1 Introduction à Ocaml

Ocaml est un langage de la famille ML développé à l'INRIA depuis 1995. Plusieurs raisons justifient son choix pour l'étude des traits objets:

- Il présente les traits usuels des langages à objets : classes, méthodes, classes virtuelles, héritage.
- C'est un langage très complet, avec des traits fonctionnels et impératifs ainsi qu'un puissant système de modules.
- Il s'agit d'un langage fortement typé (avec inférence des types), ce qui en fait un langage sûr et facile à employer.
- Ocaml est langage compilé qui possde une mode intéractif, ce qui facilite les tests.
- Ocaml autorise le polymorphisme paramétrique des classes, en y introduisant ainsi une dose de généricité.
- La sémantique d'Ocaml est bien définie et rend ainsi plus aisée l'étude des différents traits du langage, dont les traits objet.

Pour la documentation, distribution et plus de détails, voir dans
`http://caml.inria.fr/ocaml`.

2 Le mode intéractif

Ocaml possde un mode intéractif o il analyse et répond à chaque phrase entrée par l'utilisateur. Pour lancer Ocaml en mode intéractif tapez (sur `kirov`) la commande `ocaml` Vous aurez la réponse:

```
% ocaml
        Objective Caml version 3.04

#
```

Le caractère # invite l'utilisateur à entrer une phrase écrite dans la syntaxe Ocaml , phrase qui par exemple nomme une valeur, explicite une définition ou décrit un algorithme. Chaque phrase Ocaml doit terminer par `;;` puis l'utilisateur valide sa frappe par un retour chariot. Dès lors, Ocaml analyse la phrase , calcule son type (inférence des types), la traduit en langage exécutable (compilation) et enfin l'exéecute pour fournir la réponse demandée. En mode intéractif, Ocaml donne systématiquement une réponse qui contient:

- Le nom de la variable déclarée s'il y en a.
- Le type inféré par le typage.
- La valeur calculée après exécution.

Par exemple:

```
# let x = 4+2;;
val x : int = 6
```

La réponse d'Ocaml signale que l'identificateur `x` est déclaré `(val x)`, avec le type des entiers `(:int)` et la valeur calculée 6 `(=6)`. Un exemple qui utilise la valeur `x` déjà déclarée:

```
# x * 3;;
- : int = 18
```

Enfin vous pouvez mettre des commentaires comme ceci:

```
# (* ceci est un premier commentaire *)
let unefonction x  =
       (* ceci est un autres commentaire dans le corps de la fonction *)
       x+1;;
val unefonction : int -> int = <fun>
```

3 Les valeurs

Les valeurs manipulées en Ocaml sont soit les objets, soit les valeurs de base (de type int, bool, string, etc.), soit des fonctions, soit des valeurs de types construits, pre-définis (listes, tableaux, etc) ou définis par l'utilisateur. En dehors des déclarations (des valeurs, des classes, des modules, des types) toute phrase Ocaml est considérée comme une expression et dénote ainsi une valeur résultat.

4 Déclarations globales et locales

Un identificateur est déclaré globalement par le mot clef let:

```
# let x = 7;;
val x : int = 7

# x+2;;
- : int = 9
```

Un identificateur est déclaré localement par la construction `let ... in`:

```
#let y = 5 in x *y;
 - : int =45
```

Les identificateurs déclarées globalement ont une portée globale. Une déclaration locale est visible seulement dans l'expression qui suit le `in`:

```
#let y = 3 in x+y;;
- : int = 10
#y;;
Unbound value y
```

Les identificateur déclarés jusqu'ici sont constants: la liaison entre un identificateur `x` et sa valeur v, au moyen d'une déclaration `let x =`v, ne peut être modifiée. On pourra introduire éventuellement une nouvelle liaison pour x, `let x =` v_1 qui masque la première dans la suite du programme, mais on ne pourra pas la modifier. Plus tard nous introduirons les valeurs modifiables.

5 Les types

5.1 Types de base

- entiers (`int`), opérateurs: `+ - * /`
- reéls (`float`), opérateurs: `+. -. *. /.`
- caractéres (`char`), `'a', 'b', '1'` ...
- chaînes, (`string`), `"Bonjour", "ab"`; opérateur de concaténation: ^
- booléans, (`bool`), `true, false` opérateurs: `or`, `&`, `not`.
- et, les opérateurs de comparaison, définis pour tous les types: `=, > , < , >=, <=, <>`.

