Lycée Thiers **Année 2014-15**

Sup MPSI **Option Informatique**

CAML LIGHT 1 - ASPECTS IMPÉRATIFS DE **CAML**

Les fonctions constituent le concept central de CAML, qui pour cette raison est appelé un langage fonctionnel. Citons Pierre Weis, l'un des principaux développeurs de **CAML** : "il n'y a pas de restriction à la définition et à l'usage des fonctions, qu'on peut librement passer en argument ou retourner en résultat dans les programmes". La **programmation fonctionnelle** est un style de programmation où, pour l'essentiel, on définit des fonctions que l'on applique. Un programme est une expression, exécuter le programme revient à évaluer cette expression.

La **programmation impérative**[1] repose sur un schéma conceptuel différent : un programme est vu comme une séquence d'instructions ; chacune d'elles ayant pour effet de modifier l'état de la mémoire de l'ordinateur (typiquement : une affectation modifie la valeur d'une variable, c'est-à-dire remplace le contenu d'une "case mémoire" par un autre contenu). Exécuter un programme consiste, à partir d'un état initial, à effectuer une suite finie d'instructions.

A vrai dire, **CAML** n'est pas un langage *purement* fonctionnel : il comporte aussi des traits impératifs. Cela signifie qu'il est possible d'utiliser des **références** (qui correspondent aux variables du **PASCAL** : cases mémoires dont le contenu est modifiable), de manipuler des tableaux (appelés **vecteurs**) dont les éléments sont modifiables, ainsi que des **chaînes de caractères**. Plus généralement, **CAML** permet de définir des types "mutables" (ce sujet sera repris dans un document ultérieur).

D'autres constructions typiques d'une programmation impérative sont les **boucles** et (encore que cela soit discutable) les **tests**.

1. Références

En **PASCAL**, si x est une variable de type `integer` (l'analogue du type `int` de **CAML**), alors l'instruction `x := x + 1` a pour effet d'incrémenter `x` (càd : lui ajouter 1). Cette syntaxe est quelque peu ambigüe. En effet, le même symbole `x` est utilisé pour désigner deux entités distinctes : à gauche de `:=`, il s'agit de la **variable** `x` (une "case mémoire"), tandis qu'à droite, il s'agit de la **valeur** de cette variable (le "contenu" de la case mémoire).

En **CAML**, la syntaxe n'est pas ambigüe :

```
let x = ref 3;;
x : int ref = ref 3

!x;;
- : int = 3

x := 4;; (* observer : la valeur d'une affectation est () *)
- : unit = ()

!x;;
- : int = 4

x :=!x + 1;; (* On constate ici la non-ambiguité signalée plus haut *)
- : unit = ()

x;;
- : int ref = ref 5
```

1. on dit aussi "procédurale".

On dit que `x` est une **référence** vers un `int`. Le type de la variable x est `int ref`.

Les références de CAML sont l'analogue des pointeurs [2] de PASCAL ou C.

A noter que le mot-clef `ref` désigne, à la fois, le nom d'un type paramétré et le nom du constructeur.

Pour accéder à la valeur contenue dans une référence, on utilise l'opérateur (unaire) de **déréférencement**, noté **!**

Pour modifier la valeur contenue dans une référence, on utilise l'opérateur (binaire) d'**affectation**, noté **:=**

Une autre syntaxe pour l'expression `x := !x + 1` (resp. `x := !x - 1`) est `incr x` (resp. `decr x`).

En résumé :

Création d'une référence :

`let` identificateur = `ref` valeur [`in` ...]

Accès à la valeur :

! identificateur

Modification de la valeur :

identificateur := valeur

Les adresses *physiques* des objets ne sont pas accessibles en CAML.

2. TESTS

La syntaxe générale d'une expression conditionnelle est :

`if` condition `then` expr1 `else` expr2

où **condition** est une expression de type `bool`, et où **expr1**, **expr2** sont des expressions de même type (voir 2.4 ci-après). La valeur de l'expression conditionnelle est ce type commun.

