CHAPITRE IV

Programmation Fonctionnelle

Ecole Centrale de Lyon
Département Mathématiques,
Informatiques et Systèmes
1993-94
S. SAIDI

Etude de cas :
CAML : un aperçu

- Lors du travail avec CAML, on soumet des phrases. Par exemple, pour exprimer la phrase

```
soit s la somme des nombres 1, 2 et 3
```

On écrira en CAML :

```
#let s=1+2+3;;
```

✍ Remarques syntaxiques :
- Le symbole # est le prompte (l'invite) de CAML
- Le double point-virgule (;;) désigne la fin de la phrase.∎

Définitions globales

- Reprenons la définition précédente :

```
#let s=1+2+3;;
s : int = 6
```

La deuxième ligne représente la réponse de CAML : nous avons défini *s* de type entier dont la valeur est 6.

```
#let s2 = s * s
s2 : int = 36
```

- Dans ces exemples, nous avons introduit deux liaisons globales de noms d'identificateurs (s et s2) à valeurs (6 et 36).

- De telles définitions ne sont pas modifiables et leur valeur est calculées. Il est impossible de changer la valeur d'un nom donné par *let*. On peut seulement redéfinir ce nom par une nouvelle définition par *let* :

```
#let x=1;;          l'environnement                Δ=[x -> 1]
#let x=x+1;;        le nouvel environnement =  [x -> 2] . Δ
```

- Ceci est fondamentalement différent de l'affectation. Dans x=x+1, la valeur de x figurant à droite de = est la valeur de l'ancien x, on redéfinit donc un nouveau nom (x) dont la valeur sera évaluée à 2.

Définitions locales

- Les définitions précédentes sont globales et elles existeront tant que l'on travaille sous CAML. Pour faire un petit calcul, il est inutile d'utiliser des définitions globales.
  - définition temporaires pour la seule durée de calcul en question.
  - définition locales disparaissent à la fin de l'évaluation de la phrase dans laquelle elles se trouvent.

```
# let s=3 ;;                    Δ=[s -> 3]
# let s=20 in s*4;;             Δ=[s -> 20] . Δ
- : int = 80
# s;;                           Δ=[s -> 3]
- : int = 3
```

- On remarque que "s" a conservé son ancienne valeur. Les occurrences de "s" dans "s*4" sont remplacées par la valeur de "s"; c'est à dire par "20".

✍ Remarques syntaxiques :
- La portée de **in** : jusqu'à la fin de la phrase.
- Dans la réponse de CAML, " -:" désigne un calcul et "nom :" désigne une définition de nom. ∎

- Les définitions locales sont indépendantes du type actuel des noms utilisés. Par exemple, si s et s2 sont de type entier, on peut écrire :

```
#let s=" centrale " and s2=" lyon" in "école" ^ s ^ "de" ^s2;;
s : string = "école centrale de lyon"
```

✍ Remarques syntaxiques :

- Le symbole ^ est l'opérateur de concaténation de chaînes.
- Le symbole *and* permet de faire des définitions collatérales. ∎

On considère l'exemple suivant :

```
#let x=2;;
#let x=x+1 in x+2;;
- : int = 5

#x;;
- : int = 2
```

Fonctions

- Un programme CAML est une collection de fonctions. La syntaxe des fonctions est très proche des notations mathématiques :

```
soit  succ  la fonction définie par : succ(x) = x+1
```

est traduit par :

```
#let succ(x) = x+1;;
succ : int -> int = <fun>
```

- La réponse de CAML précise que "succ" est une fonction ("fun") dont le type est "int -> int" (à lire int flèche int). Ce type est calculé par le système.

- Il est souvent utile de se rappeler le type d'une fonction :

```
# succ;;
- int -> int = <fun>
```

- La valeur d'une fonction (son corps) CAML ne peut pas être obtenue car elle est codé en machine.

✍ Remarques syntaxiques :
"<fun>" précise que CAML a donné une valeur fonctionnelle à *succ*.∎

Application de fonctions

- L'application d'une fonction se fait en invoquant le nom de la fonction suivi de ses paramètres :

```
#succ(2);;
-: int  = 3
```

- Si la fonction un seul paramètre, on peut éviter les () dans la définition et/ou dans l'application de la fonction :

```
# let succ x = x+1;;
#succ 2;;
- : int = 3
```

Définition locale de fonctions

- On peut définir une fonction localement :

```
#let pred x = x-1 in (pred 3) * (pred 4);;
-: int = 6
```

La fonction "pred" n'existe que pendant le calcul du produit "(pred 3) * (pred 4)". Remarquons que la définition locale a un argument (x). Les occurrences de "pred α" sont remplacées par "α-1" dans le produit.

- On peut également avoir une définition locale de fonction dans une définition globale.

Exemple-1 : pour définir la fonction "pred_carré" : x --> $(x-1)^2$, on définira la fonction "pred" localement :

```
#let pred_carré x =
        let pred_de_x = x-1 in pred_de_x * pred_de_x;;
pred_carré : int -> int = <fun>
```

Ici, "pred_de_x" n'a pas d'argument. Les occurrences de "pred_de_x" sont remplacées par "x-1" dans le produit "pred_de_x * pred_de_x".

```
#pred_carré 3 ;;
-: int = 4
```

- Une fonction peut à son tour définir localement une autre fonction.

Exemple-2 : définition de $x^4 = (x^2)^2$:

```
#let puiss_4 x =
        let carré y = y*y in          (*définition locale de carré *)
            carré (carré x);;
puiss_4 : int -> int = <fun>
```

Ici, les occurrences de "carré α" sont remplacées par "α*α" dans "carré (carré x)".

```
puiss_4 3 ;;
-: int = 81
```

✍ Remarques syntaxiques :

On exprime les commentaires en les entourant dans (* ... *). ∎

Fonctions à plusieurs arguments

- Définissons la fonction moyenne :

```
#let moyenne a b = (a+b)/2;;
moyenne : int -> int -> int
```

Le descripteur de type "int -> int -> int" est équivalent à "(int -> int)-> int". On déduit aisément que la fonction prend deux entiers en entrée (précisé par "(int -> int)") et produit un entier. Remarquons que pour obtenir une valeur, il faudra fournir deux paramètres à la fonction "moyenne".

```
#moyenne 9 7;;
- : int = 8
```

- Nous verrons plus loin les autres possibilités d'utilisation des fonctions à n paramètres lorsqu'on les utilise avec un nombre de paramètres inférieur à n.

Fonctions anonymes

- Une fonction CAML est une donnée comme les autres. Elle est le même statut qu'un nombre entier : elle est calculée, on peut la passer en argument ou la retourner en résultat.

