PROGRAMMATION

FONCTIONNELLE

2011-2012

P. Gribomont

1. Introduction et motivation

Premier objectif : Renforcer et appliquer les connaissances acquises

Deuxième objectif : Structuration d'une application en procédures

Troisième objectif : Autonomie en programmation, de la définition du problème à l'utilisation du programme

En bref . . . Pour réussir, il faudra pouvoir programmer et spécifier !

Le moyen

Programmation fonctionnelle en Scheme (Exercices !)

Référence :

P. Gribomont,
Eléments de programmation en Scheme,
Dunod, Paris, 2000.

Version en ligne :

`http://www.montefiore.ulg.ac.be/~gribomon/cours/info0540.html`

Complément très utile :
L. Moreau, C. Queinnec, D. Ribbens et M. Serrano,
Recueil de petits problèmes en Scheme,
Springer, collection Scopos, 1999.

Voir aussi : `http://deptinfo.unice.fr/~roy/biblio.html`

Pourquoi la programmation fonctionnelle ?

- Bon rapport simplicité / généralité
- Plus "naturel" que l'approche impérative
- Favorise la construction et la maintenance

Pourquoi un nouveau langage ?

- L'important est la programmation, pas le langage . . .
  donc il faut pouvoir changer de langage
- C comporte des lacunes gênantes
  (procédures, structures de données, . . .)

Pourquoi Scheme **(dialecte de** Lisp**) ?**

– Scheme est très simple !
– Scheme est puissant
– Scheme comble des lacunes de Pascal
– Scheme est un langage et un métalangage

Remarque. Tout est relatif ; Scheme est simple, si on considère la puissance du langage ! Bien comprendre le système est difficile, même si on se limite à un sous-ensemble du langage. Obtenir une implémentation efficace en mémoire et en temps est très difficile.

C **et** Scheme **(**Pascal **et** Lisp**)**

Pascal is for building pyramids — imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms — imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place.

Alan J. PERLIS

Premier lauréat du prix TURING

(Avant-propos de *Structure and Interpretation of Computer Programs*, par Abelson et Sussman, MIT Press & McGraw-Hill, 1985)

2. Les bases de Scheme

Interpréteur : boucle **Read-Eval-Print**

L'interpréteur **lit** une *expression* e, l'**évalue** et **imprime** sa *valeur* $[[e]]$ (ou un message spécifique).
Il peut aussi **lire** une "définition" et l'enregistrer.

Une expression simple est une constante (numérique ou non) ou une variable.
Une expression composée, ou *combinaison*, comporte un opérateur et des arguments ; les composants sont eux-mêmes des expressions, simples ou composées.
Une combinaison commence par "(" et finit par ")".
Le langage des expressions est inclus dans celui des valeurs (en arithmétique, c'est le contraire).

```
==> (+ 2 5)
:-) 7
```

(On utilise systématiquement la forme préfixée.)
Cette notation implique que `(+ 2 5)` est une expression, et que sa valeur, calculée par l'interpréteur Scheme, est `7`. On écrit $[[(+\ 2\ 5)]] = 7$.


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

`3`, `-2` et `5` sont des expressions simples (nombres) à partir desquelles on forme les combinaisons `(* 3 -2)` et `(- 5 3)`, puis la combinaison `(+ (* 3 -2) (- 5 3))`.

```
==> (/ 8 5)
:-) 8/5

==> (/ 8.0 5)
:-) 1.6

==> (/ 8 3.0)
:-) 2.6666666666666665

==> (/ 2 0)
:-) error - datum out of range 0 within proc /
```

(La valeur de `(/ 2 0)` n'existe pas.)

```
==> 3
:-) 3

==> +
:-) # procedure +
```

[La valeur de + existe et est une procédure (ou fonction).]
[Les valeurs fonctionnelles ne sont pas affichables.]

```
==> pi
:-) 3.1415926535

==> fact
:-) # procedure fact

==> fict
:-) error - unbound variable fict

==> (2 3)
:-) error - bad procedure 2
```


Remarque. La notation Scheme est plus homogène que les notations mathématiques usuelles, où les foncteurs s'emploient de manière préfixée, infixée ou postfixée, ou encore font l'objet de conventions d'écriture spéciales.

Remarque. Dans la session ci-dessus, on suppose que `pi` et `fact` ont été définis par l'utilisateur (on verra comment plus loin), tandis que `fict` n'a pas été défini. Les nombres et opérations classiques sont naturellement prédéfinis.

Remarque. Les messages peuvent varier d'un évaluateur à l'autre, et être plus ou moins informatifs. Un "message d'erreur" peut ne pas comporter le mot `error`.

Remarque. La structure arborescente des expressions se prête à une définition dans le formalisme BNF. En se restreignant aux expressions arithmétiques, on a

⟨A-expression⟩ : := ⟨nombre⟩ | ⟨variable⟩ | ⟨combinaison⟩

⟨combinaison⟩ : := (⟨A-op⟩ [⟨A-expression⟩]*)

⟨A-op⟩ : := + | - | * | /

On voit qu'une combinaison est un assemblage composé (dans l'ordre), d'une parenthèse ouvrante, d'un opérateur arithmétique, d'un certain nombre (0, 1 ou plus) d'expressions arithmétiques et d'une parenthèse fermante.

`define` = liaison à une valeur

"Définition" du nombre π et de son inverse :

```
==> (define pi 3.1415926535)
:-) ...
==> (define invpi (/ 1 pi))
:-) ...
==> invpi
:-) 0.31830988619288864
```

L'évaluation de l'expression `(define symb` e`)` peut provoquer l'impression du message `symb` (ce n'est pas à proprement parler une valeur, c'est pourquoi nous préférons "..."). L'important est la liaison du symbole "défini" `symb` à la *valeur* de l'expression e ; ce n'est pas la valeur éventuelle de l'expression `(define symb` e`)`.

Si le symbole `pi` reçoit une nouvelle valeur (il est "redéfini"), cela n'influera pas sur `invpi`. En effet, un `define` lie son premier argument (un symbole) à la *valeur* de son second argument, telle qu'elle existe *au moment où l'expression* `define` *est évaluée.*

```
==> (define pi 4.0)
:-) ...

==> invpi
:-) 0.31830988619288864
```


Symboles

Les symboles sont des "mots" qui ne sont pas des nombres.

Ils ne peuvent contenir les caractères spéciaux suivants :

```
( ) [ ] { } ; , " ' ` # /
```

Un symbole est généralement utilisé comme variable ;
sa valeur est alors l'objet qui lui est lié (s'il existe).
Le caractère spécial ' indique que le symbole n'est pas utilisé comme variable ; il est alors sa propre valeur.

L'écriture "`'pi`" abrège l'écriture "`(quote pi)`".
(Le rôle de `quote` sera vu plus loin.)

On a par exemple [[`pi`]] = 3.14... et [[`'pi`]] = `pi`.

Exemples

```
==> pi
:-) 3.1415926535

==> 'pi
:-) pi

==> (quote pi)
:-) pi

==> pi.+2
:-) error - unbound variable pi.+2

==> 'pi.+2
:-) pi.+2
```


Listes I

Syntaxe. Une liste est une suite d'objets séparés par des blancs, entourée d'une paire de parenthèses (le nombre d'objets est la longueur de la liste). Les listes sont des valeurs ; certaines d'entre elles sont aussi des expressions ; toutes les expressions composées sont des listes.
La seule liste de longueur 0 est la liste vide `()`.

Exemples : `()`, `(* 3)`, `(5 (+ (d 4)) ())`, `(+ 3 5)`, `(define pi 3.14)`, ...