<u>Exemples</u>:

```
# 1+ 2;;
- : int = 3

# 1.5 +. 2.3;;
- : float = 3.8

# let x = "cou" in x^x;;
- : string = "coucou"

# 2 > 7;;
- : bool = false

# "bonjour" > "bon";;
- : bool = true

# "durand" < "martin";;
- : bool = true

# "ab" = "ba";;
- : bool = false
```

5.2 Types construits prédéfinis

Ce sont des types prédéfinis, construits à partir d'autres types:

- n-uplets (`t1 * t2 * .. * tn`), où `ti` est le type de la composante `i` du n-uplet. Exemple: (1,true,"ab") est un 3-uplet de type `int * bool * string`.
- listes (`t list`), oú `t` est le type des éléments.
- vecteurs (`t array`), ou' `t` est le type des éléments.

Exemples:

```
(* un 3-uplet *)
# (1,true,"ab");;
- : int * bool * string = 1, true, "ab"

(* une paire  *)
# let b = (2,3);;
val b : int * int = 2, 3

(* fst et snd sont des fonctions predefinies *)
# fst (2,3);;
- : int = 2

# snd (2,3);;
- : int = 3

(* une liste d'entiers *)
# [1;2;3];;
- : int list = [1; 2; 3]

(* @: op. de concatenation *)
# ["a";"bc"] @ ["bonjour"];;
- : string list = ["a"; "bc"; "bonjour"]
```

```
(* []: liste vide  *)
# [1]@[];;
- : int list = [1]

(* List: module de fonctions sur les listes  *)
# List.length [1;2;3];;
- : int = 3

(* un tableau d'entiers *)
# let a = [|1;2;3|];;
val a : int array = [|1; 2; 3|]

(* affectation de la premiere composante du tableau a *)
# a.(0) <- 50;;
- : unit = ()
# a;;
- : int array = [|50; 2; 3|]
(* Array: module pour tableaux *)
# Array.length a;;
- : int = 3
```

5.3 Types construits de l'utilisateur

Il deux sortes principales de types construits pouvant être déclaré par le programmeur:

- **enregistrements**: agrégat de valeurs hétérogènes, chacune nommée et accesible par un nom de champ ou étiquette.
- **somme disjointe**: permet la définition de types par l'énumération des identificateurs constants qui y appartiennent (somme des constantes); par la l'injection des constantes d'un ou plusieurs types à l'aide d'étiquettes, appelées *constructeurs*; et enfin, la définition de types récursifs.

5.3.1 Enregistrements

La déclaration d'un type enregistrement se fait en donnant les noms de ses champs et leurs types respectifs. À moins de préciser le contraire, les champs d'un enregistrment ne sont pas modifiables.

```
(* Un type enregistrement *)

# type client = {numero: int; nom: string; solde: float};;
type client = { numero: int; nom: string; solde: float }

(* Une valeur de ce type dans l'identificateur durand *)

# let durand = { numero = 265; nom = "Durand"; solde=  0.0};;
val durand : client =  {numero=265; nom="Durand"; solde=0}

(* Acces vers un champ *)
# durand.numero;;
- : int = 265
```

5.3.2 Types somme

Un type somme est défini par la liste des cas décrivant la forme de toutes les valeurs appartenant au type, séparés par le symbole `|`, prononcé "ou". La forme d'une valeur est donnée par son *constructeur*, accompagné éventuellement d'un

type. Dans le schéma suivant, le constructeur $Constr_1$ est accompagné du type t_1 alors que $Constr_2$ n'est acompagné d'aucun type:

$$\begin{array}{lcll} \texttt{type}\ ident_type & ::= & Constr_1 & t_1 \\ & | & Constr_2 & \\ \vdots & | & Constr_n & t_n \end{array}$$