- En utilisant les fonction anonymes, on peut construire des valeurs fonctionnelles sans leur donner de nom. Ses fonctions sont introduites par le mot clé "function" suivi de la formule qui les définit :

```
#(function x -> 2*x +1);;
- : int -> int = <fun>
```

CAML indique (par '-') que nous avons fait un simple calcul dont le résultat est "int -> int" et dont la valeur est une fonction (<fun>).

- On applique les fonctions anonymes comme les autres fonctions en les faisant suivre de leur arguments :

```
#(function x-> 2*x +1) (2);;
-: int = 5
```

Question de style

- Il existe un autre style de définition mathématique de fonction. Par exemple, si on se donne :

```
soit succ :    Z          ->      Z
               x          |->     x+1
```

Ce style insiste sur le fait que "succ" est une fonction qui, à tout élément x de Z, associe x+1. A l'aide des fonctions anonymes, cette fonction se traduit simplement par :

```
#let succ = function x -> x+1;;
succ : int -> int = <fun>
```

- De plus, on peut ajouter une contrainte de type sur le nom de la fonction qui rend compte de l'indication Z dans "succ : Z -> Z". Une telle **contrainte de type** est une indication explicite de type d'une expression CAML. Ceci aide à la relecture des programmes (même si CAML calcule, de sa part, le type des fonctions). Pour ce faire, on met entre () l'expression et son type séparés par ":" comme en ADA :

```
#("école" : string);;
-: string = "école"

#let (succ : int -> int) = function x -> x+1;;
succ : int -> int = <fun>
```

- Ceci est une affaire de style. Le style "let succ(x)=..." est plus concis, particulièrement si la fonction a plusieurs arguments alors que l'autre style oblige à introduire chacun des arguments par une construction "function -> ...". Par exemple

```
#let moyenne = function a -> function b -> (a+b)/2;;
moyenne : int -> int -> int = <fun>
```

- Notons que si la fonction a plusieurs arguments (comme moyenne a deux arguments), l'on pourra pas écrire "function a b -> ..." et il faut répéter le mot "function".

- Notons enfin que le mélange des deux styles est possible comme le montre l'exemple suivant qui calcule la somme de deux entiers. La fonction somme est écrite de 3 manières :

```
#let s1 a b = a+b;;
s1 : int -> int -> int = <fun>
```

```
#let s2 = fun a -> fun b -> a+b;;
s2 : int -> int -> int = <fun>
```

```
#let s3 a = fun b -> a+b;;          (* ici, on mélange les deux styles *)
s3 : int -> int -> int = <fun>
```

Utilisation :

```
#s1 1 2;;
- : int = 3
#s2 1 2;;
- : int = 3
#s3 1 2;;
- : int = 3
```

✍ Remarques syntaxiques :

Le mot "function" peut être abrégé en "fun". ∎

Exemple : encadre une chaîne par une autre :

```
#let (encadrer : string -> string -> string) = fun chaine -> fun cadre ->
      cadre ^chaine ^cadre;;
encadrer : string -> string -> string = <fun>

#encadrer "école" "ll";;
- : string = "llécolell"

#encadrer " tra la la  " "boom";;
- : string = "boom tra la la boom"
```

Le test et l'alternative

- Définissons en CAML la fonction val_abs dont la définition mathématique est :

$$|x| = \begin{cases} x & \text{si } x \geq 0 \\ -x & \text{sinon} \end{cases}$$

- Le schéma général de test en CAML est :
  if expression_bool **then** expression **else** expression

- Les expressions peuvent utiliser les comparateurs (**<, >, =, >=, <=**)

```
#let val_abs (x) = if x > 0 then x else -x;;
val_abs : int -> int = <fun>

#val_abs (3);;
-: int = 3

#val_abs -4;;        (* les 2 formes utilisables si un seul argument *)
-: int = 4
```

Les valeurs de vérité sont **true** et **false** du type **bool**.

```
#2 < 1;;
-:bool = false

#(val_abs 3) = (val_abs (-3));;
:-bool = true;
```

Simplification de programmes : conventions

- Les programmes CAML sont simplement des collections de fonctions. Habituellement, une de ces fonctions joue un rôle principal et utilise les autres. Les règles suivantes décrivent les simplifications utilisées dans les programmes.

Dans les définitions de fonctions :

```
     let f = function x -> ....                ≡    let f x = ....
et   f = function x ->function y -> ....       ≡    let f x y = ....
```

Dans les applications de fonctions :

```
     f  (x)                                    ≡    f x = ....
```

Cette convention est valable que si x est une variable ou une constante. Si on a des expressions complexes, on les met entre () :

```
#succ (2*3);;  ==> 7
#succ 2 *3;;   ==> 9          c'est à dire : (succ 2) * 3
```

- De même :
  moyenne (2) (6) ≡ moyenne 2 6 et l'on pourra par exemple écrire
  *#moyenne (2*3) (3+3)*

Une expression associe à gauche en CAML. C'est à dire, on peut enlever les () dans "(f x) y" qui applique le résultat de "f x" à "y". Cependant, elles sont indispensables dans "f (g x)" qui applique "f" au résultat de "g x". On peut donc écrire "(moyenne 2) 6" au lieu de "moyenne 2 6" ≡ "moyenne (2) (6)" ≡ "(moyenne 2 6)" ...

En revanche, on ne peut absolument pas regrouper les arguments 2 et 6 dans une () et écrire "moyenne (2 6)" qui provoquera une erreur. Cette écriture voudrait dire que l'on applique moyenne à un seul argument : (2 6). Qui plus est, cela voudrait dire que l'on tente d'appliquer 2 à 6 !!
Pourtant, des expressions du genre "moyenne (2 6)" ou plus généralement "f (g x)" ont un sens. Par exemple, dans "#succ (succ 1)"; on applique "succ" à "(succ 1)" et on obtient l'entier 3. On peut également écrire "#succ succ 1"; ce qui signifie qu'on applique succ à deux arguments : le premier est succ et le second est 1. C'est en effet la même écriture que "moyenne 2 6". Cependant, "succ" n'admet qu'un seul argument. Donc, si l'on écrit "succ succ 1", CAML indique une erreur en disant que "succ" (le premier argument) est du type "int -> int" (à un seul argument entier) que l'on essaie d'utiliser avec "int -> 'a -> 'b (c'est à dire, ne peut être utilisé avec un type différent).
Pour résumer :

```
f  (g x)          n'est  pas équivalent à          f g x = ....
```

- Également, il est inutile de définir une fonction qui se contente d'en appeler immédiatement une autre. Par exemple :

```
#let succ1 = function x -> succ (x);;
succ1 : int -> int = <fun>
```

"succ1" et "succ" rendent toujours les mêmes valeurs. On dit que "succ1" est *égale* à "succ". On aurait pu définir :

```
#let succ1 = succ;;
```

On en déduit la règle suivante :

```
function x -> f x ≡f
```

C'est à dire, la définition de "f" décrit :

f : x -> f(x) ou encore "f = function x -> f(x)".

