 1 2)
= (#<procedure *> 1 2)
= 2
```

- Exemple 3 avec évaluation:

```
 (+ (let ((+ *)) (+ 1 2))
    (+ 3 4))

= (+ (* 1 2)
     (+ 3 4))

= (+ (#<procedure *> 1 2)
     (+ 3 4))

= (+ 2
     (+ 3 4))

= (+ 2
     (#<procedure +> 3 4))

= (+ 2
     7)

= (#<procedure +> 2 7)

= 9
```

Scheme: Environnements (4)

- `(define` ⟨*ident*⟩ ⟨*expr*⟩`)`
  - pas une expression (i.e. pas de valeur)
  - crée la variable globale ⟨*ident*⟩ et l'initialise à la valeur de ⟨*expr*⟩
  - Exemple:

```
(define n 10)
(define m (+ 1 2))
(define carre (lambda (x) (* x x)))
```

- Un programme Scheme c'est un **ensemble de définitions globales** et une **expression** à évaluer dans l'environnement global (sa valeur c'est le résultat du programme)
- Exemple:

```
(define x 3)

(define fact
  (lambda (n)
    (if (< n 2)
        1
        (* n (fact (- n 1))))))

(fact x) => 6
```


Scheme: Environnements (5)

- Évaluation en ordre applicatif:

```
  (fact x)

= (fact 3)

= ((lambda (n)
     (if (< n 2)
         1
         (* n (fact (- n 1)))))
   3)

= (let ((n 3))
    (if (< n 2)
        1
        (* n (fact (- n 1)))))

= (if (< 3 2)
      1
      (* 3 (fact (- 3 1))))

= (* 3 (fact (- 3 1)))     ;;; après quelques réductions

= (* 3 (fact 2))           ;;; après quelques réductions

= (* 3 ((lambda (n)        ;;; après quelques réductions
          (if (< n 2)
              1
              (* n (fact (- n 1)))))
        2))

= (* 3 (* 2 ((lambda (n)   ;;; après quelques réductions
               (if (< n 2)
                   1
                   (* n (fact (- n 1)))))
             1)))

= (* 3 (* 2 1))            ;;; après quelques réductions

= 6                        ;;; après quelques réductions
```


Scheme: Environnements (6)

- Le concept de type n'est **pas attaché aux variables**
- Le **type est associé aux données** seulement (p.e. entier, réel, caractère, Booléen, etc)
- Exemple:

```
(define sinon
  (lambda (x y z)
    (if (not x) y z)))

(sinon #f #t #f) => #t
(sinon #t 11 22) => 22

(let ((a (sinon (< b 0) 1 #f)))
  ...) ; a sera liée à un entier ou un Booléen
```


Scheme: Environnements (7)

- L'environnement d'évaluation correspond à la **chaîne statique** (donc représenté par une liste de blocs contenant les variables de chaque niveau lexical)
- Chaque fermeture mémorise son environnement d'évaluation pour pouvoir y référer à son appel
- Exemples:

```
(define f
  (lambda (n)
    (let ((g (lambda (x) (+ x n))))
      (* (g 10) 2))))

(f 1) => 22

(define f
  (lambda (n)
    (lambda (x) (+ x n))))

(define a1 (f 1))   ;;; note: (f 1) = (lambda (x) (+ x 1))
(define a2 (f 2))   ;;; note: (f 2) = (lambda (x) (+ x 2))

(a1 10) => 11
(a2 10) => 12
```

- Les environnements **persistent tant qu'il y a une fermeture qui en a besoin** (en général il faut donc un GC pour récupérer l'espace associé)

Trucs de débogage avec Gambit

L'interprète Gambit offre un débogueur, intégré à `emacs`, qui permet de **générer une trace des appels**, **l'exécution pas-à-pas**, **l'inspection de la chaîne d'appel**, etc

