

Premiers pas avec Objective-Caml

par Antonio Pasinelli

Date de publication : 01/11/2006

Dernière mise à jour : 29/08/2007

Nous remercions Michel Mauny pour avoir relu et corrigé une partie de cet article.

I - Introduction

I-1 - Un peu d'histoire générale sur les langages fonctionnels

I-2 - Pourquoi choisir un langage fonctionnel ?

I-3 - Objective Caml

II - Premier contact

II-1 - Installation

II-2 - La boucle d'interaction

II-3 - Premières fonctions, notions de syntaxe Objective Caml

I - Introduction

I-1 - Un peu d'histoire générale sur les langages fonctionnels

L'histoire des langages fonctionnels remonte presque aussi loin dans le temps que celle des tous premiers langages de programmation largement diffusés, comme Fortran I ou APL (A Programming Language), le C étant relégué au stade de new-be... autant dire que la descendance est large et parfois même assez complexe, aujourd'hui !

L'histoire de l'informatique retiendra essentiellement quelques langages fonctionnels : Lisp (1958), ML (1970) et Haskell (1990 - héritier de Miranda datant de 1985, lui-même héritier de KRC en 1981 et SASL en 1976). Il s'agit en effet de deux disciplines de programmation entièrement différentes, bien que possédant des points communs.

L'idée de Lisp était de proposer un langage de programmation permettant de manipuler des listes, structures dynamiques, donnant une très grande souplesse au moment de la conception des programmes, par rapport à ce qui se faisait dans d'autres langages comme les premiers Fortran (tous jusqu'au Fortran 77) qui adoptaient une vision monolithique et statique des structures de données. Le résultat, bien que particulièrement intéressant à l'époque en termes d'abstraction, ne pouvait cependant pas concurrencer directement ce que l'on obtenait grâce aux langages plus classiques, notamment du fait de la forte consommation en termes de mémoire et de temps de calcul des programmes... à une époque où posséder 500 Ko de mémoire vive et une puissance de calcul de 1 MHz était déjà une folie !

ML a suivi cette idée en ajoutant une discipline de typage.

Miranda, moins diffusé mais tout aussi important dans la genèse des langages fonctionnels, proposait une méthode d'évaluation paresseuse d'expressions construites à l'aide d'équations mathématiques. Les expressions, à l'inverse de Lisp, n'étaient pas évaluées lors de l'appel de fonctions, mais uniquement lorsque le corps de la fonction faisait explicitement référence à cette expression, lors de l'exécution du programme. En 1987, Miranda périclita, et a donné naissance à un nouveau langage, beaucoup plus puissant et reprenant tous ses principaux traits : Haskell, qui est aujourd'hui standardisé (Haskell 98) et en très forte expansion y compris dans le monde industriel.

Fondamentalement, on peut ainsi parler de deux branches importantes : les langages dérivés de Lisp (dont Objective Caml, Scheme et Standard ML), et ceux ayant poursuivi dans la voie Miranda (essentiellement le langage Haskell, aujourd'hui). Chaque représentant de chacune des deux branches reprend dans son ensemble à peu près tous les concepts importants des ses ancêtres, en y ajoutant le plus souvent des fonctionnalités tirées des langages impératifs ou orientés objet (comme le fait Objective Caml).

On peut aussi citer quelques langages fonctionnels plus récents, moins classiques mais tout aussi intéressants, puissants et performants : Clean, Erlang (développé par Ericsson pour le parallélisme et la communication) ou encore SCOL.

I-2 - Pourquoi choisir un langage fonctionnel ?

On peut affirmer sans trop se tromper que les langages fonctionnels ont mauvaise réputation dans le domaine industriel, bien que les choses sont en train de changer progressivement et dans le bon sens ! Ceci est essentiellement dû au fait que les premières implantations de langages fonctionnels utilisaient des méthodes lourdes à l'exécution, faisant ainsi des programmes écrits dans ces langages des "machines à faire ramer l'ordinateur"... Une deuxième raison peut se trouver dans le fait qu'il est nécessaire d'adopter une attitude différente lorsque l'on aborde la programmation fonctionnelle : certains langages étant purs et ne possédant pas de notions de boucles (boucles for ou while), il n'est pas rare de trouver des personnes complètement désorientées par le système, lâchant tout de suite l'affaire.