– *Constructeur* **`cons`** (deux arguments) :
  Si [[β]] est une liste, alors [[`(cons` α β`)`]] est une liste dont le premier élément est [[α]] et le reste est [[β]].
– *Constructeur* **`list`** (nombre quelconque d'arguments) :
  [[`(list` α_1 ... α_n`)`]] est la liste dont les éléments sont (dans l'ordre) [[α_1]], ..., [[α_n]].

Les fonctions [[`cons`]] et [[`list`]] sont injectives.

`(list` α_1 α_2 ...α_n`)` équivaut à
`(cons` α_1 `(cons` α_2 ... `(cons` α_n `'()) ... ))`

Listes II

Notation. Une liste non vide dont le premier élément est x et dont le reste est la liste ℓ peut s'écrire `(`x `.` ℓ`)`. Les écritures

```
(0 1 2)
(0 . (1 2))
(0 1 . (2))
(0 1 2 . ())
(0 . (1 . (2)))
(0 . (1 2 . ()))
(0 1 . (2 . ()))
(0 . (1 . (2 . ())))
```

représentent toutes la même liste de longueur 3, dont les éléments sont `0`, `1` et `2`.

Cette notation sera généralisée plus loin. On note déjà que

[[`(cons` α β`)`]] = (`[[`α`]]` . `[[`β`]]`)
[[`(cons 3 7)`]] = `(3 . 7)` (paire pointée)

Reconnaisseurs de listes.

– Reconnaître une liste : prédicat `list?`
  [[`(list? l)`]] est vrai si [[`l`]] est une liste.
– Reconnaître la liste vide : prédicat `null?`
  [[`(null? l)`]] est vrai si [[`l`]] est `()`.

Listes III

Décomposition des listes.

Si [[`l`]] est une liste non vide, son premier élément est [[`(car l)`]] ;
les éléments restants forment la liste [[`(cdr l)`]].

On a donc [[`l`]] = [[`(cons (car l) (cdr l))`]].

Si [[`a`]] est un objet et [[`l`]] une liste, on a [[`(cons a l)`]] = (`[[a]]` . `[[l]]`) et [[`(car (cons a l))`]] = [[`a`]] et [[`(cdr (cons a l))`]] = [[`l`]].

```
(define l (list 3 '5 '(6 7) '3a 'define pi))        ...
l                                   (3 5 (6 7) 3a define 3.1415926535)
(car l)                             3
(cdr l)                             (5 (6 7) 3a define 3.1415926535)
(cdaddr l)                          (7)
(cons '(a b) '(6 7))                ((a b) 6 7)
(cons (car l) (cdr l))              (3 5 (6 7) 3a define 3.1415926535)
```

Booléens et prédicats I

Les booléens sont des symboles spéciaux :

`#t` pour *true* (vrai) et `#f` pour *false* (faux).

Remarque. Certains systèmes assimilent "`#f`" et "`()`" (liste vide) ; dans la suite, nous évitons cette assimilation fautive.

Les prédicats sont des fonctions prenant leurs valeurs dans les booléens. Beaucoup de prédicats numériques (`<`, `=`, `zero?`, ...) ou non (`null?`, `equal?`, ...) sont prédéfinis.

Les reconnaisseurs sont des prédicats à un argument, associés aux différentes catégories d'objets.

Voici des reconnaisseurs prédéfinis importants :

```
number?  symbol?  boolean?  list?  procedure?
```


Booléens et prédicats II

Exemples.

```
==> (> 7 3 -2 -5/2)
:-) #t
==> (> 7 -2 3 -5/2)
:-) #f

==> (= (- 3 2) (/ 6 6) 1 (+ 0 1))
:-) #t
==> (equal? l (cons (car l) (cdr l)))
:-) #t
```


Conditionnels

(On suppose que `pi` est défini et que `pu` ne l'est pas.)

```
(if (> 5 3) pi pu)              3.1415926535
(if (> 3 5) pi pu)              Error - unbound variable pu
(if (> pi pu) 3 4)              Error - unbound variable pu
(cond ((> 0 1) (/ 2 0))
      ((> 1 2) (/ 0 1))
      ((> 2 0) (/ 1 2)))        1/2
(cond)                          Error - no value
(cond ((> 0 1) 5))              Error - no value
(cond ((> pi 5) trucmuche)
       (else 4))                4
(cond ('trucmuche 3)
      (else 4))                 3
```


3. Règles d'évaluation

La règle d'évaluation des expressions est récursive :
l'évaluation d'une expression peut requérir
l'évaluation de sous-expressions.

Le processus d'évaluation renvoie en général une valeur ;
il peut aussi avoir un effet (outre l'affichage de la valeur).

Les nombres et les variables sont des cas de base.
Les combinaisons sont des cas inductifs.
Les formes spéciales sont des cas de base ou des cas inductifs.

Cas de base

L'évaluation d'un nombre donne ce nombre.
L'évaluation d'un booléen donne ce booléen.
L'évaluation d'un symbole donne la valeur liée à ce symbole,
si cette valeur existe et est affichable.

```
5/4     ==>     5/4
#f      ==>     #f
pi      ==>     3.1415926535
pa      ==>     Error - unbound variable
+       ==>     # procedure ...
car     ==>     # procedure ...
fact    ==>     # procedure ...
fict    ==>     Error - unbound variable
```


Cas inductif

L'expression à évaluer est une liste (e_0 ... e_n).
(La parenthèse ouverte est la marque du cas inductif.)

La combinaison

1. Les e_i sont évalués (soient v_i les valeurs) ;
   v_0 doit être une fonction dont le domaine contient $(v_1, \ldots, v_n)$.
2. La fonction v_0 est appliquée à $(v_1, \ldots, v_n)$.
3. Le résultat est affiché.

Une fonction (mathématique) est un ensemble de couples de valeurs ; la première est un élément du domaine, la seconde l'image correspondante.

Appliquer une fonction est produire l'image associée à un élément donné du domaine de la fonction.

Pour les fonctions prédéfinies (liées à `+`, `cons`, etc.) le processus d'application est l'exécution du code correspondant à la fonction. Le cas des fonctions définies par l'utilisateur est envisagé plus loin.

Les formes spéciales

La syntaxe est celle d'une combinaison, mais chaque forme spéciale a sa propre règle d'évaluation. Les formes spéciales sont identifiées par un mot-clef, qui est le premier élément de l'expression à évaluer.

Liste partielle

```
define                  let
if                      let*
cond                    letrec
quote                   ...
lambda                  ...
and                     ...
or                      ...
```

Remarques. Les valeurs des sous-expressions ne sont pas affichées.
En cas d'erreur, l'évaluation est interrompue et un message est affiché.

Exemples de combinaison.

Les sous-expressions de ces combinaisons sont des nombres ou des booléens ou des symboles ou des formes spéciales ou des combinaisons.

Les composants de la combinaison sont évalués,
la première valeur résultante est appliquée aux suivantes.

```
(+ 2 4)                                     6

(+ (- 5 3) (* 2 2))                         6

(cons (car '(5 6)) (cdr (list 7 8 9)))      (5 8 9)

(car (cons (car '(5 6)) (cdr (list 7 8))))  5

(car '(5 8 9))                              5

(+ (* 4 (car 5)) 3)                         Error - ...
```

Evaluation de `(cond (`c_1 e_1`) ...(`c_n e_n`))`

Les prédicats $c_1, \ldots, c_n$ sont évalués en séquence, jusqu'au premier c_k dont la valeur n'est pas `#f` (ce sera le plus souvent `#t`). La valeur du conditionnel est alors celle de l'expression correspondante e_k.

Si tous les prédicats sont évalués à `#f`, le conditionnel n'a pas de valeur. (C'est généralement lié à une erreur de programmation, même si l'évaluateur n'affiche pas `error`.) Le dernier prédicat c_n peut être le mot-clef `else` dont la valeur, dans ce contexte, est `#t`.