En Ocaml les constructeurs sont de mots qui commencent toujours par une majuscule. Lorsqu'un constructeur n'est accompagné d'aucun type, il est dit *constant*. Dans ce premier exemple, les constructeurs sont constants. Le type somme devient alors simplement l'énumération des constantes appartenant au type:

```
(* Type somme par enumeration des couleurs *)

# type couleur_primaire = Rouge | Vert | Bleu;;
type couleur_primaire = Rouge | Vert | Bleu
```

Cette déclaration est lue: "une valeur du type `couleur_primaire` est soit `Rouge`, ou `Vert`, ou `Bleu`".

```
(* Une valeur de type couleur_primaire *)

# Rouge;;
- : couleur_primaire = Rouge

(* Type enregistrement des points avec couleur *)
(* utilise le type couleur                     *)

# type point_colore = { x: int; c: couleur_primaire};;
type point_colore = { x: int; c: couleur_primaire }

(* une valeur de ce type dans pc *)

# let pc = {x = 1; c = Rouge};;
val pc : point_colore = {x=1; c=Rouge}
```

Les constructeurs d'un type somme peuvent s'accompagner d'une valeur à condition que celle-ci soit du type déclaré pour accompagner ce nom de constructeur. Le type `carte` décrit les cartes d'un jeu des cartes. Chacune d'entre elles est définie par sa catégorie (`Roi`, `Dame`, etc), et par sa couleur (`Pique`, `Coeur`, etc). Le type carte est décrit par l'énumération des catégories, chacune accompagnée d'une couleur, parmi toutes les couleurs possibles du type `couleur`.

```
(* Le types des couleurs des cartes *)

# type couleur = Pique | Coeur | Carreau | Trefle;;
type couleur = Pique | Coeur | Carreau | Trefle

(* Un type somme decrivant un jeu de cartes *)

# type carte = As of couleur
             | Roi of couleur
             | Dame of couleur
```

```
              | Valet of couleur
              | Petite of couleur * int
              | Joker


type carte =
    As of couleur
  | Roi of couleur
  | Dame of couleur
  | Valet of couleur
  | Petite of couleur * int
  | Joker
```

Les différentes catégories des cartes sont modélisées ici par les constructeurs `Roi`, `Dame`, etc. La catégorie `Petite_couleur` est accompagnée de deux valeurs: sa couleur et sa valeur numérique. Afin de construire une carte, il faut donner sa catégorie (constructeur) suivie de la valeur qui l'accompagne en respectant son type décrit lors de la déclaration.

```
# let roi_pique = Roi (Pique);;
roi_pique : carte = Roi Pique

# let sept_coeur = Petite(Coeur,10);;
sept_coeur : carte = Petite (Coeur, 10)

# Joker;;
- : carte = Joker
```

5.3.3 Types somme récursifs

Un type somme récursif permet de construire des structures des données récursives, telles les listes chaînées ou les arbres. Voici un type somme récursif qui permet le mélange des couleurs. Chaque valeur d'un type récursif a une structure arborescente, dont les noeuds sont les constructeurs avec arguments, et les feuilles sont les constructeurs constants.

```
# type palette = Rouge | Vert | Bleu | Melange of palette * palette;;
type palette = Rouge | Vert | Bleu | Melange of palette * palette

# let violet = Melange(Rouge, Bleu);;
val violet : palette = Melange (Rouge, Bleu)

# let violet_profond = Melange(violet, Bleu);;
val violet_profond : palette = Melange (Melange (Rouge, Bleu), Bleu)

(* Un type somme pour les arbres biraires d'entiers *)

# type int_arbre = Vide | Noeud of int_arbre * int * int_arbre;;
type int_arbre = Vide | Noeud of int_arbre * int * int_arbre

# let un = Noeud(Vide, 1, Vide);;
val un : int_arbre = Noeud (Vide, 1, Vide)
```

6 Tests et alternative

Ocaml founit la construction classique `if...then...else` pour exprimer l'alternative.

```
# if 3>5 then "trois superieur a cinq" else "trois inferieur a cinq";;
- : string = "trois inferieur a cinq"
```

Pour respecter les contraintes de typage, les expressions à exécuter dans chacune des branches doivent avoir le même type.

```
 if 3>5 then "trois superieur a cinq" else 5;;
Characters 43-44:
This expression has type int but is here used with type string
```