Du côté des performances, les choses ont bien changé avec l'apparition des compilateurs natifs : dès lors, il arrive que des programmes écrits en Haskell, en Objective Caml ou en Clean rivalisent en termes de performances des programmes faits en C, ADA ou Fortran !

Du côté de l'apprentissage des langages fonctionnels, on peut remarquer à juste titre que le développement d'internet a permis une divulgation massive des connaissances et l'on voit fleurir un peu partout des tutoriels comme celui-ci, ou encore des forums d'aide.

En ce qui concerne le monde académique et le monde de la recherche, il existe une véritable demande pour les langages fonctionnels ; cela mène à dire, sans trop se tromper, que les langages fonctionnels sont un excellent moyen de prototypage d'applications. A titre d'exemple, un rapport d'enquête mené à l'université de Yale aux Etats-Unis et portant sur les possibilités en termes de développement de différents langages (entre autres ADA, C ++, Ruby et Haskell) montre très clairement que l'utilisation d'un bon langage fonctionnel, dans ce cas, Haskell, permet de développer très rapidement en 80 lignes de code un application qui en C ++ en fait 1100, et avec un temps d'exécution supérieur. Il est donc intéressant d'utiliser un langage fonctionnel pour écrire assez rapidement une première version d'une application, afin d'évaluer de façon plus précise les contraintes, ce qui peut grandement améliorer le développement de l'application finale, étant donné que bon nombre de difficultés seront déjà apparues et/ou résolues lors de l'élaboration du prototype.

Il est cependant une chose qu'il faut garder à l'esprit : chaque langage de programmation possédant son propre mode de fonctionnement, il est inutile de vouloir plaquer sur un code Haskell, par exemple, un schéma Fortran, ADA ou Java. La programmation fonctionnelle est avant tout un style de programmation privilégiant l'abstraction et il est important, dès lors, de prendre un peu de recul par rapport au problème et à l'algorithme à implanter afin de pouvoir écrire le code facilement. Les compilateurs de langages fonctionnels ont aussi, pour la plupart, une notion assez fine de la sémantique d'un programme, de sorte qu'il est parfois inutile de croire que prendre de la hauteur au niveau du code se traduira inexorablement par une perte sèche en termes de performances.

I-3 - Objective Caml

Objective Caml fait partie de la branche Caml de la famille ML (Meta Language), l'une des familles des descendants de Lisp. La première version de ML était dédiée à programmer la recherche automatique de preuves dans un système d'aide à la démonstration (de théorèmes mathématiques) du système LCF (Logic for Computable Functions). ML est ensuite apparu comme un langage généraliste, et a été étendu en ce sens, indépendamment de LCF. Il est aujourd'hui le plus important représentant. Parallèlement à la branche Caml on trouve la branche SML (Standard ML) avec SML/NJ et *mossml*, essentiellement des langages anglo-saxons.

Caml est l'acronyme de *Categorical Abstract Machine Language*, qui pourrait se traduire par "langage informatique abstrait typé"... En fait, il s'agit surtout de la combinaison, proposée par Guy Cousineau, des mots CAM (Categorical Abstract Machine, une machine virtuelle issue de travaux de Pierre-Louis Curien sur la sémantique des langages fonctionnels) et ML (Meta Language).

Caml voit le jour en 1986 : il s'agit d'un projet reposant sur **Le Lisp** et développé par l'INRIA en collaboration avec l'Ecole Normale Supérieure et l'Université Paris 7, et utilisé à la base en tant que support à l'implantation d'un système de preuve de programmes, connu aujourd'hui sous le nom de **Coq**. Son nom est tiré du nom de sa machine virtuelle appelée CAM (acronyme de *Categorical Abstract Machine*). Une machine virtuelle est un programme permettant d'exécuter des instructions binaires écrites dans un format le plus souvent indépendant du système d'exploitation et du processeur utilisé. En réalité, la machine virtuelle émule complètement le fonctionnement d'un ordinateur et d'un système d'exploitation, permettant ainsi au code binaire d'être portable. Java est construit sur le même principe, d'où, entre autres, son succès.