`(if` e_0 e_1 e_2`)` abrège `(cond (`e_0 e_1`) (else` e_2`))`

```
(cond)                      Error - no value
(cond (1 2))                2
(if (< 5 3) (/ 2 0) 5)      5
(cond ((> 0 1) (/ 2 0))
      ((> 2 0) (/ 1 2)))    1/2
(cond ((> 0 1) 5))          Error - no value
(cond (else 4))             4
```


Evaluation de `(and` e_1 `...`e_n`)`

Les e_i sont évalués dans l'ordre et la première valeur fausse est retournée ; les valeurs suivantes ne sont pas calculées. S'il n'y a pas de valeurs fausses, la dernière valeur calculée est retournée. S'il n'y a pas de valeurs du tout, `#t` est retourné.

```
(and (= 2 2) (< 2 1) (/ 2 0) (+ 2 3))      #f
(and (= 2 2) (+ 2 3) (/ 2 0) (< 2 1))      Error - Division by zero
(and (= 2 2) (< 2 1) (/ 2 0) (+ 2 3))      #f
(and (= 2 2) (+ 2 3))                      5
(and (+ 2 3) (= 2 2))                      #t
(and)                                      #t
(if (and (= 2 2) (+ 2 3)) 'vrai 'faux)     vrai
```


Evaluation de `(or` e_1 `...`e_n`)`

Les e_i sont évalués dans l'ordre et la première valeur non fausse est retournée ; les valeurs suivantes ne sont pas calculées. S'il n'y a pas de valeurs non fausses, ou pas de valeurs du tout, `#f` est retourné.

```
(or (< 2 1) (= 2 2) (/ 2 0) (+ 2 3))       #t
(or (< 2 1) (+ 2 3) (/ 2 0) (= 2 2))       5
(or (< 2 1) (/ 2 0) (+ 2 3) (= 2 2))       Error - Division by zero
(or (= 2 2) (+ 2 3) (< 2 1))               #t
(or)                                       #f
(or #f (< 2 1) #f)                         #f
(if (or (+ 2 3) 1 2) 'vrai 'faux)          vrai
```


Evaluation de `(quote` e`)`.
La valeur est simplement e, à ne pas confondre avec la *valeur* de e !
La forme spéciale `(quote` e`)` s'abrège en `'`e.

```
(quote +)                  +
'+                         +
+                          # procedure
(quote 3)                  3
'3                         3
3                          3
(quote pi)                 pi
'pi                        pi
pi                         3.1415926535
'''pi                      ''pi
(car (a b c))              Error - unbound var
(car '(a b c))             a
(cons 'a '(b c))           (a b c)
(car (cons 'a '(b c)))     a
(car '(cons a (b c)))      cons
(car (cons a (b c)))       Error - unbound var
```

"Evaluation" de la forme `define`.
La valeur de (`define` s e) n'est pas spécifiée.
L'évaluation a pour *effet* de lier la valeur $[[e]]$ au symbole s.
Une forme spéciale `define` peut apparaître à l'intérieur d'une autre forme, mais nous n'utilisons pas cette possibilité par la suite.
Dans notre cadre, toute nouvelle liaison (par `define`) d'une valeur à un symbole `x` rend inaccessible la liaison antérieure éventuelle à ce symbole.

```
x                          Error - unbound var
(define x (+ 4 1))         ...
x                          5
(+ x 3)                    8
(define x +)               ...
x                          # procedure
'x                         x
(x 2 3 4)                  9
(define 'x 7)              Error
(define x (x 4 4))         ...
x                          8
```


4. La forme `lambda`

La fonction qui à son unique argument associe le carré de sa valeur s'écrit parfois $\lambda x\ .\ x * x$.
La variable x n'a qu'un rôle de marque-place et peut être renommée.

En Scheme on écrit `(lambda (x) (* x x))`.
La liste `(x)` est la *liste des paramètres*.
La liste `(* x x)` est le *corps* de la forme `lambda`.

```
pi                                 3.1415926535
(lambda (y) (* y y))               # procedure
((lambda (y) (* y y)) 12)          144
((lambda (pi) (* pi pi)) 12)       144
((lambda (car) (* car car)) 12)    144
y                                  Error - unbound variable y
pi                                 3.1415926535
car                                # procedure
```


Règle de calcul

Evaluation de (`(`e_0 e_1 ... e_n`)`, e_0 = `(lambda (x1 ... xn)` M`)`.

On évalue d'abord les expressions e_i, ce qui donne les valeurs v_i ($i = 0, 1, \ldots, n$).
La valeur v_0 est une fonction à n arguments.
On applique v_0 à la suite des valeurs $(v_1, \ldots, v_n)$;
le résultat est la valeur de la combinaison.

Le processus d'application est, par définition, le processus d'évaluation de l'expression M (le *corps* de la forme `lambda`), dans laquelle les variables `x1`, ..., `xn` sont liées aux valeurs $v_1, \ldots, v_n$, respectivement.

Exemple.

```
x                              Error - unbound variable x
y                              9
((lambda (x y) (+ 2 x y)) 3 7) 12
(+ 2 3 7)                      12
x                              Error - unbound variable x
y                              9
```


Remarque importante. La liaison des `xi` aux v_i disparaît dès que le processus d'application se termine. Les liaisons antérieures éventuelles des `xi` sont alors à nouveau en vigueur. Le caractère temporaire des liaisons des paramètres permet d'éviter les "téléscopages" de noms.

Exemple. Dans un environnement où `x` a une valeur numérique, l'expression

```
((lambda (x) (* 2 x))
 ((lambda (x) (- 9 ((lambda (x) (+ 3 x)) x))) x))
```

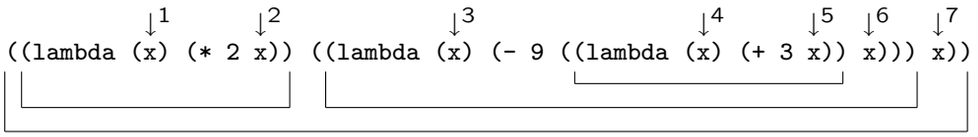
a la même valeur que l'expression

```
((lambda (w) (* 2 w))
 ((lambda (v) (- 9 ((lambda (u) (+ 3 u)) v))) x))
```

(Le fait que `u`, `v` et `w` aient des valeurs ou non dans cet environnement est sans importance.)

Remarque. Un environnement est un ensemble de liaisons variable-valeur.

Portée, réduction et combinaison I



Portée, réduction et combinaison II

```
==> (define x 5)
:-) ...
```

Les expressions qui suivent ont toutes pour valeur 2.

```
((lambda (x) (* 2 x)) ((lambda (x) (- 9 ((lambda (x) (+ 3 x)) x))) x))

((lambda (w) (* 2 w)) ((lambda (v) (- 9 ((lambda (u) (+ 3 u)) v))) x))

(* 2 ((lambda (v) (- 9 ((lambda (u) (+ 3 u)) v))) x))
((lambda (w) (* 2 w)) (- 9 ((lambda (u) (+ 3 u)) x)))
((lambda (w) (* 2 w)) ((lambda (v) (- 9 (+ 3 v))) x))

(* 2 (- 9 ((lambda (u) (+ 3 u)) x)))
(* 2 ((lambda (v) (- 9 (+ 3 v))) x))
((lambda (w) (* 2 w)) (- 9 (+ 3 x)))

(* 2 (- 9 (+ 3 x)))
```


Occurrences libres, occurrences liées I

Expression mathématique :

$$E =_{def} 1 - \int_0^x \sin(u)\,\mathrm{d}u$$

le symbole x a une valeur (dont dépend celle de E) ; le symbole u n'a pas de valeur ; Remplacer (partout !) u par t ne change pas la valeur de E. Les symboles 1, 0, sin et $\int \ldots \mathrm{d} \ldots$ ont des valeurs (nombres pour les deux premiers, fonction pour le troisième, opérateur d'intégration pour le dernier).