```
% gsi
Gambit Version 3.0

> (define fact
    (lambda (n)
      (if (< n 2)
          1
          (* n (fact (- n 1))))))
> (trace fact)
> (fact 5)
| > (fact 5)
| | > (fact 4)
| | | > (fact 3)
| | | | > (fact 2)
| | | | | > (fact 1)
| | | | | 1
| | | | 2
| | | 6
| | 24
| 120
120
> (untrace fact)
> (begin (step) (fact 5))
*** STOPPED IN (stdin)@9.16
1> ,s
| > fact
| #<procedure fact>
*** STOPPED IN (stdin)@9.21
1> ,s
| > 5
| 5
*** STOPPED IN (stdin)@9.15
1> ,s
| > (fact 5)
*** STOPPED IN fact, (stdin)@3.12
1> ,s
| | > <
| | #<procedure <>
*** STOPPED IN fact, (stdin)@3.14
```


Types: Nombres

- Scheme supporte la "tour des nombres" et le concept d'exactitude (nombres "exacts" / "inexacts")

  1. entier: `123 #b1101 -832742923398798898 123.0`
  2. rationnel: `123 1/2 .5 1.3e-10`
  3. complexe: `123 1/2 .5 +i 6+2i 1.9-5.3i`

- Fonctions primitives

```
(+ 9 6) => 15  (- 9 6) => 3  (* 9 6) => 54  (/ 9 6) => 3/2

(quotient 9 6) => 1  (modulo 9 6) => 3

(= 9 9.0) => #t      (< 4 1) => #f       (max 1 9 5) => 9

(exp 1)   => 2.718281828459045    (sqrt 1/9) => 1/3
(sin 1.5) => .9974949866040544    (sqrt -1) => +i
(asin 2)  => 1.5707963267948966-1.3169578969248166i
```

- Déf: Un nombre est **exact** s'il n'y a pas de perte de précision dans son calcul
  - exact: `(sqrt 9) => 3`
  - inexact: `(sqrt 2) => 1.4142135623730951`

Types: Caractères, chaînes et symboles

- Caractère: `#\x #\space #\newline`
- Chaîne: `"allo"`
- Fonctions primitives

```
(char=? #\x #\X) => #f    (char<? #\a #\x) => #t

(string=? "a" "x") => #f  (string<? "a" "x") => #t

(string-length "abc") => 3
(string-ref "abc" 1) => #\b
(string-append "abc" "123") => "abc123"
(number->string 123) => "123"
```

- Symbole: `allo temp-max + personne.age`
- Fonctions primitives

```
(string->symbol "allo") => allo
(symbol->string (string->symbol "allo")) => "allo"
(symbol->string 'allo) => "allo"    NOTE: ``quote'' requis
(eq? 'allo (string->symbol "allo")) => #t
(eq? 'allo "allo") => #f
```

- Forme spéciale "quote":
  `'`⟨*donnée*⟩ = `(quote` ⟨*donnée*⟩`)` => ⟨*donnée*⟩

Types: Liste (1)

- Liste = groupe ordonné de données (hétérogène)
- Syntaxe: `(` { ⟨*donnée*⟩ } `)`
- Exemples: `() (2 8 3) (allo 2) (+ 1 (* 2 3))`
- Dans un programme il faut utiliser la forme spéciale "quote" pour évaluer une liste constante (comme pour un symbole)

```
(define x (+ 1 2))    ; liaison de x à 3
(define y '(+ 1 2))   ; liaison de y à (+ 1 2)