Au début des années 1990, Xavier Leroy écrit une nouvelle machine virtuelle appelée **Zinc** (acronyme récursif de *Zinc Is Not Caml*, afin de montrer explicitement que ces deux langages partagent moins de choses qu'il n'y paraît),

autorisant une multitude d'opérations interdites avec la CAM. Ce projet donne naissance à **Caml-Light**, très utilisé malheureusement dans l'enseignement supérieur en France, puis à Caml Special-Light, la première implantation possédant un compilateur natif (c'est-à-dire produisant un code assembleur exécutable directement, sans passer par le machine virtuelle). La première version d'Objective Caml voit le jour en 1996 et apporte une couche objet au langage, lui permettant d'être extrêmement générique.

Aujourd'hui, Caml-Light et Caml Special-Light sont des projets arrêtés, la dernière mise-à-jour de Caml-Light datant de 2002 ; seul Objective Caml est activement développé par l'équipe CRISTAL de l'INRIA et disponible pour un très grand nombre d'architectures et de systèmes d'exploitation : Linux, Windows (pas moins de trois versions), Solaris, Iris, HP-Unix, FreeBSD, etc... pour les systèmes et IA32, IA64, x86_64, Dec Alpha, UltraSparc, PA-Risc, PowerPC, ARM, MIPS, etc... en ce qui concerne les architectures.

Il suffit de trois expressions pour décrire l'essence même du langage Objective Caml : "langage fonctionnel", "strict", "fortement typé".

- **langage fonctionnel** : on peut programmer sans effets de bords, c'est-à-dire sans modifier silencieusement l'environnement d'exécution du programme (une affectation ou une saisie au clavier sont des effets de bords), et les fonctions sont traitées comme des valeurs à part entière que l'on peut passer en argument, faire retourner par d'autres fonctions, etc...
- **strict** : tous les paramètres des fonctions sont évalués avant de rentrer dans le corps de la fonction en elle-même.
- **fortement typé** : l'inférence de types pour les fonctions et les expressions est faite à la compilation, de sorte que beaucoup d'erreurs pouvant apparaître dans d'autres langages lors de l'exécution sont éliminées dès la compilation. Il s'agit en effet d'un gage de sûreté de développement important.


En outre, Objective Caml est extrêmement expressif, essentiellement parce qu'il contient tous les traits des autres langages de programmation classiques :

- expressions paresseuses encapsulées dans des valeurs d'un type particulier, dont les flots de données
- boucles for et while et structures de données modifiables, autorisant ainsi les effets de bord
- une couche objet autorisant, entre autres, l'héritage multiple
- un langage de modules surpuissant permettant de créer des foncteurs, sorte de fonctions construisant un module à partir d'autres modules
- un pré-processeur, Camlp4, reléguant au stade d'antiquité celui du C ; on peut ainsi construire soi-même ses propres extensions du langage en définissant des extensions de la grammaire actuelle, ou encore se servir du pré-processeur en tant que générateur d'analyseurs syntaxiques dynamiques et reconnaissant une classe de grammaires plus large que celle reconnue par ocaml yacc (l'équivalent de Yacc en Objective Caml), c'est-à-dire permettant de faire évoluer la définition d'une grammaire au cours de l'exécution en modifiant des règles, etc... D'ailleurs, une syntaxe révisée de tout Objective Caml a été écrite grâce au pré-processeur et le pré-processeur lui-même est écrit en partie dans cette syntaxe révisée !
- machine virtuelle autorisant le chargement dynamique de code
- nombreux autres utilitaires

Ces derniers points associés à la présence d'une machine virtuelle, issue de la Zinc, munissent le langage d'un attrait indéniable : portabilité, ré-utilisabilité massive non seulement de code mais aussi de binaires, facilité de développement, etc...