En logique, la valeur de vérité de $\forall x\, P(x,y)$ dépend de la valeur de la "variable libre" y (et de l'interprétation de la constante P) mais pas de la valeur de la "variable liée" x qui peut être renommée : $\forall x\, P(x,y)$ et $\forall z\, P(z,y)$ sont des formules logiquement équivalentes.

Le même genre de distinction doit se faire en Scheme. Pour évaluer l'expression `((lambda (x) (+ x y)) 5)`, il sera nécessaire que `y` ait une valeur ; par contre, la valeur éventuelle de `x` est sans importance. On peut d'ailleurs remplacer les deux occurrences de `x` par `z`, par exemple.

Occurrences libres, occurrences liées II

Les notations Scheme sont sur ce point plus flexibles que les notations mathématiques. On ne peut écrire

$$\int_0^x \sin(x)\,\mathrm{d}x \quad \text{au lieu de} \quad \int_0^x \sin(u)\,\mathrm{d}u$$

mais on peut écrire

`((lambda (x) (+ x 5)) x)` au lieu de `((lambda (u) (+ u 5)) x)`

(même si la seconde notation est naturellement préférable).
L'utilisateur doit se méfier des "téléscopages de noms".

On observe aussi que la valeur de

```
((lambda (u) (+ u 5)) (* x 3))
```

dépend de celle de `x` mais que la valeur de

```
(lambda (x) ((lambda (u) (+ u 5)) (* x 3)))
```

n'en dépend pas.

Occurrences libres, occurrences liées III

Toute occurrence de `x` dans le corps `E` de la forme `(lambda (... x ...) E)` est dite *liée*. Par extension, une occurrence de `x` est liée dans une expression α si cette expression comporte une sous-expression (forme `lambda`) dans laquelle l'occurrence en question est liée. Une occurrence non liée (en dehors d'une liste de paramètres) est *libre*.

Dans l'expression `((lambda (x) (- 9 ((lambda (x) (+ 3 x)) x))) x)`, les deux premières occurrences de `x` sont des paramètres, les troisième et quatrième occurrences de `x` sont liées, la cinquième occurrence de `x` est libre.

Dans l'expression `(* 9 ((lambda (x) (+ 3 x)) x) x)`, la première occurrence de `x` est un paramètre, la deuxième occurrence de `x` est liée, les troisième et quatrième occurrences de `x` sont libres.

Les variables liées sont habituelles en logique, et aussi en mathématique (indice de sommation, variable d'intégration).

Statut "première classe" des procédures

Les procédures, valeurs des formes `lambda`, héritent de toutes les propriétés essentielles des valeurs usuelles comme les nombres. En particulier, on peut définir une procédure admettant d'autres procédures comme arguments, et renvoyant une procédure comme résultat. De même, on peut lier une procédure à un symbole, en utilisant `define`.

Ceci évoque les mathématiques (les domaines de fonctions ont le même "statut" que les domaines de nombres), mais contraste avec les langages de programmation usuels, qui ne permettent généralement pas, par exemple, de renvoyer une procédure comme résultat de l'application d'une autre procédure à ses arguments.

```
(define square (lambda (x) (* x x))) ==>  ...
square                               ==>  # procedure
(square (+ 4 1))                     ==>  25
```

Evaluer `square` et `(+ 4 1)`, ce qui donne les valeurs v_0 et v_1.
Appliquer la valeur v_0 (fonction unaire) à la valeur $v_1 = 5$
i.e. évaluer `(* x x)` où `x` est lié à (remplacé par) `5`
i.e. évaluer `(* 5 5)`.

Remarque. Conceptuellement, la valeur-procédure v_0 associée à la variable `square` pourrait être la table (infinie) des couples (n, n^2). Un tel objet n'est pas affichable, et Scheme affichera simplement "`# procedure`".

Arguments et valeurs fonctionnels I

```
(define compose
  (lambda (f g)
    (lambda (x) (f (g x)))))        ...

(compose car cdr)                   # procedure

((compose car cdr) '(1 2 3 4))      2

(cadr '(1 2 3 4))                   2

((compose (compose car cdr) cdr)
 '(1 2 3 4))                        3

((compose (compose car cdr)
          (compose cdr cdr))
 '(1 2 3 4))                        4
```


Arguments et valeurs fonctionnels II

Etant donné une fonction `f` et un incrément `dx`, une approximation de la fonction dérivée en `x` est (la valeur de) `(/ (- (f (+ x dx)) (f x)) dx)`. On peut définir la fonction `deriv-p` (dérivée en un point) comme suit :

```
(define deriv-p
  (lambda (f x dx) (/ (- (f (+ x dx)) (f x)) dx)))
```

On a

```
(deriv-p square 10 0.00001)  ==>  20.0000099
```

Il est plus élégant de définir d'emblée l'*opérateur* `deriv` :

```
(define deriv
  (lambda (f dx)
    (lambda (x) (/ (- (f (+ x dx)) (f x)) dx))))
```

On a alors

```
((deriv square 0.00001) 10)  ==>  20.0000099
```

5. Récursivité

La notion de fonction est cruciale en programmation parce qu'elle permet de résoudre élégamment de gros problèmes en les ramenant à des problèmes plus simples.

De même, la notion de fonction est cruciale en mathématique parce qu'elle permet de construire et de nommer des objets complexes en combinant des objets plus simples.

La définition *hypo* $=_{def}\ \lambda x, y : \sqrt{x^2 + y^2}$
de même que le programme

```
(define hypo
  (lambda (x y) (sqrt (+ (square x) (square y)))))
```

sont des exemples classiques de fonctions. La définition (opérationnelle) de la longueur de l'hypoténuse suppose les définitions préalables de l'addition, de l'élévation au carré et de l'extraction de la racine carrée.

Remarque. L'ordre dans lequel on *définit* les procédures n'a pas d'importance. On peut définir (mais non utiliser) `hypo` avant de définir `square`.

Idée de base I

L'idée de la récursivité est d'utiliser, dans la définition d'un objet relativement complexe, non seulement des objets antérieurement définis, mais aussi l'objet à définir lui-même.
Le risque de "cercle vicieux" existe, mais n'est pas inévitable. On peut faire une analogie avec les égalités et les équations. Dans un contexte où f, a et b sont définis, on peut définir x par l'égalité

$$x = f(a, b)$$

Par contre, l'égalité

$$x = f(a, x)$$

ne définit pas nécessairement un et un seul objet x ; même si c'est le cas, le procédé de calcul de x peut ne pas exister.
Un procédé de calcul parfois utilisé pour résoudre l'équation $x = f(a, x)$ consiste à construire un préfixe plus ou moins long de la suite

$$x_0,\ x_1 = f(a, x_0),\ x_2 = f(a, x_1),\ \ldots,\ x_{n+1} = f(a, x_n), \ldots$$

et à considérer que le dernier terme calculé est (proche de) la solution cherchée. Ce procédé fournit rapidement, par exemple, une bonne approximation de la solution de l'équation $x = \cos x$.

Idée de base II

Le procédé d'approximation peut aussi être utilisé pour calculer des fonctions plutôt que des nombres, et donc pour résoudre des équations fonctionnelles du type

$$f\ =\ \Phi(g, f)\,.$$

L'équation différentielle $y' = f(x, y)$ (avec $y(x_0) = y_0$) peut s'écrire

$$y(x)\ =\ y_0 + \int_{x_0}^{x} f(t, y(t))\,\mathrm{d}t$$

ce qui permet souvent une résolution approchée.