x => 3
y => (+ 1 2)
```

- La syntaxe des **expressions** et des **données** est très proche (ce qui porte souvent à confusion)
- Le langage (au sens syntaxique) des expressions est un **sous-ensemble** du langage des données (très utile pour manipuler des programmes Scheme en Scheme, par exemple dans les macros)

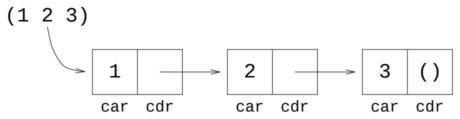
Types: Liste (2)

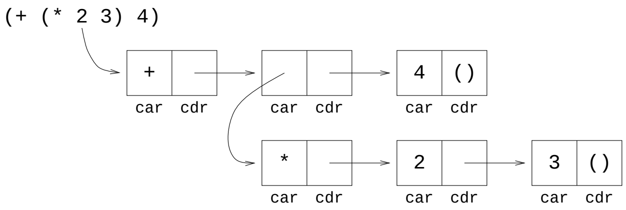
- Déf: une **liste (propre)** est soit

  1. une liste vide

  2. une donnée suivie d'une liste (propre)

- Pour mieux expliquer les fonctions primitives, il est utile de comprendre la représentation des listes sous forme de **paires chaînées**





Types: Liste (3)

- Fonctions primitives `car` et `cdr`

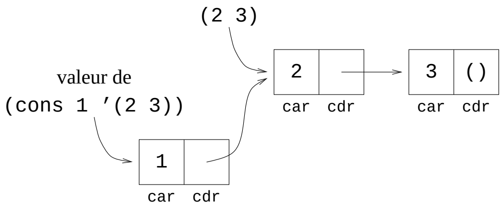
```
(define lst '(+ (* 2 3) 4)) ; note: ''quote''

(car lst)              => +

(cdr lst)              => ((* 2 3) 4)

(car (cdr lst))        => (* 2 3)

(cdr (cdr lst))        => (4)

(car (cdr (cdr lst))) => 4

(cadr lst)  => (* 2 3)
(cddr lst)  => (4)
(caddr lst) => 4
```

- Les formes composées `c...r` existent jusqu'à un niveau de 4

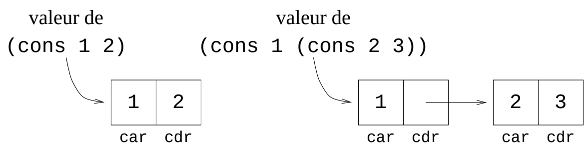
Types: Liste (4)

- Fonctions primitives `list` et `cons`

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

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




```
(cons 1 '())    => (1)
(cons 1 (cons 2 '())) => (1 2)

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



- Récupération automatique: `(car (list 1 2 3))`

Types: Liste (5)

- Déf: une **liste impropre** c'est une **paire** dont le champ `cdr` ne contient pas une liste propre

- Syntaxe:
  ($\{$ ⟨*donnée*⟩ $\}$ ⟨*donnée*⟩ . ⟨*donnée!=()*⟩)

- Autres fonctions primitives sur les listes

```
(length '(a b c)) => 3

(append '(1 2 3) '(4 5)) => (1 2 3 4 5)
(append '(1 2) '(3) '(4 5)) => (1 2 3 4 5)

(reverse '(a b c)) => (c b a)

(list-ref '(a b c d) 2) => c

(null? '()) => #t       (null? '(1 2 3)) => #f
(pair? '()) => #f       (pair? '(1 2 3)) => #t

(equal? 'allo 'allo) => #t
(equal? 'allo "allo") => #f
(equal? '(allo marc) '(allo marc)) => #t
```

Traitement de liste (1)

- Le traitement de liste est souvent réalisé à l'aide de **fonctions récursives** (il n'existe pas d'équivalent fonctionnel direct des boucles)

- Calcul de la longueur d'une liste propre

  1. **vide** => 0

  2. **non-vide** => 1 + longueur de la liste après y avoir retiré son premier élément

```
(define length
  (lambda (lst)
    (if (null? lst)
        0
        (+ 1 (length (cdr lst))))))
```

- Trace:

```
| > (length '(a b c))
| | > (length '(b c))
| | | > (length '(c))
| | | | > (length '())
| | | | 0
| | | 1
| | 2
| 3
```


Traitement de liste (2)

- Concaténation de 2 listes

  1. `(append '() '(4 5))` => `(4 5)`

  2. `(append '(1 2 3) '(4 5))` => `(1 2 3 4 5)`
     qui est `(cons 1 (append '(2 3) '(4 5)))`

```
(define append
  (lambda (lst1 lst2)
    (if (null? lst1)
        lst2
        (cons (car lst1)
              (append (cdr lst1) lst2)))))
```

- Trace:

```
| > (append '(1 2 3) '(4 5))
| | > (append '(2 3) '(4 5))
| | | > (append '(3) '(4 5))
| | | | > (append '() '(4 5))
| | | | (4 5)
| | | (3 4 5)
| | (2 3 4 5)
| (1 2 3 4 5)
```