Cette règle est appelée la règle **d'extensionnalité** ou la règle êta (η). L'action de remplacement de "function x -> f(x)" par "f" est appelée une **η-contraction**.

Quand on remplace "f" par "function x -> f(x)", on dit qu'on a fait une **η-expansion**. Résumons ces règles par :

```
let g x = f x    ≡    let g = f.
```

La programmation Fonctionnelle

- Un programme : une fonction des données en entrée vers les données en sortie (résultats) :

  données en entrées -- programme ---> données en sortie

- Le comportement impératif équivalent de ce programme est mise en oeuvre d'une manière indirecte par des commandes de lecture et de manipulation de données (en entrée) et l'émission des valeurs de résultats.

- Ces commandes interagissent et s'influencent à travers les variables stockées en mémoire. On peut dire que les liens entre deux commandes serait totalement prévisible et compréhensible si l'on se réfère à (on tient compte de) toutes les variables (en évolution constante par l'affectation) qu'elles manipulent ainsi qu'aux autres commandes interférant sur les mêmes variables. On comprend alors pourquoi il est difficile d'établir les relations entre les commandes.

- En programmation **fonctionnelle**, la fonction entre les entrées et les sorties est réalisée plus directement. Un programme est une **fonction** (ou un groupe de fonctions) composées de fonctions plus simples.
  Les relation entre les fonctions est plus simple : une fonction peut en appeler une autre ou, le résultat d'une fonction peut être utilisé par une autre fonction. Il n'y a plus de variable ("stockage partagé" au sens impératif), de commande ou d'autres effets de bord. Tout est expression (fonction) et déclaration. En programmation fonctionnelle, l'absence de stockage est compensée par l'utilisation des fonction d'ordre supérieur et par l'évaluation retardée.

- La programmation fonctionnelle est caractérisée par l'utilisation de fonctions et d'expressions alors que la programmation impérative est caractérisée par l'utilisation de variables, de commandes (nottament l'affectation) et de procédures. La programmation impérative manipule des

fonctions et des expressions, mais souvent sous une forme primaire et pauvre en pouvoir d'expression.

- Un langage impératif tel que Pascal est très pauvre en fonction et ne respecte pas le principe de complétude de type : **Aucune opération ne doit être arbitrairement restreinte aux types des valeurs impliquées.**

  Autrement dit, on doit pouvoir écrire des fonctions qui rendent non seulement des types de bases (entier, réel, ...) mais n'importe quel type et nottament des fonctions.

- Par exemple, pour définir une fonction d'addition sur des nombres complexes, on écrira en Pascal

```
type complexe = record re,im : real end;
var i,w,y,z : complexe;
procedure add(c1,c2 : complexe; var somme : complexe);
begin
      somme.re := c1.re+c2+re;
      somme.im := c1.im+c2.im;
end;
```

Utilisation (additionner w,y et i : deux appels)

```
i.re := 0.0; i.im := 0.0;
add(w, y, z); ...
add(z,i , z);
```

- Notons que l'implantation "naturelle" et intuitive de l'addition est une fonction.

- On constate que Pascal ne permet pas des constantes record, des agrégats record et des fonctions qui retournent des record (voir exemple ci-dessous). Nous somme donc obligés d'abuser des procédures et de l'affectation pour compenser ces lacunes.

- D'autre part, l'implantation des TDA, formalisme hautement récursif puissant de spécification de type est rendue difficile à cause de l'écart entre le formalisme et l'implantation des spécifications.

- Parmi les langages impératifs, ADA est mieux armé et pallie certains de ces problèmes. La version ADA de la fonction ci-dessus serait :

```
type complexe is record re,im : real end record;
i : constant complexe := (re => 0.0, im => 1.0);
w,y,z : complexe;
function "+"(c1,c2 : complexe) return complexe is
begin
      return (   re => c1.re+c2+re ,
                 im => c1.im+c2.im);
end;
```

Utilisation (un seul appel):

```
z := w + y + i;
```

- Plus le langage utilisé est riche en expression, moins nous aurons besoin de faire appel à des variables et aux commandes.

- Un langage fonctionnel pur exclue les commandes et les variables.

- Pour quoi être si dogmatique pour exclure ces commandes? Pourquoi ne pas utiliser les expressions dans les langages fonctionnels avec la partie de commandes d'un bon langage impératif pour concevoir un langage meilleur?
  Malheureusement, quelques aspects puissants de la programmation fonctionnelle, notamment l'évaluation paresseuse se conjugue mal avec les effets de bord et leur combinaison dans un seul langage peut conduire à une complexité importante et à une programmation qui ne sera pas à l'abri d'erreur ou de comportement imprévisible.

- Dans la pratique, les langages fonctionnels adoptent un degré limité des aspects de la programmation fonctionnelle et retiennent les variables (cf. ML, CAML) ou rejette les variables totalement (cf. Miranda).

Le paradigme fonctionnel

- Pour illustrer le paradigme fonctionnel, prenons l'exemple de la fonction factorielle donné dans sa version ADA et ML :

Version itérative ADA:

```
function factorielle (n : integer) return integer is
f : integer := 1;
begin
      while n > 0 loop
            f := f * n;
            n := n-1;
      end loop;
      return f;
end factorielle;
```

- On peut proposer une version récursive ADA.

- L'équivalent fonctionnel de l'itération est la récursivité. Dans l'exemple ci-dessus, les variables locales qui prennent différentes valeurs à chaque itération sont remplacées naturellement par des paramètres qui prennent différentes valeurs à chaque appel.

Version CAML :

```
#let rec fact n f = if n>0 then (fact (n-1) f*n) else 1;;
fact : int -> 'a -> int = <fun>
```

et la fonction factorielle sera de la forme :

```
#let factorielle n = fact n 1;;
factorielle : int -> int = <fun>
#factorielle 5;;
- : int = 120
```

- La traduction des schémas n'est pas toujours aussi rapide. Par exemple, les variables globales ne peuvent pas être remplacées par des variables locales.

Il n'y a pas non plus d'élément permettant d'affirmer que la fonction obtenue est "bonne" ou "meilleure" que le programme ADA de départ. En effet, écrire la fonction directement en CAML est la meilleure solution.

```
#let rec factorielle n = if n>0 then n * fact(n-1) else 1;;
factorielle : int -> int = <fun>
```

- Cet exemple montre que programmer sans les variables locales et les boucles ne pose pas de problèmes fondamentaux en programmation fonctionnelle.

- Examinons quelques aspects importants et pratiques du paradigme fonctionnel présent dans les langages fonctionnels.