Suivez-donc la voie du chameau !

 *Pour plus d'infos sur l'histoire de Caml, suivez le [lien...](#)*

 *Pour infos, il existe un langage proche d'OCaml et fonctionnant sur la machine virtuelle .Net, nommé **F#***

Suivez le [lien...](#)

II - Premier contact

II-1 - Installation

Pour installer Objective Caml, rien de plus simple si vous êtes sous Windows : il suffit de **télécharger l'application**, la copier dans son espace de travail et lancer le programme d'installation, par exemple en double-cliquant sur l'icône apparaissant dans l'explorateur de dossiers. Des indications devraient apparaître ainsi dans une boîte de dialogue : laissez-vous guider.

Notez bien qu'il existe plusieurs ports d'Objective Caml pour Windows : si vous ne savez pas et s'il s'agit de la première fois que vous l'installez, choisissez la version standard Win32. Certaines fonctionnalités, telles la possibilité d'utiliser le compilateur natif, ne sont pas disponibles, à moins de disposer de Visual C++, mais il est tout de même possible de créer des programmes binaires s'exécutant sans machine virtuelle préalablement installée, via **ocamlc** (voir l'option `-custom`). C'est une bonne distribution pour se faire la main avec le système Objective Caml.

Sous Linux, et uniquement pour les architectures du type x86, plusieurs **packages** existent, et donc leur installation ne devrait pas poser de problèmes (encore plus simple que sous Windows, hein !). Bien-sûr, il est toujours possible d'installer le tout à partir des sources !

Sous Unix, le meilleur moyen est d'installer la distribution à partir des sources. Les fichiers `INSTALL` et `README` que l'on trouve à la racine décrivent très bien les différentes étapes de l'installation ainsi que les outils nécessaires. Remarquez qu'il est hautement préférable avoir un système disposant des outils GNU afin d'exploiter au mieux certaines optimisations du code de la machine virtuelle, écrite en C, entre autres. Seules une commande **`./configure`** et quelques appels à **`make`** devraient suffir pour installer le tout. Veillez bien à passer par toutes les étapes de l'installation, à moins que vous ne vouliez une installation un peu particulière : l'installation consiste, entre autres, au boot-strap des compilateurs et de tout le système. Cela prend un certain temps, mais si toutes les phases ne sont pas exécutées, le système ne sera pas complet. Il peut aussi être intéressant de jouer sur les différentes options du script `./configure`, entre autres pour décider du compilateur C utilisé et du dossier cible.

En bref, une installation complète Objective Caml se compose des éléments suivants :

- `ocaml` : la boucle d'interaction
- `ocamlc` : le compilateur code objet
- `ocamlopt` : le compilateur natif
- `ocamlrun` : la machine virtuelle
- `ocamllex` et `ocamlyacc` : les générateurs respectivement d'analyseurs lexicaux et syntaxiques
- `ocamldep` : le calculateur de dépendances entre unités de compilation
- `ocamlbrowser` : un petit environnement de développement dédié à Objective Caml
- `ocaml-doc` : un générateur de documentation
- `ocamldebug` : un débogueur puissant
- `ocamlprof` : un profiler afin de d'analyser les temps d'exécution de chaque partie du programme
- `camlp4` : une librairie permettant de traiter des syntaxes mais aussi le puissant pré-processeur d'Objective Caml
- Pervasives : la librairie de base
- la librairie standard comportant une quarantaine de modules
- plusieurs librairies annexes, qui peuvent, selon les systèmes et les installations, ne pas être présentes :
 - Unix : programmation système portable !