7 Les fonctions

Un identificateur de fonction est déclaré comme tout autre identificateur, à l'aide d'un `let`:

```
# let double (y) = y*2;;
val double : int -> int = < fun >
```

Le type d'une fonction est noté `t` → `q`, o `t` est le type des arguments et `q` celui des résultats de la fonction. Dans le cas de la fonction `double`, l'argument et le résultat sont de type entier. Voici deux syntaxes alternatives pour la même définition: dans la première on ommet les parenthèses lors de la définition de l'argument; dans la deuxième on utilise la notation "fonctionnelle" ou du lambda-calcul.

```
# let double y = y*2;;
val double : int -> int = < fun >

# let double = fun y -> y*2;;
val double : int -> int = < fun >
```

La deuxième est lue: "double est la fonction qui a `y` associe `y*2`. On applique une fonction en la faisant suivre de son argument (éventuellement entre parenthèses):

```
# double(9);;
- : int = 18

# double 9;;
- : int = 18

# let z = double(x) ;;
val z : int = 12
```

Il est également possible de définir des fonction récursives, à l'aide du mot-clé `rec`:

```
# let rec factorielle n =
     if n<2 then 1  else n*factorielle (n-1);;
val factorielle : int -> int = <fun>
# factorielle 5;;
- : int = 120
```

Une fonction peut prendre plusieurs arguments, en séparant les noms des paramètres formels par des espaces, avant la définition du corps:

```
# let creer_pointcolore n col = {x = n; c = col};;
val creer_pointcolore : int -> couleur -> point_colore = < fun >

# let pc = creer_pointcolore 1 Rouge;;
```

```
val pc : point_colore = {x=1; c=Rouge}

#  let moyenne x y = (x+.y)/. 2.0;;
val moyenne : float -> float -> float = <fun>
```

Si f est une fonction à deux arguments, son type est alors de la forme: $t_1 \rightarrow t_2 \rightarrow t_{corps}$ où, t_1, t_2 sont les types des arguments, et t_{corps} est le type du résultat. L'application des la fonction se fait en faisant suivre tous les arguments séparaés par des espaces:

```
# moyenne 15.0 12.6;;
- : float = 13.8
```

La fonction `moyenne` peut être vue de deux façons, soit comme une fonction a deux argument entiers renvoyant un entier, soit comme une fonction a un argument entier renvoyant une autre fonction des entiers vers les entiers. De même l'argument d'une fonction peut lui-même être une fonction:

```
# let f (h) = (h (1)) +2;;
val f : (int -> int) -> int = <fun>
```

Comparez ce type avec celui de la fonction moyenne. Une autre manière de définir une fonction à deux arguments, est de les lui passer dans une paire:

```
# let somme (x,y) = x+y;;
 val somme : int * int -> int = <fun>
```

Mais attention, une fonction telle que `somme` ne possède *q'un seul* argument, qui en l'occurence, est une paire. Ainsi, parmi les appels suivants, le premier est correct alors que le deuxième ne l'est pas:

```
# somme (2,3);;
- : int = 5
# somme 2 3;;
This function is applied to too many arguments
```

8 Le filtrage

Le filtrage de ML est un moyen de tester les cas de la structure d'un objet et de choisir des actions à effectuer selon chaque cas. La structure de chaque cas est décrite à l'aide d'un *filtre*. La syntaxe d'une expression de filtrage est de la forme:

```
 match expr
  with filtre_1 -> action_1
  |    filtre_2 -> action_2
  |    ....
  |    _        -> action_tous_autres_cas
```

La valeur filtrée est comparée succésivement à chacun des filtres selon l'ordre de leur défintion. Le symbole _ es lu "dans tous les autres cas". La fonction suivante filtre ses arguments (une paire) pour faire leur addition en tenant compte du cas où l'un d'entre eux est zéro:

```
# let somme x y =  match (x,y)
  with (0,n) -> n
  |    (n,0) -> n
  |    (a,b) -> a+b;;
val somme : int -> int -> int = < fun >

# somme 0 3;;
- : int = 3
# somme 2 4;;
- : int = 6
```