Pattern-Maching (filtrage)

- Le filtrage est un concept commun dans les langages fonctionnels modernes. A première vue, le filtrage a un comportement proche de "case" de ML ou la construction "match expr with" de CAML. Cependant, le filtrage présente une meilleure puissance d'expression : il peut être utilisé dans la définition d'une fonction.

- Le filtrage supprime le besoin d'utiliser les fonctions destructrices de Lisp telles que "car" et "cdr".

Exemple : soit la définition suivante :

```
#type forme = point | cercle of float | rectangle of (float * float);;

#let surface = function
        point -> 0.00
    |   cercle r -> 3.14 *. r *. r
    |   rectangle (w,h) -> w*.h;;

surface : forme -> float = <fun>
```

Dans cet exemple, *forme* est une union disjointe de unit+réel+(réel × réel) avec les étiquettes *point*, *cercle* et *rectangle*. La fonction surface est de type "forme -> réel", définie par trois équations. La première équation s'applique si l'argument est "point()" et filtre le terme (pattern) "point"; dans ce cas, la fonction retourne 0.0. La seconde équation est applicable lorsque l'argument est de la forme "cercle 0.5" qui correspond au terme "cercle r". Dans ce cas, l'argument "r" est lié à 0.5 et la partie à droite de "->" est évaluée. Le troisième équation est appliquée lorsque l'argument est une valeur telle que "rectangle(2.0, 3.0)" qui correspond à "rectangle (w,h)" auquel cas les identificateurs "w" et "h" sont liés à 2.0 et à 3.0 et l'expression "w*h" est évaluée.

Exemples d'utilisation

```
#surface (cercle 1.2);;
- : float = 4.5216

#surface point
- : float = 0

#surface (boite (1.2, 1.5));;
- : float = 1.8
```

- Une fonction peut en général être évaluée par différentes équations chacune avec une partie gauche différente. Afin d'être appliqué, les filtres figurant ainsi à gauche contiennent l'argument de la fonction. Un pattern filtre un argument si l'on peut donner des valeurs à ses identificateur de telle sorte que le pattern et l'argument soient identiques. Dans ce cas, les identificateurs seront liés aux valeurs et l'expression à droite de '->' est évaluée.

- Définir une fonction par des équations est très pratique, concise et rappelle des notations mathématiques familières. De plus, les équations peuvent être comprises d'une manière indépendante comme des phrases vraies à propos de la fonction ainsi définie; ce qui facilite le raisonnement sur la fonction.

Valeurs et types

- Comme les autres langages de programmation, les langages fonctionnels proposent des types similaires tels que les valeurs de vérité, entiers, réels, chaînes, produit cartésien (record), union disjointe (type somme : record à champs variants) ...
  Cependant, comme l'affectation et la mise à jour sélective sont interdites, on doit manipuler les valeurs composées d'une manière différente. Là où le programmeur dans un langage impératif mettra à jour une composante d'une valeur composée, un programmeur fonctionnel fera une copie de cette valeur composée avec des valeurs différentes.

Exemple : insertion dans un ABOH d'entiers :

```
type arbre = nul | noeud of (arbre * int * arbre);;

let rec insere = function
        (new , nul) -> noeud (nul, new, nul)
    |   (new , noeud (gauche , racine , droit)) ->
        if new <= racine then
            noeud(insere (new, gauche), racine, droit)
        else noeud (gauche, racine, insere(new, droit));;
```

Exemples d'utilisation :

```
#insere (1, nul);;
- : arbre = noeud (nul, 1, nul)
#insere(1, noeud(nul, 2, nul));;
- : arbre = noeud (noeud (nul, 1, nul), 2, nul)
```

- A première vue, cette fonction copie entièrement un arbre pour effectuer une petite modification; ce qui semble très coûteux. Mais une étude plus en détail de la fonction "insere" permet de constater que seuls les noeuds se trouvant sur le chemin qui va de la racine jusqu'au point d'insertion sont copiés; les anciens noeuds sont "partagés" par les deux arbre (précédent et le nouveau). Ainsi, le coût de l'insertion dépend de la profondeur du niveau d'insertion.

- Les langages fonctionnels utilisent beaucoup les pointeurs (comme dans les langages impératifs). Cependant, une différence majeur est que le programmeur n'a pas accès à ces pointeurs. La gestion (comme la création de "noeud(...)" par un allocateur de cellules dans le TAS) est laissée aux techniques avancées de gestion de mémoire.
  Sachant que l'espace mémoire nécessaire est important, on peut envisager de proposer une desallocateur (comme le "dispose" de Pascal), mais ceci donne à l'utilisateur la liberté de laisser des références non cohérentes ou du moins non efficacement récupérés et les langages fonctionnels proposent pas de tel outils. En revanche, la récupération se fait d'une manière automatique lorsqu'un objet devient inaccessible. Ainsi, le programmeur est libéré de la tâche de gestion de mémoire. On imagine mal comment un programmeur peut mettre en place le partage des données (comme dans l'exemple d'arbre précédent) et de penser d'une manière abstraite à la solution à son problème. Il sera par exemple obligé de considérer l'arbre précédent comme un "graphe" et abandonner l'abstraction "arbre" qui lui facilité sa description. D'autant plus que savoir ou pas si des noeuds de l'arbre sont partagés n'a pas d'effet sur le comportement de son programme fonctionnel.

- L'exemple précédent montre que les structures récursives telles que les listes et les arbres peuvent être manipulées avec une efficacité raisonnable dans les langages fonctionnels. Ceci n'est pas le cas pour les tableaux (au sens suite de cases en mémoire). Deux arbres qui diffèrent en un seul composant peuvent partager une grande partie de leur structure mais deux tableaux qui diffèrent en un seul composant ne peuvent pas en faire autant. De ce fait, une fonction qui modifie un tableau doit renvoyer (en le copiant) la totalité du tableau. Ce coût est prohibitif si la fonction modifie un seul composant du tableau. C'est pourquoi peu de langages fonctionnels proposent les tableaux et ceux qui offrent une telle structure de données proposent des opérateurs capables de modifier une large partie du tableau (en incitant le programmeur à les utiliser); d'autres traitent les tableaux comme des structures mutables avec des opérateurs d'accès et de modification comme en Pascal ou en ADA (cf. CAML).