- Num : nombres de précision arbitraire
- Str : expressions rationnelles
- Threads
- Graphics
- Dbm : bases de données NDBM
- Dynlink : chargement dynamique de code dans la machine virtuelle
- LablTk : interface graphique Tcl/Tk
- Bigarray : grandes matrices et grands vecteurs

Les noms des binaires se terminant par `.opt` correspondent aux versions compilées avec le compilateur natif, plus performantes. Les distributions binaires d'Objective Caml, en général, ne distribuent que ces dernières. Certaines distributions n'incluent pas tous les programmes : se reporter donc à la documentation fournie.

II-2 - La boucle d'interaction

La boucle d'interaction d'Objective Caml est souvent, à tort, considérée comme le coeur du langage. En réalité, il ne s'agit que d'un interprète de commandes qui se contente d'évaluer les expressions entrées au clavier, comme ce que l'on peut trouver en MATLAB, Haskell avec Hugs et GHCi, MAPLE ou Scilab. L'utilisation la plus courante est lors de la phase de développement d'un projet, afin de tester ponctuellement certaines fonctions ou fonctionnalités. Cependant, elle permet également d'apprendre le langage en toute tranquillité, car elle fournit un environnement sûr, tolérant vis-à-vis des erreurs, chose qui n'existe pas en C ou en Java, par exemple.

Pour lancer la boucle d'interaction sous Windows, cliquez sur le chameau dans l'onglet Objective Caml du menu Démarrer, ou sur l'icône de bureau si vous en avez installé une. Sous Linux et Unix, entrez la commande **ocaml**. Que ce soit sous Windows ou sous Linux et Unix, une chose du genre de ce qui suit devrait alors apparaître à l'écran.

```
[InOCamlWeTrust @ Sahara Documents]$ ocaml
Objective Caml version 3.09.2

#
```

Le symbole `#` indique le prompt, c'est-à-dire l'espace où l'utilisateur pourra entrer une expression à évaluer. Un session sous la boucle d'interaction est donc constituée d'une suite d'expressions que l'on cherche à évaluer... voire à exécuter. On peut bien-sûr utiliser des résultats précédemment calculés, du moment qu'on les a liés à un identificateur au moyen d'une expression de la forme **let ident = exp;;**. On peut également saisir des directives, pour par exemple exécuter un script contenu dans un fichier, changer de dossier de travail ou tout simplement sortir de la boucle d'interaction. Un exemple de session est fourni ci-après.

```
[InOCamlWeTrust @ Sahara Documents]$ ocaml
Objective Caml version 3.09.2

# 5;;
- : int = 5
# let a = 7;;
val a : int = 7
# type t = int;;
type t = int
# print_endline "Hey ! The Caml is looking at You !";;
Hey ! The Caml is looking at You !
- : unit = ()
# #quit;;
[InOCamlWeTrust @ Sahara Documents]$
```

Sous la boucle d'interaction, les expressions à évaluer se terminent toujours par ;;. Si ;; est omis, un retour à la ligne s'effectuera et l'utilisateur pourra saisir la suite de son expression sur la ligne suivante. La séquence ;; marque la fin de l'expression et donc le début de l'évaluation.

```
[InOCamlWeTrust @ Sahara Documents]$ ocaml
Objective Caml version 3.09.2

# let b = true;;
val b : bool = true
# let a =
  if b || false then
    5
  else
    1
  ;;
val a : int = 5
# #quit;;
[InOCamlWeTrust @ Sahara Documents]$
```

Des informations de contexte sont affichées à gauche du résultat. Dans l'exemple ci-avant, on peut donc voir que **b** est du type **bool** et qu'il s'agit d'une valeur, comme spécifié par le mot-clef **val**, par opposition au type **t** défini dans l'exemple précédent, pour lequel la boucle d'interaction affiche le fait qu'il s'agit bel et bien d'une déclaration de type, indiqué par le mot-clef **type**, ici un alias pour le type des entiers standards.

Lorsque l'évaluation n'est associée à aucun identificateur, donc lorsque le résultat n'est pas conservé dans le contexte d'exécution courant, la boucle affiche un tiret - pour indiquer l'absence d'une telle association. L'évaluation de l'entier 5 en est un exemple, tout comme l'affichage du message "Hey ! The Caml is looking at You !". A ce titre, on remarquera qu'y compris le fait d'afficher un message à l'écran, par exemple, est une valeur, munie d'un type : le type **unit** dont toute valeur se réduit, après évaluation, à la valeur dénotée par **()**. Ce point sera abordé plus loin dans le tutoriel.