8.1 Types somme et filtrage

Un type somme décrit ses valeurs comment une disjonction des catégories. La manière la plus naturelle d'inspecter de telles valeurs est d'utliser le filtrage:

```
let valeur_belote (atout, carte) =
 match carte
 with As _         -> 11
   |  Roi _        -> 4
   |  Dame _       -> 3
   |  Valet c      -> if c = atout then  20 else 2
   |  Petite (_,10) -> 10
   |  Petite (c,9)  -> if c = atout then 14 else 0
   | _ -> 0;;
val valeur_belote : couleur * carte -> int = <fun>
```

en Ocaml, le type des listes, noté t `list`, et lu :"listes d'éléments de type t", est prédéfini. Il s'agit d'un type récursif. Sa définition en interne correspond à:

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

Les valeurs du type sont inspectés par le filtrage via deux constructeurs: `::` qui construit une liste à partir d'un élément `a` mis en tête, et une liste `l`, par `a::l`; et `[]` qui construit une liste vide. La fonction qui fait la somme des éléments dans une liste:

```
# let rec somme_liste l =  match l
 with []  -> 0                            (* []  filtre de liste vide     *)
  |  a::r -> a + somme_liste r;;          (* a::r liste avec a en premier *)
val somme_liste : int list -> int = < fun >(* et r en reste               *)

# somme_liste [2;3;7];;
- : int = 12
```

La fonction qui fait la somme des éléments dans un arbre d'entiers (de type `arbre_int` défini plus haut).

```
# let rec somme_arbre arb =  match arb
    with Vide  -> 0
     |  Noeud (a,n,b) -> somme_arbre a + n + somme_arbre b;;
val somme_arbre : int_arbre -> int = < fun >

somme_arbre (Noeud ((Noeud (Vide, 1, Vide)), 3, (Noeud (Vide, 5, Vide))));;
- : int = 9

#let rec member x btree =
   match btree with
     Vide -> false
   | Noeud (left, y,  right) ->
       if x = y then true else
       if x < y then member x left else member x right;;

val member : int -> int_arbre -> bool = <fun>
```

9 Inférence de types et polymorphisme

En Ocaml, il n'est pas nécessaire de déclarer les types des identificateurs ni des arguments d'une fonction intervenant dans une définition. En effet, le typeur infère leur type d'après leur contexte.

Exemple 1:

```
let f(x) = x + 1
```

Sachant que + est un opérateur définit uniquement sur les entiers, et dont le type est +: `int` $*$ `int` $\rightarrow$ `int`, le typeur déduit les contraintes suivantes:

- le type de `f` est de la forme:
$$\texttt{f} : t_x \rightarrow t_{corps}$$
où t_x est le type de x (argument de la fonction), et t_{corps} est le typen du corps.
- pour que `x+1` soit bien typé, `x` doit être de type `int` $\Rightarrow \boxed{t_x = \texttt{int}}$
- si ces contraintes sont respectées, alors le résultat de l'opération `x + 1` est de type `int`, $\boxed{t_{corps} = \texttt{int}}$.

Conclusion: `f` est de type `int` $\rightarrow$ `int`
□

Exemple 2:

```
let a = 3.0 in a +. 5.2
```

Dans cet exemple, Ocaml commence par typer la déclaration locale `let a = 3.0`:

- `3.0` est une constante de type `float`,
- ensuite, lors du typage de l'expression `a +. 5.2`, le type de l'opérateur `+. : float` $*$ `float` $\rightarrow$ `float`, impose les contraintes supplémentaires:
  - `a : int`,
  - `5.2 : int`

  Ce qui est éffectivement le cas. L'expression `a +. 5.2` est donc de type `float`

Conclusion: `(let a = 3.0 in a +. 5.2) : float`
□

Exemple 3:

```
let mot_compose (x,y) = x^"-"^y
```

- La fonction `mot_compose` prend deux arguments, ce qui impose une première contrainte, à savoir, son type est de la forme:
$$\texttt{mot_compose} : t_x * t_y \rightarrow t_{corps}$$
où t_x, t_y et t_{corps} sont respectivement, les types de `x`, de `y` et du corps de la fonction.
- L'opérateur ^ est de type `string` $*$ `string` $\rightarrow$ `string`. Son utilisation dans le corps de la fonction impose les contraintes:
  - `x` et `"-"` doivent être de type `string`, donc $\boxed{t_x = \texttt{string}}$,
  - si ces contraintes sont respectées, alors `(x^"-")` est de type `string`. Par conséquent `y` doit l'être aussi, donc $\boxed{t_y = \texttt{string}}$,
  - Dans le cas où toutes les contraintes sont respectées, le corps de la fonction `(x^"-")^y` est de type `string`, c.a.d, $\boxed{t_{corps} = \texttt{string}}$