- En revanche, les **listes** comme donnée composite sont proposées par tous les langages fonctionnels. Par exemple, le type liste est définie en CAML par :

```
type 'a list = nil | cons of ('a * 'a list)
```

prédéfini en CAML avec la constante [] (pour nil) et l'opérateur infixé "::" par:

```
type 'a list = [] | prefixe :: of ('a * 'a list)
```

Il existe des notations pour la construction de listes :

[] = la liste vide
x :: xs = cons (x, xs) séparation de la tête et de la queue
[a; b; ... ; z] = a:: b :: ... :: z :: nil

- Une collection de fonctions prédéfinies rendent l'utilisation des listes plus simple :

hd	:	'a list	-> 'a	(la tête)
tl	:	'a liste	-> 'a liste	(la queue)
length	:	'a list	-> integer	(la longueur)
@	:	'a list × 'a liste	-> 'a list	(concaténation)

- Exemple : définition des fonction somme et produit qui retournent la somme et le produit des éléments d'une liste d'entiers :

```
#let rec sum = fun
        [] -> 0
    |   (n :: ns) -> n + sum ns;;

sum : int list -> int = <fun>

#let rec prod = fun
        [] -> 1
    |   (n :: ns) -> n * prod ns;;
```

```
prod : int list -> int = <fun>

#sum [1;2;3];;
- : int = 6
#prod [1;2;3];;
- : int = 6
#prod (1::2::4::[]);;
- : int = 8
```

- La définition suivante permet de produire une liste d'entiers *de x jusqu'à y* :

```
##infix "jusqua";;                (* directive précédée de # *)

#let rec prefix jusqua = fun x y -> if x > y then [] else x :: ((x+1) jusqua
y);;
jusqua : int -> int -> int list = <fun>

#4 jusqua 10;;
- : int list = [4; 5; 6; 7; 8; 9; 10]
```

- On peut alors utiliser cette fonction pour définir la fonction factorielle d'une autre manière :

```
#let factorielle n = produit (1 jusqu'à n);;
factorielle : int -> int = <fun>
#factorielle 5;;
- : int = 120
```

- Comme nous l'avons remarqué, l'itération est souvent exprimée par l'application de fonctions aux listes. Cette itération sur les listes étant importante, certains langages fonctionnels tels que Miranda (ou Gofer, mais pas CAML) proposent une notation spéciale, appelée "list comprehension", pour spécifier les listes comme pour les ensembles au sens mathématique. Par exemple, si "ints" est une liste d'entiers, on peut en extraire une liste des nombres paires en écrivant :

[n | n <- ints ; n mod 2 = 0]

qui est à comparer avec la notation mathématique ensembliste appliquée aux listes :

{n | n ∈ ints; n mod 2 = 0}

L'occurrence de "n" dans "n <- ..." est une assignation dont la portée est le "list comprehension".

- Par exemple, pour construire une liste d'entiers chacun le successeur des éléments de la liste "ints", on écrit :

[n+1 | n <- ints] (i.e. {n+1 | n∈ ints})

- Pour écrire une fonction qui calcule la somme des carrés des n premiers entiers (en syntaxe CAML) :

```
#let somme_sq(n) = somme [i * i | i <- 1 jusqua n]
```

- Et pour écrire la fonction quick-sort :

```
#let sort = fun
      []    ->   []
|     (n::ns) ->
      sort [i | i <- ns; i < n] @
      [n] @
      sort [i | i <- ns; i >= n]
```

- Cette notation abstraite peut être traduite en CAML en utilisant les fonctions d'ordre supérieur.

Les fonctions d'ordre supérieur

- Les langages fonctionnels traitent les fonctions comme des objets de **première classe** : elles peuvent être passés comme argument, être retournées comme résultat, faire partie d'une valeur composite, etc. Ceci a un impact important sur le style de programmation.

- On appelle *fonction de premier ordre* une fonction dont les paramètres et les résultats sont tous non fonctionnels.
  Une fonction qui a des paramètres le résultat fonctionnels est appelée une *fonction d'ordre supérieur*.

  Les fonctions d'ordre supérieur sont souvent utilisées pour produire du code réutilisable en faisant abstraction sur les parties spécifiques à une application particulière. De cette manière, les fonction sont utilisées pour représenter les données d'une façon nouvelle.

- Permettre à une fonction de retourner une autre fonction comme résultat ouvre des possibilités intéressantes qui sont largement utilisées en programmation fonctionnelle.

- La plus simple manière d'engendrer une nouvelle fonction est de *composer* deux fonctions déjà existantes.

  On se donne l'opérateur binaire "°" (noté *compose* ci-dessous) qui prend en arguments deux fonctions f et g et retourne leur composition : c'est à dire une fonction h telle que h(x) = f(g(x)).

Exemple -1 :

```
#let compose (f : 'b -> 'c) (g : 'a -> 'b) = fun (x : 'a) -> f(g(x));;
compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

#let even = fun x -> x mod 2 =0;;
even : int -> bool = <fun>

#let odd= compose (prefix not) even;;
even : int -> bool = <fun>
```

Exemples d'utilisation :

```
#odd 1;;
- : bool = true
```

```
#odd 2;;
- : bool = false
```

Exemple -2 :

```
#let double (f : 'a -> 'a) = compose f f;;
double : ('a -> 'a) -> 'a -> 'a = <fun>

#let inc x = x+1;;
inc : int -> int = <fun>
```

Exemples d'utilisation :

```
#double inc;;
- : int -> int = <fun>

#(double inc) 1;;
- : int = 3
```

Curryfication

Considérons la fonction suivante qui calcule le x^n ($n \geq 0$, x : réel).

```
#let rec pow (n, x) = if n = 0 then 1.0 else x *. pow((n-1),x);;
pow : int * float -> float = <fun>
```

- L'application de cette fonction à un couple (entier,réel) retourne un réel. Par exemple pow(2,a) retourne a^2. Considérons maintenant la fonction powC proche de pow :

```
#let rec powC n x = if n = 0 then 1.0 else x *. powC (n-1) x;;
powC : int -> float -> float = <fun>
```

- Le type de powC est *int -> float -> float = int -> (float -> float)*, l'opérateur '->' étant associatif à droite. L'application de powC à un couple

entier-réel retourne un réel. Cependant, powC est plus intéressante que pow car elle peut être utilisée avec un seul argument :

```
#let sqr = powC 2 and cube = powC 3;;    (* déclaration collatérale *)
sqr : float -> float = <fun>
cube : float -> float = <fun>
```

- On notera que *pow* prend en fait un argument qui est un couple alors que *powC* prend un couple d'argument (deux arguments).

  La version powC de pow est appelée une version **curryfiée** (d'après le logicien Haskel Curry). La technique d'appeler une fonction curryfiée avec un nombre d'arguments inférieur au maximum de ses arguments est appelée une *application partielle* de *powC*.