Traitement de liste (3)

- L'approche fonctionnelle a l'avantage de permettre de prouver facilement certaines propriétés des fonctions

- Soit $P(x,y)$ = nombre de paires créées par l'appel `(append` x y`)`

- Soit $L(x)$ = résultat de l'appel `(length` x`)`

- Prouvons que $P(x,y) = L(x)$

- Preuve par induction sur x

  1. Base: x = `'()`
     $P(x,y) = P(\texttt{'()},y) = 0 = L(\texttt{'()}) = L(x)$

  2. Hypothèse: $P(R,y) = L(R)$
     montrons que $P(x,y) = L(x)$ où x = `(cons` T R`)`

     $P(x,y) = P(\texttt{(cons } T\ R\texttt{)},y) = 1+P(R,y) = 1+L(R) = L(\texttt{(cons } T\ R\texttt{)}) = L(x)$

Traitement de liste (4)

- Les listes peuvent représenter des ensembles

  Par exemple: `(rouge jaune bleu)` = ensemble des couleurs primaires

- Tester l'appartenance à un ensemble

  1. `(member 5 '())` => `#f`

  2. `(member 5 '(5 7))` => `(5 7)`

  3. `(member 5 '(3 5 7))` = `(member 5 '(5 7))`

```
(define member
  (lambda (x lst)
    (if (null? lst)
        #f
        (if (equal? x (car lst))
            lst
            (member x (cdr lst))))))
```

- Note: `member` est une fonction prédéfinie

Traitement de liste (5)

- Pour une évaluation conditionnelle à plusieurs branches on se sert normalement de la forme spéciale "`cond`":

  `(cond` $\{$ `(`⟨*expr*$_1$⟩ ⟨*expr*$_2$⟩`)` $\}$
  `(else` ⟨*expr*$_3$⟩`))`

- Cela évite d'imbriquer l'expression trop profondément:

```
(define member
  (lambda (x lst)
    (cond ((null? lst)
           #f)
          ((equal? x (car lst))
           lst)
          (else
           (member x (cdr lst))))))
```


Traitement de liste (6)

- Les listes peuvent représenter des dictionnaires (structure associant une donnée à une clé)
- Déf: une **liste d'association** c'est une liste de paires de la forme (⟨*clé*⟩ . ⟨*donnée*⟩)

  Par exemple: `((pomme . 100) (orange . 0))` = inventaire d'un épicier

- Recherche dans une liste d'association
  - Liste + clé => donnée
  - Comment indiquer clé pas trouvée?
  - Retourner plutôt `#f` ou paire contenant la clé et donnée

```
(define assoc
  (lambda (cle lst)
    (if (null? lst)
        #f
        (let ((paire (car lst)))
          (if (equal? cle (car paire))
              paire
              (assoc cle (cdr lst)))))))
```


Traitement de liste (7)

- Renverser une liste

  1. `(reverse '())` => `()`
  2. `(reverse '(a b c))` =
     `(append (reverse '(b c)) (list 'a))`

```
(define reverse
  (lambda (lst)
    (if (null? lst)
        '()
        (append (reverse (cdr lst))
                (list (car lst))))))
```

- Soit $R(n)$ = nombre de paires créées par l'appel `(reverse` x`)` où $L(x) = n$
- $R(0) = 0$
  $R(n) = n + R(n-1)$, si $n > 0$
- $R(n) = n + (n-1) + ... + 0 = n(n+1)/2$
- C'est beaucoup d'espace pour un résultat contenant n paires!