Ecrire tout un programme dans une boucle d'interaction se révèle très vite être une très mauvaise méthode. Il vaut mieux, au début, faire un script avec l'ensemble des expressions que l'on souhaite utiliser, charger ce script, puis faire appel à ces valeurs depuis la boucle d'interaction. Un script s'écrit de la même façon que l'on rentre des expressions dans la ligne de commande : la fin de l'évaluation d'une expression est repérée par ;;. Lorsqu'un script est chargé dans la boucle d'interaction, toutes les valeurs sont évaluées, et le résultat est affiché sur le terminal, comme si on les avait rentrées une à une, à la main, sans pour autant que leur définition, c'est-à-dire le code, ne soit visible à l'écran. Voici un exemple de script : il reprend les expressions du premier exemple.

```
(* Cette ligne est un commentaire Objective Caml et n'est pas prise en compte *)

(* Remarque : les commentaires peuvent s'imbriquer (* les uns dans les autres *) *)

5
;;

let a = 7
;;

type t = int
;;

print_endline "Hey ! The Caml is looking at You !"
;;
```

Après avoir chargé le précédent script (de nom "script.ml" ici) dans la boucle d'interaction, on peut voir apparaître un écran du type suivant.

```
[administrateur@localhost Tutoriel]$ ocaml
Objective Caml version 3.09.2


# #use "script.ml";;
- : int = 5
val a : int = 7
type t = int
Hey ! The Caml is looking at You !
- : unit = ()
#
```

La boucle d'interaction affiche alors les valeurs présentes dans l'environnement de travail : il s'agit ici de **a** et du type **t**, donc les valeurs avec lesquelles on peut poursuivre la session.

On rappelle enfin deux directives utiles : **#use "nom_de_fichier.ml"** pour charger un script, et **#quit** pour quitter.

II-3 - Premières fonctions, notions de syntaxe Objective Caml

Après avoir fait un tout petit tour d'horizon de la boucle d'interaction, il est bon d'avoir quelques rudiments de syntaxe Objective Caml. Remarquons une chose : la syntaxe Objective Caml est très particulière, et demande l'usage de quelques parenthèses (mais pas beaucoup, en fait).

 *Tout, absolument tout est une valeur en Objective Caml... sauf les types, bien-sûr !*

Une fonction, par exemple, est une valeur, tout comme un entier ou le simple fait d'afficher une chaîne de caractères à l'écran : la fonction d'affichage est une valeur, la chaîne de caractères en est une deuxième et le résultat, c'est-à-dire l'affichage, en est une troisième ! Ceci permet beaucoup de souplesse au moment d'écrire du code, car le niveau d'abstraction est assez élevé.

Les entiers sont de type **int**, les flottants de type **float**, les caractères de type **char** et les chaînes de caractères de type **string**. Tous ces types de valeurs s'écrivent de façon classique.

```
# 666;;
- : int = 666
# 3.1416;;
- : float = 3.1416
# 'a';;
- : char = 'a'
# "Hardware tabulations rule !!!";;
- : string = "Hardware tabulations rule !!!"
# print_endline "\\t\\t\\t rules !!!";;
\\t\\t\\t rules !!!
- : unit = ()
#
```

Les appels de fonctions se font sans les parenthèses, comme dans l'exemple ci-dessus avec la fonction **print_endline** permettant d'afficher une chaîne de caractères avec un retour à la ligne en fin de chaîne. C'est l'une des particularités des langages fonctionnels issus de la théorie du lambda-calcul : une telle notation permet, en effet, de créer des applications partielles facilement, comme on le verra dans la suite du tutoriel.

Pour nommer une valeur, on utilise la construction **let ident = exp in exp'**. La valeur **exp** est alors repérée par le nom **ident** dans la sous-expression **exp'**.