  La fonction (prédéfinie en ML) *filtre : ('a -> bool) -> ('b list -> 'b list)* applique une fonction f aux éléments x d'une liste L et retourne une liste d'éléments de L pour lesquels (f x) est vrai.

```
#let rec filtre (f : 'a -> bool) = fun
            []          -> []
      | (x :: xs)       -> if (f x) then (x:: filtre f xs) else filtre f xs;;

filtre : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Etant donné la fonction

```
#odd;;
- : int -> bool = <fun>
```

on peut filtrer les éléments d'une liste d'entiers et ne conserver que ceux qui sont impaires :

```
#filtre odd [1;2;3;4];;
- : int list = [1; 3]
```

Notons que la fonction *filtre* nous permet de traduire le schéma ensembliste :

{n | n ∈ ints; n mod 2 = 0}.

Le langage CAML dispose d'un certains nombre de fonction prédéfinies d'ordre supérieur. Par exemple, les fonctions suivantes pour la manipulation de listes (il en existe un nombre important) :

for_all : ('a -> bool) -> 'a list -> bool où
for_all f [a1;...; an] = (f a1) & ... & f(an) & : et logique

map ('a -> 'b) -> 'a list -> 'b list où
map f [a1;...; an] = [(f a1) ; ... ; f(an)]

flat_map ('a -> 'b list) -> 'a list -> 'b list où
flat_map f [l1;...; ln] = (f l1) @ ... @ f(ln) @ : concaténation

Exemple : quick-sort générique :

Le but de cet exemple est de produire une fonction quick-sort générique où la relation d'ordre sur les éléments de la liste à ordonner est passée en paramètre.

Version CAML :

Pour cela, nous allons nous définir une fonction d'ordre supérieur *filtre2 f x l* qui prend en entrée une fonction booléenne binaire f, un paramètre x et une liste l=[a1;...; an] et calcule la liste [(f x a1); ... ; (f x an)].

```
#let rec filtre2 (f : 'a -> 'a -> bool) (x : 'a) = fun
      []            -> []
      | (y :: ys)       -> if (f x y) then (y:: filtre2 f x ys) else (filtre2 f x ys);;

filtre2 : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list = <fun>
```

Par exemple :

```
#filtre2 (prefix <) 3 [1;2;3];; (* les éléments plus_grand que 3 *)
- : int list = []

#filtre2 (prefix <) 2 [1;2;3];; (* les éléments plus_grand que 2 *)
- : int list = [3]
```

On se donne également la fonction de récupération de l'inverse d'un ordre :

```
#let inverse = fun
      (prefix <) -> (prefix >=)
|     (prefix >) -> (prefix <=)
|     ...              -> .. ;;
```

La fonction générique quick_sort sera :

```
let rec gen_q_sort ordre = fun
      []          -> []
      | (x :: xs) -> gen_q_sort ordre (filtre2 ordre x xs)
                     @ [x]
                     @ gen_q_sort ordre (filtre2 (inverse ordre) x xs);;
```

Remarques : on ne peut pas remplacer inverse par not car par exemple not (x= y) est différent de ((not =) x y).

Exemple d'utilisation :

```
#gen_q_sort (prefix <) [1;4;5;3];;
-: int list =[5;4;3;1]
```

```
#gen_q_sort (prefix >) [1;4;5;3];;
-: int list =[1;3;4;5]
```

- On peut également compliquer le paramètre "ordre" passé à "quick-sort" générique. Par exemple, on peut trier une liste de tuples (jour, mois, an) chacun représentant une date en définissant une fonction *plus_petit* sur les dates; puis en triant une liste de dates.

Autre solution à la fonction q_sort :

```
#let rec gen_q_sort ordre = fun
            []          -> []
            l (x :: xs)  -> let l = (filtre2 ordre x xs) in
                                (gen_q_sort ordre l)
                                @ [x]
                                @ gen_q_sort ordre (subtract xs l);;
gen_q_sort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

La fonction *subtract l1 l2 : 'a liste -> 'a list -> 'a list* retourne la liste l1 privée de ses éléments figurant dans la liste l2.

Fonctions d'ordre supérieur : accumulation sur les listes

Exemple

Les fonctions *somme* et *produit* que nous avions définies ont une structure similaire. Elles diffèrent seulement en l'opérateur appliqué aux éléments d'une liste non-vide ainsi qu'en l'élément neutre de cet opérateur. On peut donc définir une fonction *list_it* qui généralise ce types de fonctions.

Ayant :

```
#let rec somme_1 = function
      []     ->   0
      l tete::reste -> tete+ (somme_1 reste);;
somme_1 : int list -> int = <fun>
```

```
#let rec prod_1 = function
      []     ->   1.0
      l tete::reste -> tete *.  (prod_1 reste);;
prod_1 : float list -> int = <fun>
```

```
#let rec concat_1 = function
      []     ->   ""
      l tete::reste -> tete ^  (concat_1 reste);;
concat_1 : string list -> int = <fun>
```

- Dans ces différents cas et d'une manière générale, on applique une fonction binaire f à une liste [a1; a2;...; ap]. Le développement de la récursivité fait apparaître la forme
  f a1 (f a2 (..(f a_p neutre) ..)
  où neutre est l'élément neutre de la fonction f : (f x neutre) = x.

Ici, les opérandes (paramètres) de f sont a_i et $f(a_{i+1}...)$

On généralise avec

```
#let rec list_it = function f -> function neutre -> function
      []     -> neutre
      | tete :: reste -> f  tete (list_it f neutre reste);;
list_it : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

```
#list_it (prefix ^) "" ["aa"; "bb"];;
- : string = "aabb"
```

- Remarque : également, la function d'ordre supérieur *it_list* réalise la forme
  f(..(f(f neutre a1) a2) .. ap).

Ici, les opérandes (paramètres) de f sont $f(a_{i+1}...)$ et a_i.

La différence réside dans la position du premier paramètre de f :
x1+x2+...+xp =
x1 + (x2+...+xp) = -- list_it
(x1+x2+...)+xp -- it_list

- Ces exemples montrent que très souvent, lorsque l'on fait abstraction sur la fonction, on fait également souvent abstraction sur les types. On constate qu'il y a peu d'intérêt à réutiliser une fonction pour ordonner une liste d'entiers selon différents ordre. En revanche, il serait plus intéressant de pouvoir ordonner des listes de types différents. Ainsi, la plupart des langages fonctionnels possèdent un système polymorphe ou bien il sont dynamiquement typés.

Evaluation paresseuse

- Utilisée dans certains langages fonctionnels.
- Permet de retarder l'évaluation des arguments des fonctions jusqu'à leur première utilisation au lieu de les évaluer au moment où la fonction est appelée. Si un argument n'est jamais utilisé, il ne sera jamais évalué; ce qui permet un gain en temps et permet de passer des arguments partiellement évalués aux fonctions et de renvoyer des résultats partiellement évalués.

- Une conséquence immédiate remarquable de ceci est que l'on peut construire des structures de données **infinies**. Par exemple, on pourra avoir des listes infinies dont la queue reste non évaluée (listes paresseuses).

- En considérant le dialecte paresseuse de ML, on peut écrire une fonction qui permet de manipuler des nombres >= n :