Exercice. Résoudre l'équation $y' = y$, avec $y(0) = 1$.

Idée de base III

En programmation, on doit à toute construction associer un procédé de calcul clair, précis et raisonnablement efficace. Pour cette raison, la définition récursive de fonctions sera limitée à des instances particulières du schéma $f = \Phi(g, f)$.

En pratique, on constatera que l'évaluation de $f(x)$ nécessitera la détermination préalable d'un ensemble de valeurs $f(y_1), \ldots, f(y_n)$ où les $y_1, \ldots, y_n$ sont, en un certain sens, "plus simples" que x.

Si le domaine de la fonction f à définir est tel que tout élément n'admet qu'un ensemble fini d'éléments "plus simples", on conçoit que le risque de "tourner en rond" pourra être évité.

Deux exemples classiques I

Définition de la factorielle (fonction de $\mathbb{N}$ dans $\mathbb{N}$) :

$$n! \;=\; [\text{if } n = 0 \text{ then } 1 \text{ else } n * (n-1)!\,]\,.$$

Exemple d'exploitation de la définition :

$$3! = 3*2! = 3*2*1! = 3*2*1*0! = 3*2*1*1 = 6\,.$$

Deux exemples classiques II

Définition de la suite de Fibonacci :

$$f_n \;=\; [\text{if } n < 2 \text{ then } n \text{ else } f_{n-1} + f_{n-2}\,]\,.$$

$$\begin{array}{l} f_0 = 0\,,\\ f_1 = 1\,,\\ f_2 = 1+0 = 1\,,\\ f_3 = 1+1 = 2\,,\\ f_4 = 2+1 = 3\,,\\ f_5 = 3+2 = 5\,,\\ f_6 = 5+3 = 8\,,\\ \ldots \end{array}$$

Dans les deux cas, "plus simple" s'identifie à "plus petit".
Chaque entier *naturel* n'admettant qu'un nombre fini de naturels plus petits, on conçoit que ces définitions *récursives* de fonctions soient "acceptables" (le calcul se termine).

Deux exemples classiques III

Tout ceci se traduit aisément en `Scheme` :

```
(define fact
  (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))     ...

(fact 0)                                                    1
(fact 4)                                                   24
(fact (fact 4))                          620448401733239439360000
(fact -1)                                          Aborting! ...

(define fib
  (lambda (n)
    (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))      ...

(fib  0)                                                    0
(fib  6)                                                    8
(fib 12)                                                  144
```


Double rôle de `define`

La forme **`(define var`** *exp*`)` peut être évaluée dans deux cadres distincts.

- L'expression *exp* ne comporte pas d'occurrence libre de `var`. Dans ce cas non récursif, *exp* a une valeur v au moment où la forme `define` est évaluée, et le seul rôle de cette évaluation est de provoquer la liaison de v à `var`.
- L'expression *exp* est une forme `lambda`, pouvant comporter des occurrences libres de `var`. Dans ce cas, *exp* n'a pas de valeur (du moins, pas de valeur utile) avant l'évaluation de la forme `define`. Cette évaluation a alors le double rôle de définir une procédure récursive et de la lier à (de lui donner le nom) `var`.

Avant l'évaluation de la forme spéciale
`(define fact (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))`
la forme `(lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))` n'a pas de valeur utile puisque `fact` n'a pas de valeur. Après l'évaluation, la procédure de calcul de la factorielle est liée à (est la valeur de) la variable `fact`.

Processus de calcul I

Simulation d'un calcul récursif :

```
(fact 3)
(if (zero? 3) 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
...
(* 3 (* 2 (fact 1)))
...
(* 3 (* 2 (* 1 (fact 0))))
...
(* 3 (* 2 (* 1 1)))
...
6
```

On conçoit que ce type de calcul requiert un espace-mémoire dont la taille dépend (ici, linéairement) de la valeur de l'argument.
On note aussi que l'introduction de la récursivité ne semble pas exiger de mécanisme particulier pour appliquer une fonction à des arguments : les règles déjà introduites suffisent. Cet avantage peut entraîner des déboires pour l'utilisateur peu soucieux du processus de calcul.

Processus de calcul II

Scheme accepte les définitions

```
(define f (lambda (x) (f x)))
(define g (lambda (x) (g (+ x 1))))
(define h (lambda (x) (+ (h x) 1)))
```

Evaluer `(f 0)`, `(g 1)` ou `(h 2)` ne donne rien ...
... sauf un gaspillage de ressources !
Même chose pour `(fact -1)` : procédure utile, mais argument inapproprié.

L'utilisateur peut éviter ce risque en vérifiant que tout appel pour des valeurs données donne lieu à un nombre fini d'appels subséquents, pour des valeurs "plus simples". Des schémas de programmes récursifs existent, qui garantissent la propriété de terminaison.

Exercice. Que penser, dans le domaine des réels positifs, de la définition suivante :

$$f(0) = a\,, \quad f(x) = g(b, f(x/2)) \text{ si } x > 0\,?$$

Processus de calcul III

La terminaison du calcul de `(fib` n`)` ($n \in \mathbb{N}$) : se démontre par récurrence.

L'efficacité du calcul est inacceptable :

```
(fib 9)
(+ (fib 8) (fib 7))
(+ (+ (fib 7) (fib 6)) (fib 7))
...
```

L'expression `(fib 7)` sera évaluée deux fois. En fait,
1 évaluation de `(fib 9)` implique
1 évaluation de `(fib 8)`,
2 évaluations de `(fib 7)`,
3 évaluations de `(fib 6)`,
5 évaluations de `(fib 5)`, etc.

Temps de calcul : proportionnel à la valeur calculée $f_n \simeq 1.6^n$.

Deux remèdes possibles seront vus plus loin :
1) Utiliser un autre principe de calcul (moins naïf) ;
2) Forcer l'interpréteur à réutiliser les résultats intermédiaires.

Processus de calcul IV

On sait que les nombres de Fibonacci peuvent se calculer plus efficacement, au moyen d'une simple boucle. L'algorithme correspondant se traduit immédiatement en scheme :

```
(define fib_a
  (lambda (n a b)
    (if (zero? n)
        a
        (fib_a (- n 1) b (+ a b)))))
```

Si `n`, `a` et `b` ont pour valeurs respectives n, f_i et f_{i+1}, alors `(fib_a n a b)` a pour valeur f_{n+i}.
En particulier, `(fib_a n 0 1)` a pour valeur f_n.

Signalons déjà que ce type de récursivité "dégénérée" se reconnaît par une simple analyse de la syntaxe du programme. Cette analyse permet à l'interpréteur d'exécuter le programme aussi efficacement qu'une simple boucle en pascal, sans consommation inutile de mémoire.

Récursivité croisée

Exemple classique.

```
(define even?
  (lambda (n) (if (zero? n) #t (odd? (- n 1)))))

(define odd?
  (lambda (n) (if (zero? n) #f (even? (- n 1)))))
```

Processus de calcul.

```
(odd? 4) -> (even? 3) -> (odd? 2) -> (even? 1) -> (odd? 0) -> #f
```

Rappel. L'ordre dans lequel on définit les procédures n'a pas d'importance. Seule exigence (naturelle) : les procédures doivent être définies avant d'être utilisées (appliquées à des arguments). Le double rôle de `define` existe dès qu'il y a récursivité, croisée ou non.

6. Récursivité structurelle

Principe de la récursivité structurelle

Le système "accepte" toute définition récursive syntaxiquement correcte, même si la procédure associée donne lieu à des calculs infinis. L'utilisateur doit savoir si, dans un domaine donné, le calcul se terminera toujours. Pour les domaines usuels, des schémas existent qui garantissent la terminaison.