⚠ Notez bien qu'il ne s'agit en aucun cas de la déclaration d'une variable : la valeur **ident** n'est pas modifiable... jamais !

C'est encore l'une des particularités des langages fonctionnels : les identificateurs repèrent des valeurs et non des variables... d'ailleurs, la notion de variable n'existe pas. Cependant, rien n'empêche au contenu de la valeur d'être modifiable ! En effet, il existe plusieurs types physiquement modifiables en Objective Caml, comme on le voit par ailleurs.

⚠ La construction **let ident = exp in exp'** est très différente de **let ident = exp;;**. La première lie **exp** à **ident** dans l'expression **exp'**, tandis que la deuxième effectue la liaison dans tout le programme. En fait, la deuxième constitue en soi une phrase Objective Caml, alors que la première n'est qu'une expression, de valeur **exp'**, de sorte qu'il est possible d'écrire des associations imbriquées.

```
# let a = 7 in
    let b = a + 8 in
        a * b
;;
- : int = 105
#
```

La valeur 105 est non liée et définie à l'aide de deux constructions **let ident = exp in exp'** imbriquées, mais on aurait également pu écrire la chose de la façon suivante, si on avait voulu en faire une valeur présente dans l'environnement d'exécution.

```
# let n =
    let a = 7 in
        let b = a + 8 in
            a * b
;;
val n : int = 105
#
```

Si on veut effectuer plusieurs associations indépendantes, on peut utiliser la construction **let ident1 = exp1 and ident2 = exp2 ... in exp** dans des valeurs, ou **let ident1 = exp1 ... and identn = expn;;** pour une phrase Objective Caml.

```
# let hello = "Hello " and ker = "Kernighan " and amp = "& " and rit = "Ritchie " and bang = "!" in
    hello ^ ker ^ amp ^ rit ^ bang
;;
- : string = "Hello Kernighan & Ritchie !"
# hello;;
Unbound value hello
# ker;;
Unbound value ker
# amp;;
Unbound value amp
# rit;;
Unbound value rit
# bang;;
Unbound value bang
# let caml = "Caml" and haskell = "Haskell";;
val caml : string = "Caml"
val haskell : string = "Haskell"
# caml ^ " & " ^ haskell;;
- : string = "Caml & Haskell"
#
```


On aura remarqué que les valeurs **hello**, **ker**, **amp**, **rit** et **bang** sont uniquement visibles depuis l'expression **hello ^ ker ^ amp ^ rit ^ bang**, et ne sont pas utilisables par la suite, contrairement aux valeurs **caml** et **haskell**. L'opérateur **^** permet de concaténer deux chaînes de caractères.

On peut écrire des expressions conditionnelles grâce à la construction **if cond then exp else exp'**, où **cond** est nécessairement du type **bool**. Les opérateurs booléens sont **||** pour le OU logique, **&&** pour le ET logique et **not** pour le non logique. Les deux valeurs booléennes sont **true** et **false**. Les opérateurs de comparaison sont ceux utilisés habituellement : **=**, **<**, etc...

```
# let v = true and f = false;;
val v : bool = true
val f : bool = false
# if v || f then
    'V'
    else
        'F'
;;
- : char = 'V'
# let c =
    if v && f then
        'V'
    else
        'F'
;;
val c : char = 'F'
#
```

On peut déclarer une fonction de la façon suivante : **let ident arg1 ... argn = exp;;** si on veut rendre la fonction **ident** visible globalement ou **let ident arg1 ... argn = exp in exp'** si on veut qu'elle ne soit accessible uniquement depuis l'expression **exp'**.

```
# let est_pair n =
    if (n mod 2) = 0 then
        true
    else
        false
;;
val est_pair : int -> bool = <fun>
# let somme a b = a + b in
    somme 5 6
;;
- : int = 11
#
```