Examinons quelques exemples ...

2.1. **Signe d'un entier.** La fonction `signe` renvoie -1, 0 ou 1 selon que l'argument est négatif, nul ou positif :

```
let signe x =
  if x < 0 then -1 else if x > 0 then 1 else 0
;;
signe : int -> int = <fun>
```

2.2. **Comparaison de trois quantités.** La fonction `max3` renvoie le plus grand de ses trois arguments :

```
let max3 a b c =
  let m = if a < b then b else a in
    if m < c then c else m
;;
max3 : 'a -> 'a -> 'a -> 'a = <fun>
```

On notera au passage que `max3` est **polymorphe** (comme l'opérateur <) :

2. pointeur = une variable ayant pour valeur l'adresse en mémoire d'une autre.

```
max3 4 3 8;;
- : int = 8

max3 'B' 'I' 'C';;
- : char = 'I'

max3 "chameau" "dromadaire" "bosse";;
- : string = "dromadaire"
```

2.3. **Utilisation de begin et end.** Si l'expression **expr1** est une séquence d'expressions (séparées par des points-virgules), il est nécessaire de l'encadrer par un couple de parenthèses, ou (si l'on préfère) par les mots-clefs `begin` et `end`. Même chose, bien sûr, pour **expr2** :

```
let affiche_bizarre a b =
  if a < b then (
    print_int a;
    print_string " est plus petit que ";
    print_int b
  )
  else (
    print_string "le produit de ";
    print_int a;
    print_string " par ";
    print_int b;
    print_string " vaut : ";
    print_int (a*b)
  );
  print_newline ()
;;
affiche_bizarre : int -> int -> unit = <fun>

affiche_bizarre 3 4;;
3 est plus petit que 4
- : unit = ()

affiche_bizarre 4 3;;
le produit de 4 par 3 vaut : 12
- : unit = ()
```

L'omission des parenthèses (ou de `begin` et `end`) peut conduire à des erreurs de syntaxe (qui sont donc détectées à la compilation), mais aussi à des problèmes qui ne se manifestent qu'à l'exécution. Exemple :

```
let f x =
  if x = 0 then
    print_string "Division par 0\n"
  else
    print_string "L'inverse est : \n";
    print_float (1. /. (float_of_int x));
    print_newline ()
;;
f : int -> unit = <fun>

f 8;;
L'inverse est :
0.125
- : unit = ()
```

```
f 0;;
Division par 0
inf.0
- : unit = ()
```

On constate que la division par 0 a quand même eut lieu !

La solution correcte consiste à grouper ce qui suit le `else` :

```
let f x =
  if x = 0 then
    print_string "Division par 0\n"
  else (
    print_string "L'inverse est : \n";
    print_float (1. /. (float_of_int x));
    print_newline ();
  )
;;
f : int -> unit = <fun>

f 0;;
Division par 0
- : unit = ()
```

2.4. **Erreurs fréquentes.** Les expressions `expr_1` et `expr_2`, correspondant aux deux branches du test, doivent être du même type :

```
let racine x =
   if x > 0.0
  then sqrt x
  else "erreur"
;;
Toplevel input:
>  else "erreur"
>       ^^^^^^^^
This expression has type string,
but is used with type float.
```

Si le type de `expr1` est `unit`, alors la partie `else` devient optionnelle :

`if` condition `then` expr

```
let affiche mot1 mot2 saut_de_ligne =
  print_string mot1;
  if saut_de_ligne then print_newline ();
  print_string mot2
;;
affiche : string -> string -> bool -> unit = <fun>
affiche "voici " "voila" false;;
voici voila- : unit = ()
affiche "voici " "voila" true;;
voici  voila- : unit = ()
```

mais sinon, sa présence est obligatoire :

```
let racine x =
   if x > 0.0
  then sqrt x
;;
Toplevel input:
>..if x > 0.0
>  then sqrt x
This expression has type unit,
but is used with type float.
```

3. Boucles

Une boucle consiste en l'évaluation répétée d'une expression. Cette répétition se poursuit soit un nombre prédéterminé de fois (boucle inconditionnelle), soit tant qu'une certaine condition est vraie (boucle conditionnelle).