Conclusion: `mot_compose` est de type $\texttt{string} * \texttt{string} \rightarrow \texttt{string}$
□

Certaines définitions de fonctions n'imposent aucune contrainte de type sur leurs arguments. Dans ce cas, leur type peut-être quelconque. On parle alors de *polymorphisme paramétrique*.

Exemple 4:

```
let premier (x,y) = x
```

- Le type de cette fonction est de la forme: `premier` : $t_x * t_y \rightarrow t_{corps}$, où t_x, t_y et t_{corps} sont les types de `x`, de `y` et du corps de la fonction.
- La seule contrainte imposée par l'examen du corps de la fonction, est que son type est le même que celui de la première composante de la paire argument de la fonction, i.e: $\boxed{t_{corps} = t_x}$,

Aucune autre contrainte ne se dégage de l'examen du code de cette fonction. D'après les hypothèses de départ, on obtiendrait alors:

$$\texttt{premier} : t_x * t_y \rightarrow t_x$$

t_x, t_y ne sont pas de véritables types mais des *variables de type*. Puisqu'aucune contrainte ne pèse sur eux, on pourrait choisir n'importe quel type à leur place et on obtiendrait un typage cohérent pour la fonction `premier`. Par exemple, on pourrait choisir parmi les typages suivants:

$$\texttt{premier} : \begin{cases} \texttt{int} * \texttt{bool} \rightarrow \texttt{int} \\ (\texttt{int} \rightarrow \texttt{int}) * \texttt{string} \rightarrow (\texttt{int} \rightarrow \texttt{int}) \\ \texttt{couleur} * \texttt{int} \rightarrow \texttt{couleur} \\ \vdots \end{cases}$$

Plutôt que de choisir pour t_x et t_y, des types éffectifs quelconques, mais arbitraires, Ocaml introduit dans le langage, les notions de *variables de types* et de *types paramètrés par des variables de types*. Ainsi, le type donné à `premier` contiendra des variables de type (pou t_x et t_y), notées `'a` et `'b`:

```
premier :  'a * 'b -> 'a
```

□

9.1 Polymorphisme paramétrique ou universel

Le type donné à `premier` est *paramétré par les variables* `'a` et `'b`. On peut le lire:

pour n'importe quels types `'a` *et* `'b`, $'\texttt{a} *' \texttt{b} \rightarrow '\texttt{a}$

Mais il n'existe pas des types éffectifs dont les noms soient `'a` et `'b`. Donc, lors du typage d'une application de `premier`, on doit spécialiser ces deux variables de manière à obtenir un type adapté aux contraintes imposées par le contexte. Par exemple, dans

```
premier (1,true)
```

l'argument passé est de type `int * bool`. Ocaml spécialise alors le type de `premier` par:

$$\left.\begin{array}{r} '\texttt{a} \leftarrow \texttt{int} \\ '\texttt{b} \leftarrow \texttt{bool} \end{array}\right\} \Rightarrow \texttt{premier} : \texttt{int} * \texttt{bool} \rightarrow \texttt{int}$$

Par conséquent, le type obtenu pour `premier (1,true)` est `int`, qui est bien le type de la première composante de la paire passée en argument.
□

Le type donné par Ocaml à la fonction `premier` est dit *polymorphe paramétrique ou ouniversel*:

- Il est *polymorphe* car cette fonction a plus d'un type, en l'occurrence, une infinité de types: tous ceux obtenus par spécialisation des variables de type `'a` et `'b` par un type quelconque.
- Il est *paramétrique ou universel*, car la multiplicité des types est obtenue par la substitution des variables des types – vues comme des paramètres – par tout type (universalité) qui s'adapte aux besoins du typage.