Forme itérative (1)

- Souvent, une meilleure solution peut être obtenue en s'attaquant à un problème plus général
- `(append-rev` x y`)` = `(append (reverse` x`)` y`)`

  1. `(append-rev '() '(b a))` => `(b a)`
  2. `(append-rev '(c d e) '(b a))` =
     `(append-rev '(d e) '(c b a))`

```
(define append-rev
  (lambda (lst1 lst2)
    (if (null? lst1)
        lst2
        (append-rev (cdr lst1)
                    (cons (car lst1) lst2)))))

(define reverse
  (lambda (lst)
    (append-rev lst '())))
```

- Crée exactement n paires lorsque $L($`lst`$) = n$

Forme itérative (2)

- Exemple avec factorielle

- (mult-fact n m) = (* (fact n) m)

  1. (mult-fact 0 100) => 100

  2. (mult-fact 3 100) = (mult-fact 2 300)

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

(define fact
  (lambda (n)
    (mult-fact n 1)))
```

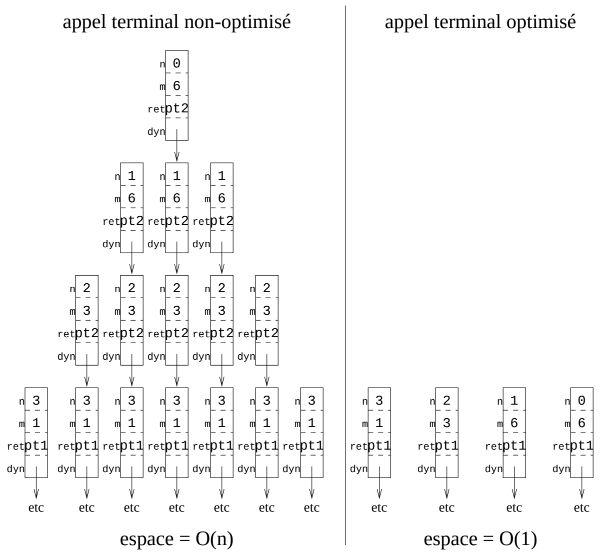
- En Pascal:

```
program forme_iterative;

  function mult_fact( n, m : integer ) : integer;
   begin
    if n = 0 then
     mult_fact := m
    else
     mult_fact := mult_fact(n-1,n*m);
   end;

begin writeln(mult_fact(3,1)); end.
```


Forme itérative (3)

```
/* appel terminal non-optimisé */     | /* appel terminal optimisé */
                                      |
typedef struct bloc                   | typedef struct bloc
  { int n, m;                         |   { int n, m;
    void *ret;                        |     void *ret;
    struct bloc *dyn;                 |     struct bloc *dyn;
  } bloc;                             |   } bloc;
                                      |
bloc *bac, *p;                        | bloc *bac, *p;
int mult_fact_res;                    | int mult_fact_res;
                                      |
void forme_iterative ()               | void forme_iterative ()
{bac = NULL;                          | {bac = NULL;
                                      |
 p = (bloc*)malloc(sizeof(bloc));     |  p = (bloc*)malloc(sizeof(bloc));
 p->n = 3;                            |  p->n = 3;
 p->m = 1;                            |  p->m = 1;
 p->ret = &&point1;                   |  p->ret = &&point1;
 p->dyn = bac;                        |  p->dyn = bac;
 bac = p;                             |  bac = p;
 goto mult_fact;                      |  goto mult_fact;
point1:                               | point1:
 p = bac->dyn;                        |  p = bac->dyn;
 free(bac);                           |  free(bac);
 bac = p;                             |  bac = p;
 printf("%d",mult_fact_res);          |  printf("%d",mult_fact_res);
 return;                              |  return;
                                      |
mult_fact:                            | mult_fact:
 if (bac->n == 0)                     |  if (bac->n == 0)
  mult_fact_res = bac->m;             |   mult_fact_res = bac->m;
 else                                 |  else
  { p = (bloc*)malloc(sizeof(bloc));  |   { p = (bloc*)malloc(sizeof(bloc));
    p->n = bac->n - 1;                |     p->n = bac->n - 1;
    p->m = bac->n * bac->m;           |     p->m = bac->n * bac->m;
    p->ret = &&point2;                |     p->ret = bac->ret; /********/
    p->dyn = bac;                     |     p->dyn = bac->dyn; /********/
    bac = p;                          |     free(bac);         /********/
    goto mult_fact;                   |     bac = p;
  point2:                             |     goto mult_fact;
    p = bac->dyn;                     |   }
    free(bac);                        |  goto *bac->ret;
    bac = p;                          | }
  }                                   |
 goto *bac->ret;                      |
}                                     |
```


Forme itérative (4)



Forme itérative (5)

- Déf: une expression E est en **position terminale** par rapport à une fonction F si le résultat de l'évaluation de E est retourné **directement** comme résultat de F