 *Toutes les expressions en Objective Caml sont typées et il n'existe pas de conversion implicite. De ce fait, tous les opérateurs arithmétiques existent pour les valeurs de type **int** et de type **float** : il suffit de les suffixer avec un point ., par exemple + et +., - et -., * et *., etc...*

On peut néanmoins utiliser les deux fonctions de conversions prédéfinies suivantes.

```
# float_of_int;;
- : int -> float = <fun>
# int_of_float;;
- : float -> int = <fun>
#
```

Pour le système d'inférence de types, les opérateurs sont également des fonctions, et il est donc possible de les redéfinir. En effet, si on ne peut modifier une valeur, il est toutefois possible de créer une nouvelle valeur de même nom, auquel cas l'ancienne valeur est perdue, même si cela est très fortement déconseillé ! Un exemple amusant est celui-ci...

```
# 4 - 2;;
- : int = 2
# 4 * 2;;
- : int = 8
# 4 / 2;;
- : int = 2
# let ( - ) = ( + ) and ( * ) = ( + ) and ( / ) = ( + );;
val ( - ) : int -> int -> int = <fun>
val ( * ) : int -> int -> int = <fun>
val ( / ) : int -> int -> int = <fun>
# 4 - 2;;
- : int = 6
# 4 * 2;;
- : int = 6
# 4 / 2;;
- : int = 6
# #quit;;
```

Les fonctions ont également un type : le type de chaque argument apparaît devant une flèche, et le dernier type est celui de la valeur de retour de la fonction.

```
# let somme_mixte a x b = float_of_int (a + (int_of_float x) + b)
;;
val somme_mixte : int -> float -> int -> float = <fun>
#
```

a et **b** sont des entiers, **x** et la valeur de retour sont des flottants.

On peut également définir des fonctions récursives. Le mot-clef **rec** doit apparaître après le **let**.

```
# let rec fact n =
    if n = 0 then
        1
    else
        n * (fact (n - 1))
;;
val fact : int -> int = <fun>
# fact 2;;
- : int = 2
# fact 4;;
- : int = 24
# fact 5;;
- : int = 120
#
```

Les entiers sont de largeur fixe (31 bits sur les machines 32 bits et 63 bits sur les processeurs 64 bits... le bit manquant étant dû au garbage collector), de sorte que des problèmes de débordement peuvent être visibles très tôt avec une telle fonction.

```
# fact 15;;
- : int = -143173632
# fact 16;;
```

```
- : int = -143294464
# fact 17;;
- : int = -288522240
#
```

Les fonctions peuvent aussi être mutuellement récursives : on emploie alors les constructions **let rec ident1 arg1 ... = exp1 and ident2 arg2 ... = exp2 ... in exp'** ou **let rec ident1 arg1 ... = exp1 and ident2 arg2 ... = exp2 ... ;;**.

Il existe aussi la syntaxe adaptée à la manipulation des structures impératives : boucles, affectations, etc... Ce point est abordé plus en avant, mais on donne ici deux exemples.

```
# let n = ref 0;;
val n : int ref = {contents = 0}
# for i = 1 to 10 do
  (print_int i;
   print_newline ());
  n := !n + i)
done
;;
1
2
3
4
5
6
7
8
9
10
- : unit = ()
#
```

Les références, telles que **n**, sont modifiables grâce à l'opérateur **:=**, et la valeur d'une référence est obtenue grâce à l'opérateur **!**. Les instructions sont les valeurs de type **unit** et peuvent s'enchaîner avec le point-virgule **;**. L'indice d'une boucle **for** n'est jamais modifiable.

On peut également utiliser des boucles conditionnelles.

```
# let n = ref 10;;
val n : int ref = {contents = 10}
# while !n > 0 do
  (print_int !n;
   n := !n - 1;
   print_newline ())
done
;;
10
9
8
7
6
5
4
3
2
1
- : unit = ()
#
```

Comme dit plus haut, tout ceci est abordé dans la suite. On en donne ici deux exemples afin de montrer très explicitement que **Objective Caml n'est pas un langage de programmation entièrement dédié à la récursivité.**