3.1. **Boucle inconditionnelle.** La syntaxe générale est :

`for` i = expr1 `to` expr2 `do` expr `done`

Les expressions `expr1` et `expr2` sont d'abord évaluées en des entiers `i1` et `i2`, puis l'expression `expr` est évaluée `i2-i1+1` fois consécutivement (sauf si `i1 > i2`, auquel cas elle n'est pas évaluée du tout). Comme dans le cas des tests, `expr` peut être une séquence d'expressions (mais cette fois, plus besoin de parenthèses, puisque les mots-clefs `do` et `done` jouent déjà le rôle de délimiteurs).

La valeur d'une expression `for ... done` est `()`. Pour effectuer une boucle "descendante", on remplace `to` par `downto`.

3.1.1. *Factorielle d'un entier.*

```
let fact n =
  let p = ref 1 in
    for k = 1 to n do p := !p * k done;
   !p
;;
fact : int -> int = <fun>

fact 5;;
- : int = 120
```

3.1.2. *Suite de Fibonacci.*

```
let fib n =
  let a = ref 0 and b = ref 1 in
    for k = 2 to n do
       let c = !a + !b in
         a := !b;
         b := c
    done;
   !b
;;
fib : int -> int = <fun>

fib 12;;
- : int = 144
```

3.1.3. *Deux boucles imbriquées.* Soit à calculer, pour $n \in \mathbb{N}^*$ donné, l'expression $S_n = \sum_{i=0}^{n} \left(\sum_{j=0}^{n} \frac{1}{i+j^2+1} \right)$:

```
let u n =
  let s = ref 0. in
    for i = 0 to n do
      for j = 0 to n do
        let r = float_of_int (i + j*j + 1) in
          s :=!s +. 1. /. r
      done
    done;
   !s
;;
u : int -> float = <fun>

u 50;;
- : float = 21.5105062553
```

3.2. **Boucle conditionnelle.** La syntaxe générale est :

`while` condition `do` expr `done`

L'expression `expr` est évaluée de façon répétée, tant que l'expression `condition` est évaluée à `true`. Comme pour les boucles `for`, `expr` peut être une séquence d'expressions.

La valeur de l'expression `while ... done` est `()`.

Principal problème posé par cette construction : si la condition n'est jamais évaluée à `false`, la boucle ne se termine pas ... Il faudra donc, pour chaque cas étudié, s'assurer de la terminaison. Pour cela, on essaie généralement de mettre en évidence une quantité entière et positive qui décroît strictement à chaque tour de boucle. Ce thème sera développé en classe.

3.2.1. *Liste des caractères ASCII.*

```
let ascii debut fin =
  let i = ref debut in
    while!i <= fin do
      print_char (char_of_int!i);
      print_char ' ';
      incr i;
      if ((!i - debut) mod 25 = 0) then print_newline ()
    done;
    print_newline ()
;;
```

Les caractères sont numérotés, selon le code ASCII (American Standard Code for Information Interchange [3]), de 0 à 127. Les caractères 0 à 31 ainsi que 127 ne sont pas imprimables (ce sont des caractères de contrôle, par exemple : 9 → TAB (tabulation horizontale), 10 → LF (Line Feed = saut de ligne), 13 → CR (carriage return = retour chariot), 127 → DEL (suppression de caractère) ...). Le caractère de code ASCII 32 est l'espace ...

```
ascii 32 126;;
 ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8
9 : ; < = >? @ A B C D E F G H I J K L M N O P Q
R S T U V W X Y Z [ \ ] ^ _ ` a b c d e f g h i j
k l m n o p q r s t u v w x y z { | } ~
- : unit = ()
```