```
(define fact
  (lambda (n)
    (if (< n 2)     ; (if ...) est en position terminale
        1           ; 1 est en position terminale
        (* n        ; (* ...) est en position terminale
           (fact (- n 1))))))
```

- Déf: un **appel terminal** c'est un appel de fonction qui est en position terminale

- Déf: une fonction récursive est en **forme itérative** si les appels récursifs sont tous des appels terminaux

Forme itérative (6)

- `fact` n'est pas en forme itérative, mais `mult-fact` l'est:

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

- Une fonction en forme itérative consomme un **espace constant** (au niveau des blocs d'activation) tout au long de la récursion
- Ce comportement (espace constant) est requis par Scheme
- La forme itérative en Scheme peut donc jouer le **rôle des boucles** dans les langages impératifs

Forme itérative (7)

- Certains compilateurs de langage impératif (par exemple `gcc` et `cc` de SUN) **détectent et optimisent les récursions terminales**

```
int mult_fact(int n, int m)          int mult_fact(int n, int m)
{ if (n != 0)                        { while (n != 0)
    return mult_fact(n-1, n*m);  ==>     { m = n*m; n = n-1; }
  else                                 return m;
    return m;                        }
}
```

- Les bons compilateurs Scheme compilent les fonctions en forme itérative dans le **même code qu'une boucle**
- Ils **réutilisent** le bloc d'activation courant pour la nouvelle activation

```
/* allocation d'un nouveau bloc */  | /* réutilisation du bloc */
                                    |
mult_fact:                          | mult_fact:
 if (bac->n == 0)                   |  if (bac->n == 0)
  mult_fact_res = bac->m;           |   mult_fact_res = bac->m;
 else                               |  else
  {p = (bloc*)malloc(sizeof(bloc)); |   {
   p->n = bac->n - 1;               |    bac->m = bac->n * bac->m;
   p->m = bac->n * bac->m;          |    bac->n = bac->n - 1;
   p->ret = bac->ret;               |
   p->dyn = bac->dyn;               |
   free(bac);                       |
   bac = p;                         |
   goto mult_fact;                  |    goto mult_fact;
  }                                 |   }
 goto *bac->ret;                    |  goto *bac->ret;
```


Forme itérative (8)

- Les traces produites par `trace` **tiennent compte des appels terminaux** (l'indentation ne change pas, ce qui indique que le résultat de la fonction sera le résultat de l'appel terminal)
- Exemples

```
avec appel terminal              si appel était non-terminal

| > (mult-fact 5 1)              | > (mult-fact 5 1)
| > (mult-fact 4 5)              | | > (mult-fact 4 5)
| > (mult-fact 3 20)             | | | > (mult-fact 3 20)
| > (mult-fact 2 60)             | | | | > (mult-fact 2 60)
| > (mult-fact 1 120)            | | | | | > (mult-fact 1 120)
| > (mult-fact 0 120)            | | | | | | > (mult-fact 0 120)
| 120                            | | | | | | 120
                                 | | | | | 120
                                 | | | | 120
                                 | | | 120
                                 | | 120
                                 | 120

| > (append-rev '(1 2 3 4) '())  | > (append-rev '(1 2 3 4) '())
| > (append-rev '(2 3 4) '(1))   | | > (append-rev '(2 3 4) '(1))
| > (append-rev '(3 4) '(2 1))   | | | > (append-rev '(3 4) '(2 1))
| > (append-rev '(4) '(3 2 1))   | | | | > (append-rev '(4) '(3 2 1))
| > (append-rev '() '(4 3 2 1))  | | | | | > (append-rev '() '(4 3 2 1))
| (4 3 2 1)                      | | | | | (4 3 2 1)
                                 | | | | (4 3 2 1)
                                 | | | (4 3 2 1)
                                 | | (4 3 2 1)
                                 | (4 3 2 1)
```


Tests de type

- Il existe des fonctions primitives pour tester **dynamiquement** le type d'une donnée (ces fonctions sont applicables sur n'importe quel type de donnée et retournent `#f` ou `#t`)

`(null?` x`)` => x est la liste vide?

`(pair?` x`)` => x est une liste non-vide?

`(char?` x`)` => x est un caractère?

`(number?` x`)` => x est un nombre?

`(string?` x`)` => x est une chaîne de caractères?

`(symbol?` x`)` => x est un symbole?

`(boolean?` x`)` => x est un Booléen (`#f` ou `#t`)?

`(procedure?` x`)` => x est une procédure?

- Exemple