Cas particulier : les schémas *structurels*, basés sur la manière dont les objets du domaine de calcul sont construits.

Le processus d'évaluation réduit le calcul de $f(v)$ au calcul de $f(v_1), \ldots, f(v_n)$ où les v_i sont des *composants (immédiats)* de v. Cette technique est sûre tant que l'on se limite aux domaines dont les objets ont un nombre fini de composants (immédiats ou non), clairement identifiés.

Domaines usuels de base : Nombres naturels
Listes
Expressions symboliques

En plus : domaines dérivés des précédents, surtout par produit cartésien.

Récursivité structurelle : les naturels I

Conceptuellement, les naturels sont construits à partir de 0 et de la fonction successeur. 0 n'a pas de composant (objet de base) ; 16 est le seul composant immédiat de $17 = succ(16)$.

Schéma de base

```
(define F
  (lambda (n u)
    (if (zero? n)
        (G u)
        (H (F (- n 1) (K n u))
           n
           u))))
```

Les fonctions `G`, `H` et `K` sont supposées déjà définies.
`u` représente une suite de 0, 1 ou plusieurs arguments ;
(une suite de 0 argument se réduit à rien !).

Le calcul de `(F 0 u)` n'implique pas d'appel récursif.
Le calcul de `(F n u)` implique celui de `(F (- n 1) (K n u))` si `n` n'est pas nul.

Récursivité structurelle : les naturels II

Schéma de base simplifié

Le cas où `u` est absent suffit souvent :

```
(define F
  (lambda (n)
    (if (zero? n)
        c
        (H (F (- n 1))
           n))))
```

Récursivité structurelle : les naturels III

Exemples :

```
(define harmonic-sum
  (lambda (n)
    (if (zero? n)
        0
        (+ (/ 1 n) (harmonic-sum (- n 1))))))

(define mult
  (lambda (n u)
    (if (zero? n)
        0
        (+ u (mult (- n 1) u)))))
```


Récursivité structurelle : les naturels IV

Exemples :

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (- n 1))))))

(define cbin
  (lambda (n u)
    (if (zero? n)
        1
        (/ (* (cbin (- n 1) (- u 1)) u) n))))
```


Récursivité structurelle : les naturels V

Les schémas sont d'abord des schémas de pensée ; ils suggèrent d'exprimer $f(n)$ en termes d'expressions indépendantes de f, mais aussi de $f(n-1)$, si $n > 0$. Les schémas imposent en outre la syntaxe du programme : le programmeur doit seulement définir les fonctions `G`, `H` et `K`.

Exemple

```
(define cbin
  (lambda (n u)
    (if (zero? n)
        1
        (/ (* (cbin (- n 1) (- u 1)) u) n))))
```

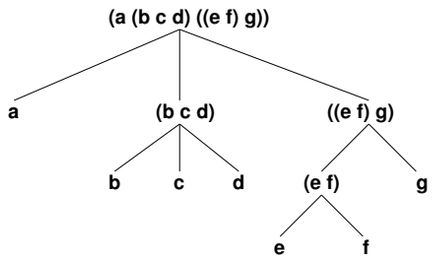

Récursivité structurelle : les naturels VI

Le même exemple . . .

```
(define F
  (lambda (n u)
    (if (zero? n)
        (G u)
        (H (F (- n 1) (K n u)) n u))))

(define G (lambda (u) 1))
(define K (lambda (n u) (- u 1)))
(define H (lambda (r n u) (/ (* r u) n)))
(define cbin F)
```

Listes : représentation arborescente

Il existe une correspondance naturelle entre les listes et les arbres.
Chaque nœud de l'arbre a un nombre fini quelconque de fils.



ATTENTION !
Cette représentation commode des listes n'est pas celle employée en machine !

Remarque. Un arbre réduit à sa racine et étiqueté par un symbole atomique ne sera pas représenté par une liste mais par ce symbole.

Remarque. Conceptuellement, seuls les nœuds terminaux de l'arbre représenté ici sont étiquetés ; les étiquettes attachées aux nœuds internes sont synthétisées à partir des étiquettes des nœuds successeurs. Les arbres dont les nœuds internes sont étiquetés (indépendamment des feuilles) forment un autre type de données.

Listes : récursivité superficielle I

[[l]] est soit `()`, soit [[`(cons (car l) (cdr l))`]].

Toute liste s'obtient à partir de `()` et de `cons` ; la règle de construction est simple : `([a1]] [[a2]] ... [[an]])` s'obtient en évaluant
`(cons a1 (cons a2 (cons ... (cons an '()) ...)))`

Schéma de base

```
(define F
  (lambda (l u)
    (if (null? l)
        (G u)
        (H (F (cdr l) (K l u))
           l
           u))))
```

Les fonctions `G`, `H` et `K` sont supposées déjà définies ;
`u` représente une suite de 0, 1 ou plusieurs arguments.

Calculer `(F () u)` n'implique pas d'appel récursif.
Calculer `(F l u)` implique l'appel `(F (cdr l) (K l u))` si `l` n'est pas vide.

Listes : récursivité superficielle II

Le cas où $|u| = 0$ suffit souvent :

```
(define F
  (lambda (l)
    (if (null? l)
        c
        (H (F (cdr l))
           l))))
```


Listes : récursivité superficielle III

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

(define append
  (lambda (l v)
    (if (null? l) v (cons (car l) (append (cdr l) v)))))

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

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

Listes : récursivité superficielle IV

```
(define map
  (lambda (f1 l)
    (if (null? l)
        '()
        (cons (f1 (car l)) (map f1 (cdr l))))))
```

s'obtient en instanciant le schéma

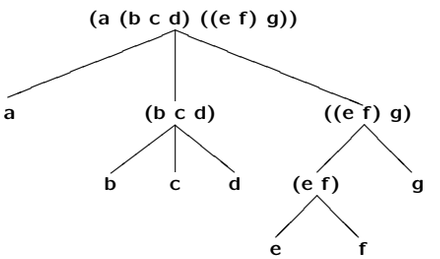
```
(define F
  (lambda (l u)
    (if (null? l)
        (G u)
        (H (F (cdr l) (K l u))
           l
           u))))

(define G (lambda (u) '()))
(define H (lambda (r l u) (cons (u (car l)) r)))
(define K (lambda (l u) u))
(define map (lambda (f1 l) (F l f1)))
```


Retournement superficiel ("reverse")

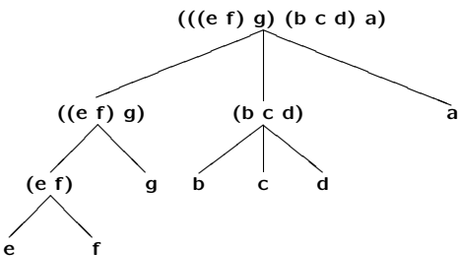
```
(define reverse
  (lambda (l)
    (if (null? l)
        '()
        (append (reverse (cdr l)) (list (car l))))))
```

L'appel récursif porte sur le cdr seul.

Argument :



Résultat :



Tri par insertion I

```
(define sort
  (lambda (l comp)   ;; argument procédural
    (if (null? l)
        l
        (insert (car l) (sort (cdr l) comp) comp))))

(define insert
  (lambda (x l comp)
    (cond ((null? l) (list x))
          ((comp x (car l)) (cons x l))
          (else (cons (car l) (insert x (cdr l) comp))))))

(insert 3 '(0 2 3 3 5 7 8 9) <)      (0 2 3 3 3 5 7 8 9)

(sort '(8 3 5 7 2 3 9 0) <=)         (0 2 3 3 5 7 8 9)