3. On pourra consulter par exemple *http ://fr.wikipedia.org/wiki/ASCII*

3.2.2. *Algorithme d'Euclide.* Bien que l'opérateur `mod` soit prédéfini, écrivons une fonction qui calcule le reste de la division euclidienne de a par b (a,b entiers positifs et b non nul) :

```
let reste a b =
  let r = ref a in
    while (!r >= b) do
      r :=!r - b
    done;
   !r
;;
reste : int -> int -> int = <fun>

let pgcd u v =
 let a = ref u and b = ref v in
    while!b <> 0 do
      let t =!b in
        b := reste!a!b;
        a := t
    done;
   !a (* le pgcd est donné par le dernier reste non nul *)
;;
pgcd : int -> int -> int = <fun>

pgcd 171568 45320;;
- : int = 8
```

4. Deux Types Mutables

4.1. **Vecteurs.**

4.1.1. *Généralités.* Un vecteur peut être vu comme une succession de cases mémoire contigües. Chacune est directement accessible par son numéro d'ordre (son index) et la numérotation commence à 0.

La création d'un vecteur peut se faire par citation, en utilisant les délimiteurs [4] `[|` et `|]` :

```
let v = [|1;4;1;4;2|];;
v : int vect = [|1;4;1;4;2|]

let w = [|"Ceci","est","un","vecteur"|];;
w : string vect = [|"Ceci","est","un","vecteur"|]
```

On accède en lecture à un élément par la syntaxe **vecteur.(index)**

```
v.(1);;
- : int = 4

w.(3);;
- : string = "vecteur"
```

L'accès à une case qui n'existe pas déclenche une exception :

```
w.(10);;
Uncaught exception: Invalid_argument "vect_item"
```

On peut aussi créer un vecteur en utilisant la fonction `make_vect` :

4. Attention : les délimiteurs `[` et `]` seront utilisés pour les listes.

```
let v = make_vect 5 0;; (* vecteur de longueur 5, initialement nul *)
v : int vect = [|0; 0; 0; 0; 0|]
```

puis en "remplissant" à l'aide d'affectations :

```
v.(0) <- 1; v.(1) <- 4; v.(2) <- 1; v.(3) <- 4; v.(4) <- 2;;
- : unit = ()
```

D'une façon ou d'une autre, **la longueur d'un vecteur est fixée définitivement à sa création** (un vecteur est une structure de données **statique**, contrairement à une liste, qui est une structure de données **dynamique**). La fonction pré-définie `vect_length` renvoie la longueur d'un vecteur :

```
vect_length v;;
- : int = 5
```

On a vu ci-dessus qu'un vecteur peut être, par exemple, de type `int vect` ou de type `string vect`. En fait, il peut s'agir d'un vecteur de n'importe quoi ... Le type `vect` est un **type paramétré**. Cela entraîne logiquement que la fonction `vect_length` soit **polymorphe** :

```
vect_length;;
- : 'a vect -> int = <fun>
```

De même, le vecteur vide, noté `[||]`, n'a pas de type précis :

```
[||];;
- : 'a vect = [||]
```

`'a` (lire α) est une **variable de type** : elle désigne n'importe quel type.