```
#let rec from n = n :: from(n+1);;
```

qui engendre une liste infinie de nombres. Avec l'évaluation classique; cette fonction ne terminera jamais; avec l'évaluation paresseuses, la récursivité est intégrée dans la liste elle-même et sera évaluée quand la queue de la liste est invoquée lors des calculs.

- La fonction suivante permet de calculer les premiers nombres premiers :

```
# let  rec firstprime (n::ns) = if premier(n) then n else firstprime ns;;
```

En composant ces deux dernières fonctions, on aura les premiers nombres premiers >=n :

```
#firstprime (from n);;
```

- En principe, cette expression calcule d'abord une liste infinie puis teste les premiers éléments de cette liste à la recherche d'un premier nombre premier. En pratique, cette liste reste partiellement évaluée et le reste n'est évaluée que si *firstprime* sélectionne le reste de la liste.

- Un intérêt majeur de l'évaluation partielle est la séparation du contrôle et du calcul. L'idée est de diviser un calcul (itératif ou récursif) en une partie calcul pur qui produit les valeurs nécessaires les rangent dans une structure (comme dans une liste paresseuse) et une autre partie qui traverse cette structure.
  Cet exemple est à rapprocher à la combinaison enum+contrainte de PrologIII (génération indépendante du contrôle par des contraintes rejetant certaines valeurs).
  Le mécanisme de *freeze* est un autre exemple de cette méthode où le contrôle n'est appliqué que si la valeur est présente (instanciée).

  On peut également envisager un espace de recherche (un arbre infini engendré comme une structure paresseuse) indépendant d'une stratégie de recherche appliquée à cet espace. La génération de l'espace pourra être assujettie à un contrôle de niveau.

  Un inconvénient de l'évaluation paresseuse est que le programmeur a peu de contrôle sur l'ordre de l'évaluation car les valeurs sont produites que si elles sont sollicitées.

Transformation et preuve

- Une des motivation pour étudier les langages fonctionnels est la possibilité de transformer les programmes fonctionnels et d'apporter leur preuve formelle. Apporter la preuve des programmes fonctionnels est facilitée par l'application de la règle simple : les identificateurs peuvent être remplacées librement par leur valeur.

  L'apport de preuve dans les langages impératifs est beaucoup plus difficile.

- Considérons la définition de l'algorithme pgcd d'Euclid :
  pgcd(0,n) = n
  pgcd(m,n) = pgcd(n mod m) pour m > 0

Une définition Pascal :

```
function pgcd(m,n : integer) : integer ;
```

```
        var tmp : integer;
begin
        while m <> 0 do
                begin tmp := m; m := n mod m; n:=tmp end;
        pgcd := n;
end;
```

Cette définition est peu claire par rapport de la spécification de l'algorithme d'Euclid. De plus, on peut très difficilement apporter la preuve et il faudra avoir recours à la logique de Floyd-Hoare ; une entreprise ô combien difficile.

- La version fonctionnelle de l'algorithme d'Euclid :

```
#let rec pgcd m n = if m=0 then n else (pgcd (n mod m) m);;
pgcd : int -> int -> int = <fun>
```

- Les procédures récursives des langages tels que Pascal ont également un passé très lourde en complexité et en erreurs.
  On pourra donc en toute rigueur avoir recours aux fonctions. Il y a également des problèmes sur les paramètres et il faudra se restreindre à ne manipuler que des paramètres en entrée. De plus, l'utilisation des variables globales à l'intérieur de ces fonction rend la preuve difficile.

- Les langages fonctionnels ont la réputation d'être lents par rapport aux langages impératifs. Les causes de ces inefficacités sont :

  • les fonction étant des objets de première classe, la plupart des allocations doivent se faire dynamiquement sur le TAS et libérer le programmeur de cette tâche en évitant des pointeurs "baladeuses" (dangling references).
  • l'évaluation paresseuse nécessite beaucoup de tests pour décider de l'évaluation (ce qui n'est pas le cas dans les langages impératifs).
  • Le style de la programmation fonctionnelle utilise beaucoup de structures intermédiaires (TDA implanté par des fonctions, absence des mutables).

Polymorphisme et inférence de types

- Le polymorphisme des langages fonctionnels tels que ML et CAML évitent les erreurs d'exécution tout en conservant la a flexibilité des langages sans types.
  Pour toute fonction, on infère un type qui est le plus général possible.

Calcul de type

- Une constante (0,1,2..., true,false) dont le type est simple (int ou bool);

- Un identificateur (x, y, z, succ, eq, gt, positive...) dont le type est définie dans un environnement *ENV* par un type simple, une variable de type ou par*(α -> β)* où α et β sont des types. L'environnement est structuré par le règle BNF : *ENV ::= void ; ID, TYPE, ENV*.

- Une conditiannelle de la forme *if e1 then e2 else e3* dont le type est *T* si le type de *e1* est *bool* et *T* est le type de *e2* et de *e3*.

- Une lambda abstraction of the form *(function x -> e)* dont le type est *(T1 -> T2)* où *T1* est le type de *x* et *T2* est le type de l'expression *e*;

- Une application de la forme *(e e')* dont le type de la partie function *e* est *(T1 -> T2); T1* est le type associé à *e'* et *T2* est le type de l'application.

  On note que l'application partielle est autorisée car le constructeur de type "->" est associatif à droite.

Exemples:

-Le type de la lambda abstraction *lambda x . (x > 0)* sera *(int-> bool)* si le type de ">" est (int -> (int -> bool)). Par conséquent, le type de *x* est *int*.

-Le type de l'application *(positive x)* est *bool* où *x* est de type *int* et *positive* a pour type *(int-> bool)*.

-Le type de l'application *(function x -> (eq x x))(succ y)* est *bool* où *y* est de type *int*, *succ* a pour type *(int-> int)* et *eq* a pour type *(α -> (α -> bool))*.

-Le type de l'expression

(function x -> if (positive x) then (succ y) else (pred y)) est *int* où *y* est de type *int*, *succ* et *pred* ont le type *(int-> int)* et *positive* est de type *(int-> bool)*.

Comment définir le type d'une expression

Exemple :

```
#let comp = function f -> function g -> function x -> f(g x);;
comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Soit les types suivants :

f : σ1
g : σ2
x : σ3
(g x) : σ4
f(g x) : σ5

Soit le type de *(g x)* = σ4

g : σ2 = σ3 -> σ4
↓
x

Soit le type de *f(g x)* = σ5

f : σ1 = σ4 -> σ5
↓
(g x)

Remarque : si g : A -> B et f : B -> C alors f(g) : A -> C
Donc :

f : σ1 = σ4 -> σ5
g : σ2 = σ3 -> σ4
x : σ3
f(g) : σ3 -> σ5 par la remarque ci-dessus

Posons :

σ3 = ‘c, σ4 = ‘a et σ5 = ‘b

on a :

σ1 = ‘a -> ’b

σ2 = ‘c -> ‘a

Donc comp : (‘a -> ‘b) -> (‘c -> ‘a) -> ‘c -> ‘b

↓ ↓ ↓ ↓

f g x f(g x)

Application :

quel est le type de l’application partielle *(comp succ)* étant donné

succ : int -> int ?