```
(define f
  (lambda (nom)
    (string-append "allo "
                   (if (symbol? nom)
                       (symbol->string nom)
                       nom))))

(f 'marc) => "allo marc"
(f "marc") => "allo marc"
```

Fonctions d'ordre supérieur (1)

- En Scheme, les fonctions sont des données de **première classe**
- Déf: une donnée est de **première classe** si les opérations suivantes sont permises
  1. Stocker dans des structures de donnée (ou variables) et transférer d'une structure (ou variable) à une autre
  2. Passer en paramètre à des fonctions
  3. Retourner comme résultat de fonction
- Il est avantageux que toutes les données soient de première classe afin que le langage soit uniforme (aucune exception d'utilisation)
- En C (contrairement à Pascal), les tableaux ne sont pas de première classe

```
int t1[10]; int t2[10];  t1 = t2; /* erreur */
```

- En Pascal (contrairement à C), les fonctions ne sont pas de première classe (retour de fonction interdit)

Fonctions d'ordre supérieur (2)

- Exemple

```
> (lambda (x) (* x x))
#<procedure #x100380E1>
> (define fns (list (lambda (x) (* x x))
                    (lambda (x) (+ x 1))))
> fns
(#<procedure #x1003830C> #<procedure #x10038311>)
> (car fns)
#<procedure #x1003830C>
> ((car fns) 10)
100
> ((cadr fns) 10)
11
```

- Les fonctions comme données de première classe permettent une programmation plus modulaire:
  - Algorithmes génériques qu'on peut spécialiser en passant une ou des fonctions (p.e. tri)
  - Programme "data driven": table de fonctions consultée pour connaître quel traitement faire en fonction du contexte
  - "Callback": fonction qu'on attache à un événement (p.e. cliquer sur un bouton)

Fonctions d'O.S. sur les listes (1)

- Souvent on a besoin de faire un traitement sur tous les éléments d'une liste
- Exemple: élever au carré chaque élément d'une liste pour produire une nouvelle liste:

```
> (define tous-au-carre
    (lambda (lst)
      (if (null? lst)
          '()
          (cons (* (car lst) (car lst))
                (tous-au-carre (cdr lst))))))
> (tous-au-carre '(1 2 3))
(1 4 9)
```

- Et si on cherche plutôt à mettre au cube:

```
> (define tous-au-cube
    (lambda (lst)
      (if (null? lst)
          '()
          (cons (* (car lst) (car lst) (car lst))
                (tous-au-cube (cdr lst))))))
> (tous-au-cube '(1 2 3))
(1 8 27)
```

- Les fonctions sont identiques sauf pour le calcul à faire sur chaque élément (cette duplication de code est source d'erreur)

Fonctions d'O.S. sur les listes (2)

- Une meilleure solution est de concevoir une fonction "`map`" qui permet d'appliquer un calcul quelconque sur les éléments de la liste
- Le "calcul quelconque" sera spécifié par une fonction qui est passée en paramètre à `map`:

```
(map (lambda (x) (* x x)) '(1 2 3))    =>  (1 4 9)

(map (lambda (x) (* x x x)) '(1 2 3))  =>  (1 8 27)
```

- Implantation de `map` (note: prédéfinie en Scheme)

```
(define map
  (lambda (f lst)
    (if (null? lst)
        '()
        (cons (f (car lst))
              (map f (cdr lst))))))
```

- Ainsi `map` est une fonction **générique** qu'on peut spécialiser au besoin