Un type polymorphe n'est pas un type "passe-partout". Par exemple, les utilisations suivantes de `premier` sont mal typées:

```
# let premier (x,y) = x;;
val premier : 'a * 'b -> 'a = <fun>

# premier 1;;
This expression has type int but is here used with type 'a * 'b

# (premier(true,2)) + 3;;
This expression has type bool but is here used with type int

# premier (1,2,3);;
This expression has type int * int * int but is here used with type 'a * 'b
```

Exemples de constructions polymorphes prédéfinies:

```
# [];;
- : 'a list = []

# [1;2;3]@[];;
- : int list = [1; 2; 3]

# ["cou"; "cou"] @ [];;
- : string list = ["cou"; "cou"]

# fst;;
- : 'a * 'b -> 'a = <fun>

# snd;;
- : 'a * 'b -> 'b = <fun>

# (=);;
- : 'a -> 'a -> bool = <fun>

# (>);;
- : 'a -> 'a -> bool = <fun>
```

Exemple de fonction très générale: Grâce au polymorphisme, il est possible de définir des fonctions très générales et utiles:

```
# let rec map f l   match l with
     []       -> []
   | a::reste -> (f(a)) :: map f reste;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>


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

```
# map plus_un [1;2;3];;
- : int list = [2; 3; 4]

# map String.length ["Bonjour"; "salut"; "ab"];;
- : int list = [7; 5; 2]

# map String.length [1;2;3];;
                     ^^
This expression has type int but is here used with type string
```

10 Traits impératifs

Ocaml possède plusieurs constructions permettant la programmation impérative: variables et structures des donneées modifiables, affectation, instructions d'entrée/sortie, boucles itératives. Les *identificateurs modifiables* sont appelés références, et l'on peut définir des structures des données modifiables telles que les tableaux (type `array`), ou les enregistrements avec champs modifiables.

10.1 Les séquences d'instructions

Jusqu'à présent, les exemples présentés mettent en oeuvre des calculs à l'aide d'expressions. Par exemple,

```
# let x = 3 in x + 4;;

# let f (x) = x + 1 in f(2);;
```

sont deux expressions désignant des valeurs (de type entier). Leur formulation est très proche des mathématiques, et de manière tout à fait naturelle, leur exécution aboutit à une valeur *rendue en résultat*.

Une autre manière d'exprimer les calculs, plus proche des langages classiques, consiste à exécuter des instructions d'entrées/sorties, d'affectation ou de modification de la mémoire. Ces actions sont appélés *effets*: elles n'ont pas de valeur résultat associé, mais elles ont pour effet, de *changer l'état* des entrées/sorties ou de la mémoire.

```
# print_string"Bonjour";;
Bonjour- : unit = ()
```

En Ocaml, *tout est expression*, et donc, tout doit avoir un résultat associé. Le résultat d'un effet est la *non valeur*, notée `()` et pronocée "unit". Son type est `unit`. En programmation impérative, on fait suivre plusieurs instructions. Une *séquence d'instructions* est une suite d'instructions séparées par des point-virgules. Les instructions sont alors exécutées en séquence.

```
#  print_string"Bonjour"; print_newline(); print_string"Ciao";;
Bonjour
Ciao- : unit = ()
```

10.2 Les références

Les définitions d'identificateurs considérées jusqu'ici correspondent à l'introduction de constantes. La déclaration `let x` $= v$, pour une valeur v, introduit un nouvel identificateur `x` lié de manière constante à la valeur v dans le reste du programme. Aucune modification de la valeur de `x` n'est posible. Tout au plus, on pourra introduire une nouvelle déclaration pour `x`, qui masque la première, en lui donnant une autre valeur. La nouvelle liaison masque la première, mais ne la change pas.

Une référence est une cellule modifiable en place. Plus précisement, il s'agit d'un identificateur lié à une adresse mémoire, dont le contenu est une valeur v du langage. La syntaxe `ref`(a) permet de construire une nouvelle référence. Elle a pour effet d'allouer un nouvel emplacement mémoire, en y mettant la valeur référencée a. Le résultat renvoyé par `ref`(a) est l'adresse de l'emplacement alloué. L'accès indirecte au contenu de r se fait par !r, qui renvoie la valeur référencée. L'affectation utilise la syntaxe habituelle. Elle renvoie en résultat la valeur des effets: `()` de type `unit`.