On a vu :

comp : (‘a -> ‘b) -> (‘c -> ‘a) -> ‘c -> ‘b

↓

succ : int -> int car succ est le 1er paramètre ==> ‘a = int, ‘b = int

(comp succ) aura pour type le reste, i.e. (‘c -> ‘a) -> ‘c -> ‘b

i.e. (‘c -> int) -> ‘c -> int

ou encore d’une manière générale

(comp succ) : (‘a -> int) -> (‘a -> int)

Par exemple, on a

((comp succ) succ) : int -> int

et

```
#((comp succ) succ) 1;;
- : int = 3
```

Résumé des caractéristiques des langages fonctionnels

- Tout est fonction et application de fonction. Les fonctions doivent être libérées de restrictions arbitraires. Elle doivent prendre tout type d'argument et renvoyer tout type (y compris des fonctions).

- La récursivité.

Les fonctions prennent leur valeurs de l'extérieur (lors de l'appel) ou par des définitions. Ces valeurs ne peuvent pas être modifiées mais les appels récursives peuvent produire une série d'arguments changeants.

- Structures de données riches telles que les listes et les arbres.

- Les fonctions d'ordre supérieur où les fonctions sont elles mêmes des valeurs calculatoires. Certains langages permettent de passer des fonctions en paramètres mais peu de langages permettent que les fonctions jouent un rôle dans les structures de données.

- Structures de données infinies simplifiant la résolution des problèmes complexes (lazy evaluation). Un exemple typique de ces structures infinies sont les "pipes" d'Unix.

- grâce aux fonctions, on dispose des **spécifications exécutables** (e.g. PGCD)**.** A ce titre, ils sont dans la classe des langages déclaratifs : on spécifie (mathématique) et la machine exécute la spécification. Un objectif plus réaliste de la programmation déclarative est de disposer des programmes faciles à comprendre.

- Grâce au polymorphisme, la généricité est plus simple à mettre en place.
- Certains langages tels que SML permettent de spécifier la généricité au sens TDA à travers des modules génériques (cf. ADA).

- Les exceptions

Programmation impératif et fonctionnel en CAML

Nous avons vu des exemples en exploitant la partie déclarative de CAML en écrivant des fonctions un résultat calculé. Ces fonctions calculent des résultats au sens mathématique : par simplification successive. Ce style de programmation à l'aide de fonction s'appelle la programmation fonctionnelle.

Une autre façon de calculer dépendant du temps consiste à modifier un état : la machine commence les calculs dans un état initial que l'exécution du programme modifie jusqu'à parvenir à un état final représentant le résultat.

L'état courant évolue par la modification du contenu de la mémoire (à l'aide de l'affectation) ou par les entrées sorties. Toutes ses opérations qui modifient physiquement le contenu des cases mémoires sont appelées des effets de bord. Ainsi, un effet de bord est une modification d'une case mémoire ou une interaction avec le monde extérieur. Ce style de programmation par effet s'appelle la programmation impérative. Le style impératif s'applique aux programmes qui décrivent une suite d'opérations à effectuer pour modifier un état initial vers un état final. Par exemple, pour calculer x^2, on écrit x*x en programmation fonctionnelle alors que dans une méthode impérative, on initialise une case mémoire par x puis on multiplie le contenu de la case par sa valeur qui représentera le résultat. Programmer dans le style impératif nécessite des structures de données modifiables (par exemple les tableaux dont les éléments seront modifiés pendant l'exécution) et l'emploi de commandes. Ces commandes ne rendent pas de valeur mais modifient l'état courant. Une fonction qui exécute une série de commandes est appelée une procédure.

CAML permet les deux modes de programmation. De plus, certains algorithmes s'expriment en terme de modification d'états.

Puisque la notion de temps est importante dans la programmation impérative, il faut pouvoir décrire des séquences.

Exemples d'effets de bord :

Exemple-1 :

```
#print_string "bonjour !!";;          (* imprime_chaine : un prédéfini
CAML*)
bonjour !!- : unit = ()
```

CAML nous indique que nous avons calculé un résultat de type *unit* qui vaut (). Le type unit est un type prédéfini qui ne contient le seul élément "()" qui signifie rien. Puisque toute fonction CAML prend un argument et rend un résultat, cette valeur est rendue par CAML lorsqu'une fonction n'a en fait pas de résultat. Elle est également utilisée en guise d'argument.

Exemple-2 :

```
#print_string "bonjour " ;  print_string "tout le monde !!";;
bonjour tout le monde !!- : unit = ()
```

On a exprimé la séquence par ";". L'évaluation de deux expressions e1 et e2 en séquence consiste à évaluer e1, oublier le résultat puis, évaluer e2. Il ne faut donc pas utiliser une expression à la place de e1 dont la valeur est utile. Par exemple, dans

```
#sin 45.0 ; print_string "ca y est";;
ca y est- : unit = ()
```

La valeur de sin 45.0 est perdue.

Une séquence peut être encadrer entre *begin* et *end* :

```
#begin
      print_string "hello ";
      print_string "tout le monde!";
      print_newline ()          (* impression d'un retour-chariot, sans
paramètre *)
end;;
hello tout le monde!
- : unit = ()
```

Exemple d'opérateur

```
##infix "union";;          (* le ## pour dire : taper sous caml #infix....)
#let prefix union = fun x y -> 1;;
union : 'a -> 'b -> int = <fun>
#1 union 2;;
- : int = 1
```

Exemple de déclarations de types :

```
#type x == int list;;       voir la différence entre les deux
Type x defined.

#type c = a | b;;
Type c defined.
```

Exemple de typage explicite des arguments:

```
#let compose (f : 'b -> 'c)  (g : 'a -> 'b) = fun (x : 'a) -> f(g(x));;
compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

exemple de définition de fonction d'ordre supérieur :

```
let rec filtre (f : 'a -> bool)  = fun
 [] -> []
| (x :: xs) -> let y = (if (f x) then [x] else []) in  (y @ filtre f xs);;
filtre : ('a -> bool) -> 'a list -> 'a list = <fun>
```

```
#filtre odd [1;2;3];;
- : int list = [1; 3]
```

<il faudra donner l'exemple d'unifiaction écrit en CAML et d'un interprère Prolog, voir ex KB>

< aussi voir p. 337 de livre langage caml pour un automate à coder comme exemple >