```
(map reverse '((a b) () (c d e)))  =>  ((b a) () (e d c))
```

- Ont dit aussi que `map` est une fonction **polymorphique** car ses paramètres peuvent être de plusieurs formes (liste de nombres/listes/...)

Fonctions d'O.S. sur les listes (3)

- Autre fonction d'O.S. utile: filtrer les éléments d'une liste ayant une certaine propriété (fonction pour tester la propriété)

```
(garder odd? '(5 8 2 3 1 9))  =>  (5 3 1 9)

(garder (lambda (x) (< x 5)) '(5 8 2 3 1 9))  =>  (2 3 1)
```

- Implantation

```
(define garder
  (lambda (f lst)
    (cond ((null? lst)
           '())
          ((f (car lst))
           (cons (car lst)
                 (garder f (cdr lst))))
          (else
           (garder f (cdr lst))))))
```

- Exemple: tester si tous les éléments d'une liste possèdent une propriété

```
(tous odd? '(9 3 5)) => #t   (tous odd? '(9 3 2 5)) => #f

(define tous
  (lambda (f lst)
    (null? (garder (lambda (x) (not (f x))) lst))))
```


Fonctions d'O.S. sur les listes (4)

- Autre fonction d'O.S. utile: trier une liste dans un certain ordre (fonction pour comparer 2 éléments)

```
(trier < '(5 8 2 3 1 9))  =>  (1 2 3 5 8 9)

(trier string>? '("b" "a" "c"))  =>  ("c" "b" "a")
```

- Implantation avec algorithme "Quicksort":
  1. liste vide: déjà triée!
  2. sinon: *pivot* = 1er élément; *a* = liste des éléments restants < *pivot*; *b* = liste des éléments restants $\geq$ *pivot*; résultat = *a* triée + *pivot* + *b* triée

```
(define trier
  (lambda (f lst)
    (if (null? lst)
        '()
        (let ((pivot (car lst)))
          (let ((a (garder (lambda (x) (f x pivot))
                           (cdr lst)))
                (b (garder (lambda (x) (not (f x pivot)))
                           (cdr lst))))
            (append (trier f a)
                    (cons pivot (trier f b))))))))
```


Fonctions d'O.S. sur les listes (5)

- Autre fonction d'O.S. utile: réduction d'une liste en combinant les éléments (ex. somme ou produit des éléments)

```
(define somme
  (lambda (lst)
    (if (null? lst)
        0
        (+ (car lst) (somme (cdr lst))))))

(somme '(1 2 3 4))  =>  10

(define produit
  (lambda (lst)
    (if (null? lst)
        1
        (* (car lst) (produit (cdr lst))))))

(produit '(1 2 3 4))  =>  24
```

- Ces fonctions diffèrent sur
  1. le résultat pour le cas de base (liste vide): 0 ou 1
  2. la fonction de combinaison: `+` ou `*`

```
(foldr + 0 '(1 2 3 4))  =>  10

(foldr * 1 '(1 2 3 4))  =>  24
```


Fonctions d'O.S. sur les listes (6)

- Implantation de `foldr`

```
(define foldr
  (lambda (f base lst)
    (if (null? lst)
        base
        (f (car lst) (foldr f base (cdr lst))))))
```

- Il est à remarquer que `foldr` combine les éléments à partir de la **droite**, c'est-à-dire:

```
(foldr + 0 '(1 2 3 4)) = (+ 1 (+ 2 (+ 3 (+ 4 0))))

(foldr * 1 '(1 2 3 4)) = (* 1 (* 2 (* 3 (* 4 1))))
```

- Mais on pourrait combiner à partir de la **gauche**:

```
(foldl + 0 '(1 2 3 4)) = (+ (+ (+ (+ 0 1) 2) 3) 4)

(foldl * 1 '(1 2 3 4)) = (* (* (* (* 1 1) 2) 3) 4)
```

- Implantation de `foldl` (note: forme itérative)

```
(define foldl
  (lambda (f base lst)
    (if (null? lst)
        base
        (foldl f (f base (car lst)) (cdr lst)))))
```