```
#let mavariable = ref 0;;
val mavariable : int ref = {contents=0}

# mavariable :=5;;
- : unit = ()

#  !mavariable;;
- : int = 5

# let compteur = let r = ref(0) in
    fun () -> r:= !r+1; !r;;
val compteur : unit -> int = <fun>

#  compteur();;
- : int = 1
#  compteur();;
- : int = 2
```

10.3 Les enregistrements modifiables

Un enregistrement peut être déclaré de manière à rendre (explicitement) certains champs modifiables. Cela est fait en faisant précéder le nom d'un champ par le mot-clé `mutable`:

```
# type point_mutable = {mutable x: float;mutable y: float};;
type point_mutable = { mutable x: float; mutable y: float }

# let translate p dx dy  p.x <-p.x +. dx; p.y <- p.y +. dy;;
val translate : point_mutable -> float -> float -> unit = <fun>

#let monpoint = {x=0.0;y=0.0};;
val monpoint : point_mutable = {x=0; y=0}

#translate monpoint 1.0 2.0;;
  - : unit = ()

# monpoint;;
- : point_mutable = {x=1; y=2}
```

10.4 Les tableaux

Les tableaux sont des structures des données mutables. Comme les listes, ils sont polymorphes: le type d'un tableau dont les éléments ont le type `t` est noté: `t array`. La syntaxe des tableaux constants est similaire à celle des listes, avec des barres `|` en plus. L'affectation d'une cellule de tableau se fait via la flèche inversée `<-`. Les indices sont des entiers, dont le premier est zéro.

```
# let a =  [| 0; 1; 3|]  ;;
val a : int array = [|0; 1; 3|]
```

```
# a.(0)<-7;;
- : unit = ()

# a;;
- : int array = [|7; 1; 3|]
```

Les opérations suivantes sont prédéfinies (dans le module `Array`) sur les tableaux:

- `Array.lenght`,
- `Array.create long init`, crée un tableau de taille `long` initialisé à `init` partout.

10.5 Les boucles

Boucle `For`:

La boucle `for` permet de rpéter une série d'instructions un nombre de fois fixé à l'avance.

```
# let digits = Array.create 10 0;;
val digits : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]

#  for i = 0 to 9 do digits.(i) <- digits.(i) + i done;;
- : unit = ()

#for i = 0 to 10 do print_int i done;;
012345678910- : unit = ()

# for i = 10 downto 0 do print_int i done;;
109876543210- : unit = ()

# let fact (n) =
  let accu = ref 1 in
  for i = 1 to n do accu := i * !accu done;
  !accu;;
val fact : int -> int = <fun>

# fact(2);;
- : int = 2

# fact(3);;
- : int = 6
```

Voici un exemple qui utilise des tableaux et la boucle `for`:

```
# let add_vect v1 v2 =
   let len = min (Array.length v1) (Array.length v2) in
   let res = Array.create len 0.0 in
   for i = 0 to len - 1 do
     res.(i) <- v1.(i) +. v2.(i)
   done;
   res;;
val add_vect : float array -> float array -> float array = <fun>

#add_vect [| 1.0; 2.0 |] [| 3.0; 4.0 |];;
- : float array = [|4.; 6.|]
```

Boucle `While`:

La boucle `while` permet de rpéter une série d'instructions tant qu'une condition est vraie.

```
#let  j =ref 10 in
 while (!j)> 0 do begin print_int !j; j := !j-1; end done;;
10987654321- : unit = ()

# j;;
Characters 0-1:
Unbound value j
```

Notez que j est une variable locale à l'expresion suivant le `in`, et de ce fait, elle est n'est plus definie en dehors de celle-ci.

```
#let insertion_sort a =
   for i = 1 to Array.length a - 1 do
     let val_i = a.(i) in
     let j = ref i in
     while !j > 0 && val_i < a.(!j - 1) do
       a.(!j) <- a.(!j - 1);
       j := !j - 1
     done;
     a.(!j) <- val_i
   done;;
val insertion_sort : 'a array -> unit = <fun>
```