4.1.2. *Deux exemples.* La fonction suivante calcule le plus grand élément d'un vecteur :

```
let max_vect v =
  let n = vect_length v in
  let m = ref v.(0) in
    for i = 1 to n-1 do
      if v.(i) > !m then m := v.(i)
    done;
   !m
;;
max_vect : 'a vect -> 'a = <fun>

max_vect [|3;1;4;1;5;9;2|];;
- : int = 9
```

La fonction suivante teste la présence d'un élément dans un vecteur :

```
let is_in_vect x v =
  let n = vect_length v in
    let result = ref false in
      let i = ref 0 in
        while !i < n & !result = false do
          if v.(!i) = x then result := true;
          incr i
       done;
       !result
;;
is_in_vect : 'a -> 'a vect -> bool = <fun>

is_in_vect 7 [|3;1;4;1;5;9;2|];;
```

```
- : bool = false

is_in_vect 5 [|3;1;4;1;5;9;2|];;
- : bool = true
```

4.2. Chaines de caractères.

4.2.1. *Généralités.* Une chaîne de caractère est délimitée par une paire de guillemets [5] :

```
let s = "Hello world !";;
s : string = "Hello world !"
```

On accède à chaque caractère de la chaine par son rang (à partir de 0, comme pour les vecteurs). La syntaxe est **chaine.[index]** :

```
s.[0];;
- : char = 'H'

s.[50];;
Uncaught exception: Invalid_argument "nth_char"
```

Le message d'erreur ci-dessus nous apprend que cette syntaxe est une abbréviation [6] pour **nth_char chaine index** :

```
nth_char s 4;;
- : char = 'o'
```

Exercice → Trouver l'analogue de `nth_char` *pour les vecteurs.*

Pour modifier la valeur d'un caractère, la syntaxe est similaire à celle vue pour les vecteurs :

```
s.[0] <- 'h';;
- : unit = ()

s;;
- : string = "hello world !"
```

La fonction `string_length` renvoie la longueur d'une chaine :

```
string_length s;;
- : int = 13
```

Pour mettre bout à bout deux chaines de caractères, on dispose de l'opérateur de **concaténation** ^ :

```
"un, deux" ^ " et trois !";;
- : string = "un, deux et trois !"

let s1 = "Ceci explique" in
  let s2 = " cela\n" in     (* le symbole \n représente le saut de ligne *)
    print_string (s1 ^ s2);;
Ceci explique cela
- : unit = ()

let concatene = prefix ^;;
concatene : string -> string -> string = <fun>
concatene "un, deux" " et trois !";;
- : string = "un, deux et trois !"
```

5. le caractère " doit être précédé d'un backslash s'il doit faire partie de la chaîne.

6. A ne pas confondre avec `v.(k)`, qui suppose que v est un vecteur. Une chaine de caractères n'est pas un vecteur de caractères !

Montrons pour finir comment on extrait une sous-chaine d'une chaine de caractères donnée :

```
let s = "Ma chaine de caractères" in
  sub_string s 3 5;;
- : string = "chain"
```

La syntaxe de `sub_string` est donc `sub_string chaine debut longueur` :

```
sub_string;;
- : string -> int -> int -> string = <fun>
```

4.2.2. *Deux exemples.* La fonction suivante compte le nombre de chiffres d'une chaine de caractères :

```
let nb_chiffres s =
  let n = string_length s in
    let nb = ref 0 in
      for i = 0 to n-1 do
        let c = s.[i] in
          if (c >= '0') & (c <= '9') then incr nb
      done;
      !nb
;;
nb_chiffres : string -> int = <fun>

nb_chiffres "Le mois de février comporte 28 ou 29 jours";;
- : int = 4
```

Un palindrome est une chaine de caractères qui coïncide avec son "image mirroir".

Exemple : "radar" est un palindrome. La fonction suivante détermine si la chaine passée en argument est un palindrome :

```
let is_palindr s =
  let n = string_length s in
    let s' = make_string n '-' in
      for i = 0 to n-1 do
        s'.[i] <- s.[n-1-i]
      done;
      s = s'
;;
is_palindr : string -> bool = <fun>

is_palindr "tulastropecrasecesarceportsalut";;
- : bool = true

is_palindr "cette phrase n'est pas un palindrome";;
- : bool = false
```

Lorsque l'écriture de fonctions récursives aura été abordée, on pourra revenir sur cet exemple ...