(sort '(8 3 5 7 2 3 9 0) >=)         (9 8 7 5 3 3 2 0)
```


Tri par insertion II

```
(define insert
  (lambda (x l comp)
    (if (null? l)
        (list x)
        (if (comp x (car l))
            (cons x l)
            (cons (car l) (insert x (cdr l) comp))))))

(define F
  (lambda (l u1 u2)
    (if (null? l)
        (G u1 u2)
        (H (F (cdr l) (K1 l u1 u2) (K2 l u1 u2)) l u1 u2))))

(define G (lambda (u1 u2) (list u1)))
(define H
  (lambda (r l u1 u2) (if (u2 u1 (car l)) (cons u1 l) (cons (car l) r))))
(define K1 (lambda (l u1 u2) u1))
(define K2 (lambda (l u1 u2) u2))
(define insert (lambda (x l comp) (F l x comp)))
```

Tri par insertion III

```
(define sort
  (lambda (l comp)   ;; argument procédural
    (if (null? l)
        l
        (insert (car l) (sort (cdr l) comp) comp))))

(define F
  (lambda (l u)
    (if (null? l)
        (G u)
        (H (F (cdr l) (K l u)) l u))))

(define G (lambda (u) '()))
(define H (lambda (r l u) (insert (car l) r u)))
(define K (lambda (l u) u))
(define sort (lambda (l comp) (F l comp)))
```


Tri par insertion (ordre lexicographique)

```
(sort '("bb" "ac" "bc" "acb") string<=?)          ("ac" "acb" "bb" "bc")

(define lex
  (lambda (str-ord iden)  ;; arguments procéduraux
    (lambda (u v)
      (cond ((null? u) #t)
            ((null? v) #f)
            ((str-ord (car u) (car v)) #t)
            ((iden (car u) (car v)) ((lex str-ord iden) (cdr u) (cdr v)))
            (else #f)))))

(define numlex (lex < =))
(define alpha (lex string<? string=?))

(sort '((2 3) (1 4) (2 3 2) (1 3)) numlex)    ((1 3) (1 4) (2 3) (2 3 2))

(sort '(("acb" "ac") ("ac" "acb") ("ac")) alpha)
                                        (("ac") ("ac" "acb") ("acb" "ac"))
```


Récursivité profonde sur les listes I

Schéma de base

```
(define F
  (lambda (l u)
    (cond ((null? l) (G u))
          ((atom? (car l))
           (H (F (cdr l) (K l u))
              l
              u))
          ((list? (car l))
           (J (F (car l) (Ka l u))
              (F (cdr l) (Kd l u))
              l
              u)))))
```

Schéma simplifié

```
(define F
  (lambda (l)
    (cond ((null? l) c)
          ((atom? (car l))
           (H (F (cdr l)) l))
          ((list? (car l))
           (J (F (car l))
              (F (cdr l))
              l)))))
```

Remarque. On exclut (provisoirement) de rencontrer dans la liste `l` des objets qui ne soient ni des listes ni des atomes.

Récursivité profonde sur les listes II

```
(define flat_l
  (lambda (l)
    (cond ((null? l) '())
          ((atom? (car l))
           (if (null? (car l))
               (flat_l (cdr l))
               (cons (car l)
                     (flat_l (cdr l)))))
          ((list? (car l))
           (append (flat_l (car l))
                   (flat_l (cdr l)))))))

(flat_l 5)                          Error - 5 passed as argument to car

(flat_l '())                        ()

(flat_l '(0 (1 2) ((((3))) 4) 5))   (0 1 2 3 4 5)
```

Récursivité profonde sur les listes III

Les listes ont pour éléments des objets d'un certain type (par exemple, des atomes) et aussi d'autres listes. On peut décider d'adjoindre à ce domaine celui des objets (atomes). Cela revient à admettre les arbres réduits à une racine (qui est en même temps l'unique feuille).
Les schémas précédents s'adaptent facilement.

```
(define F                          (define F
  (lambda (l)                        (lambda (l)
    (cond ((null? l) c)                (cond ((null? l) c)
          ((atom? (car l))                   ((atom? l)
           (H (F (cdr l)) l))                 (G l))
          ((list? (car l))                   ((list? l)
           (J (F (car l))                     (J (F (car l))
              (F (cdr l))                        (F (cdr l))
              l)))))                             l)))))
```

Remarque. Le remplacement de `list?` par `pair?` améliore l'efficacité.

Récursivité profonde sur les listes IV

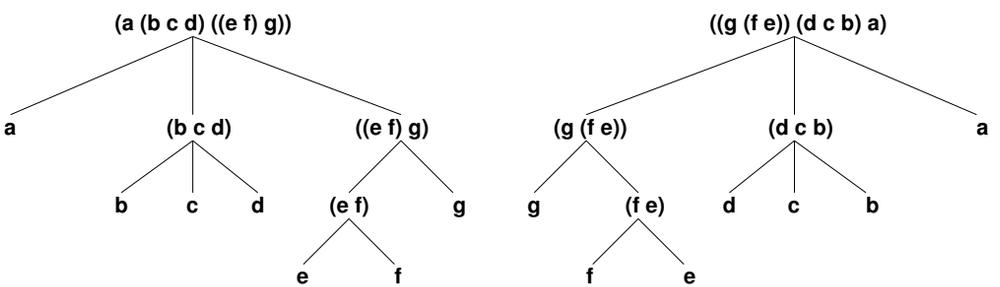
```
(define flat_le
  (lambda (l)
    (cond ((null? l) '())
          ((atom? l) (list l))
          ((list? l) (append (flat_le (car l)) (flat_le (cdr l)))))))

(flat_le 5)                          (5)

(flat_le '())                        ()

(flat_le '(0 (1 2) ((((3))) 4) 5))   (0 1 2 3 4 5)
```


Retournement profond ("`deeprev`")

```
(define deeprev
  (lambda (l)
    (cond ((null? l) '())
          ((atom? l) l)
          ((list? l)
           (append (deeprev (cdr l))
                   (list (deeprev (car l))))))))
```

L'appel récursif porte sur le `car` **et** le `cdr`.

Argument *Résultat*



Remarque sur les schémas

Suivre un schéma "à la lettre", c'est-à-dire se limiter strictement à instancier les paramètres qu'il contient, rend la programmation particulièrement méthodique et sûre.
On peut cependant, pour diverses raisons, garder "l'esprit" d'un schéma sans respecter la lettre (sa syntaxe précise). Un exemple classique est le programme `flatten`, version équivalente mais plus efficace de `flat_le` :

```
(define flatten
  (lambda (l u)
    (cond ((null? l) u)
          ((atom? l) (cons l u))
          ((list? l) (flatten (car l) (flatten (cdr l) u))))))
```

On montre facilement que les valeurs de `(flatten l u)` et de `(append (flat_le l) u)` sont égales pour toutes listes `l` et `u` ; en particulier, `(flatten l '())` et `(flat_le l)` ont même valeur. Par contre, `flatten` n'est pas une instance du schéma, même s'il s'en inspire nettement.

On concilie méthode, discipline et créativité ...

Récursivité structurelle généralisée I

Le principe de la récursivité structurelle est que la structure du programme est calquée sur celle des données.

Exprimer $f(x,\ldots)$ *en termes de* $\{f(y,\ldots) : y \prec x\}$
où $y \prec x$ *signifie* y *composant de* x*.*

On peut généraliser :
– Admettre aussi les composants non immédiats :
  `(- n 1)`, mais aussi `(- n 2)`, `(/ n 2)`, ...
  `(cdr l)`, mais aussi `(cddr l)`, ...
– Admettre plusieurs appels récursifs.
– Admettre les appels imbriqués.
La terminaison reste garantie mais pas l'efficacité !

Récursivité structurelle généralisée II

Attention aux erreurs grossières, telles les évaluations de
`(f (- n 2))` avec `n` ≤ 1, et de
`(f (cddr l))` où `l` comporte moins de deux éléments.

```
(define fib ;; inefficace
  (lambda (n)
    (if (< n 2)
        n
        (+ (fib (- n 1)) (fib (- n 2))))))

(define exp ;; très efficace
  (lambda (m n)
    (cond ((zero? n) 1)
          ((even? n) (exp (* m m) (/ n 2)))
          ((odd? n) (* m (exp m (- n 1)))))))
```


Récursivité structurelle mixte

Le principe de la récursivité structurelle est :
Exprimer $f(x,\ldots)$ *en termes de* $\{f(y,\ldots) : y \prec x\}$ *;*
l'induction porte sur un seul argument.

Le principe de la récursivité structurelle mixte est :
Exprimer $f(x1, x2, \ldots)$ *en termes de*
$\{f(y1, x2, \ldots), f(x1, y2, \ldots), f(y1, y2, \ldots) : y1 \prec x1 \wedge y2 \prec x2\}$.
L'induction porte sur plusieurs arguments ; si $f(a, b)$ *dépend de* $f(c, d)$*, alors* $c \prec a \wedge d \preceq b$ *ou* $c \preceq a \wedge d \prec b$*.*
La terminaison reste garantie.

```
(define gcd
  (lambda (x y)
    (cond ((= x y) x)
          ((> x y) (gcd (- x y) y))
          ((< x y) (gcd x (- y x))))))
```

Exemples classiques (voir plus loin) : *knapsack*, *money change*, ...

Séparation fonctionnelle I

On peut "dérécursiver" le schéma classique
$fact(n) :=$ if $n = 0$ then 1 else $n * fact(n-1)$ en l'écriture
$fact_m(n) :=$ if $n = 0$ then 1 else $n * fact_{m-1}(n-1)$.

La récursivité revient (sous forme dégénérée) pour définir globalement les fonctions $fact_m$:

```
(define f
  (lambda (m c)
    (if (zero? m)
        c
        (f (- m 1) (lambda (n) (if (= n 0) 1 (* n (c (- n 1))))))))))
```

Si [[m]] $= m$ et [[c]] $= fact_p$, alors [[(f m c)]] $= fact_{p+m}$

```
(define fact0 'emptyfunction)
(define fact (lambda (n) ((f (+ n 1) fact0) n)))
```

On observe que $fact_m$ a pour domaine $\{0, 1, \ldots, m-1\}$.
Sur ce domaine, on a $fact_m(n) = n!$.

Séparation fonctionnelle II

Exemples

```
(define fact8 (f 8 fact0))

(fact8 7)   5040
(fact8 8)   Error
(fact 8)    40320
```

On a séparé
– le *calcul* de la fonction $fact_p$
– l'*application* de cette fonction à un argument.

On a $fact(n) = fact_p(n)$ si $p > n$; on choisit $p = n + 1$.

Cette technique de *séparation fonctionnelle* n'apporte rien dans le cas très simple de la fonction factorielle mais, dans le cadre général de la définition récursive de fonctions, cette technique sera parfois très utile !

Plus généralement, l'introduction de paramètres fonctionnels et/ou de fonctionnelles auxiliaires définies récursivement est une technique importante.

Séparation fonctionnelle III Processus de calcul

```
(define fact (lambda (n) ((f (+ n 1) fact0) n)))
(define f
  (lambda (m c)
    (if (zero? m)
        c
        (f (- m 1) (lambda (n) (if (= n 0) 1 (* n (c (- n 1))))))))))

(fact 8)
((f 9 fact0) 8)
((f 8 (lambda (n) (if (= n 0) 1 (* n (fact0 (- n 1)))))) 8)
((f 8 fact1) 8)
...
((f 0 fact9) 8)
(fact9 8)
(* 8 (* ... 1))
...
(* 8 5040)
40320
```

L'expansion fonctionnelle précède le calcul arithmétique.

Séparation fonctionnelle IV

Double comptage.

```
(count '(a b a c d a c) 'a)          (3 . 4)
;; 3 occurrences de "a", 4 autres occurrences
```

Solution simple. Utiliser deux fonctions auxiliaires.

```
(define count0 (lambda (l s) (cons (c-eq l s) (c-dis l s))))
```

Problème : on parcourt la liste `l` deux fois.

Solution naïve. Utiliser le schéma habituel.

```
(define count1
  (lambda (l s)
    (if (null? l)
        (cons 0 0)
        (if (eq? (car l) s)
            (cons (1+ (car (count1 (cdr l) s))) (cdr (count1 (cdr l) s)))
            (cons (car (count1 (cdr l) s)) (1+ (cdr (count1 (cdr l) s))))))))
```

L'inefficacité est catastrophique, mais on peut y remédier simplement.

Séparation fonctionnelle V

La séparation fonctionnelle est une solution possible :

```
(define c2  ;; écrire une spécification !
  (lambda (l s c)
    (if (null? l)
        c
        (if (eq? (car l) s)
            (c2 (cdr l) s (lambda (u) (c (cons (1+ (car u)) (cdr u)))))
            (c2 (cdr l) s (lambda (u) (c (cons (car u) (1+ (cdr u))))))))))

(define id (lambda (v) v))

(define count2 (lambda (l s) ((c2 l s id) '(0 . 0))))
```

Si ℓ est la longueur de la liste `l`, les temps d'exécution de `(count0 l)` et de `(count2 l)` sont proportionnels à ℓ ; celui de `(count1 l)` est proportionnel à 2^ℓ.

(On verra d'autres solutions pour ce problème plus loin.)

Séparation fonctionnelle VI

Produit de liste : $\ell \mapsto \prod_{x \in \ell} x$.

```
(define pl1
  (lambda (l)
    (cond ((null? l) 1)
          ((zero? (car l)) 0)
          (else (* (car l) (pl1 (cdr l)))))))
```

Si un facteur est nul, comment éviter *toutes* les multiplications ?
Ici aussi, la séparation fonctionnelle permet une solution.

```
(define p2  ;; écrire une spécification !
  (lambda (l c)
    (cond ((null? l) c)
          ((zero? (car l)) (lambda (v) 0))
          (else (p2 (cdr l) (lambda (u) (* (car l) (c u))))))))

(define pl2 (lambda (l) ((p2 l id) 1)))
```

On verra une autre solution plus loin.

Séparation fonctionnelle VII

Spécifications de `c2` *et de* `p2`.

```
(define (cx p1) ;; uniquement pour faciliter la spécification
  (lambda (p2) (cons (+ (car p1) (car p2)) (+ (cdr p1) (cdr p2)))))
```

Si `[[(count2 l s)]]` = `[[(cons a b)]]`,
alors `[[(c2 l s (cx (cons u v))))]]` = `[[(cx (cons (+ u a) (+ v b))))]]`

Si `[[l]]` est une liste de nombres dont le produit vaut a,
et si `[[c]]` est la fonction $[x \mapsto bx]$,
alors `[[(p2 l c)]]` est la fonction $[x \mapsto abx]$.

7. Accumulateurs, processus itératifs

Idée de base

Un accumulateur est un argument supplémentaire,
lié à des résultats intermédiaires à mémoriser.

L'exploitation de la définition

```
(define fact (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))
```

crée le processus de gauche, alors que l'associativité de la multiplication permettrait celui de droite, plus simple :

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
(* 4 (* 3 (* 2 (* 1 1))))
(* 4 (* 3 2))
(* 4 6)
24
```

```
(fact 4)
(* 4 (fact 3))
(* 12 (fact 2))
(* 24 (fact 1))
(* 24 (fact 0))
(* 24 1)
24
```