TP1 – Les langages de programmation

À la découverte de Objective CAML

UV IDL

1 Introduction

Objective CAML, appelé ensuite ocaml, est un langage fonctionnel. Autrement dit, les fonctions sont des valeurs de bases du langage. Elles peuvent donc être manipulées (passées en paramètres à d'autres fonctions ou retournées comme résultat d'un appel de fonction). C'est un langage :

- typé statiquement par inférence : l'utilisation de valeurs et variables est vérifiée au moment de la compilation ; l'information de type n'a pas besoin d'être fournie par le programmeur, elle est calculée par le compilateur.
- offrant du polymorphisme paramétrique : si une fonction n'explore pas la totalité de la structure d'un de ses arguments, elle a un type non entièrement déterminé (un type variable).
- dont les allocations et désallocation de données en mémoire sont gérées de manière automatique par un ramasse miette.
- possédant des traits impératifs : il est possible d'utiliser des structures de contrôle impératives et de manipuler physiquement certaines valeurs (tableaux, références, ...).
- qui intègre un mécanisme d'exceptions.
- qui dispose d'un interpréteur (`ocaml`), d'une machine virtuelle (`ocamlrun`), de deux compilateurs (un pour la machine virtuelle `ocamlc` et un natif `ocamlopt`) et de nombreux outils de développement (trace d'exécution, calcul des dépendances, analyse de performance, ...).

Un programme ocaml peut être structuré sous deux formes : par modules ou suivant le modèle objet. Le choix entre ces deux modèles de structuration offre une grande souplesse au langage. Ils sont duaux les modules facilitant l'extension des traitements et les objets facilitant l'extension des données. Dans le cadre du cours, nous n'explorerons pas les aspects objet du langage. Le lecteur intéressé par cet aspect est renvoyé à la lecture des documents référencés ci-dessous.

En ocaml, un module est une unité de compilation qui regroupe des données et du code qui sont décrits dans une interfaces. Le langage intègre de multiples notions permettant une manipulation avancée des modules (modules paramétrés, foncteurs, ...). Dans la mise en pratique, nous nous

limiterons à utiliser des modules simples réalisés par des fichiers séparés. Le TP décrira un peu plus en détail la manière de construire un module dans la section B.2 page 9.

Le cours associé a présenté principalement les concepts fonctionnels que vous ne connaissez sans doute que très peu. Ce sujet de TP a pour but de vous faire pratiquer la partie fonctionnelle. En annexe A page 5, vous trouverez une version un peu rédigée de ce qui a été vu en cours sur ocaml. Pour découvrir les objets et la parties impératives d'ocaml, vous pouvez consulter :

- Le manuel `http://caml.inria.fr/pub/docs/manual-ocaml`
- Le portail `http://ocaml.org`
- Une page qui contient de nombreux liens sur des livres sur ocaml : `http://ocaml.org/books.html`. Parmi ces liens, je conseille particulièrement les suivants :
  - Un livre récent en anglais pragmatique et complet : `https://realworldocaml.org`
  - Un livre sur ocaml : `http://archives.pps.univ-paris-diderot.fr/Livres/ora/DA-OCAML/index.html`

Nous utiliserons également `opam` le gestionnaire de paquet ocaml (`https://opam.ocaml.org/`). Cet outil permet le téléchargement, la compilation et l'installation de bibliothèques pour ocaml. Cet outil installe les logiciels dans un repertoire `.opam`[1] de votre repertoire racine et ajoute l'accès à ces bibliothèques dans votre environnement[2].

Le compilateur `ocaml` ainsi qu'`opam` sont disponibles dans l'environnement des salles de TP.

2 Une introduction

Dans cette section, nous conseillons d'utiliser l'interpréteur (appelé parfois boucle d'interaction).

2.1 L'interpréteur

L'interpréteur d'ocaml est lancé par la commande `ocaml`. Nous allons utiliser une version plus conviviale de cet interpréteur : `utop`[3]. Il faut donc commencer par l'installer `opam install utop`.

La figure 1 contient une capture d'écran d'`utop` lors de son lancement. Le *prompt* de l'interpréteur est le caractère `#`. Une fois ce caractère affiché, l'interpréteur lit les données entrées aux claviers jusqu'à rencontrer `;;`. Les phrases ainsi entrées peuvent être des expressions (qui donnerons un résultat) ou des définitions qui introduiront de nouvelles variables (instruction `let`). L'interpréteur répond à une expression en affichant le résultat de son évaluation précédé de son type et à une définition en donnant le type de la valeur définit et éventuellement sa valeur. Par exemple :

```
#1+2*3;;
- : int = 7
#let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265358979312
#let square x = x *. x;;
val square : float -> float = <fun>
#square(sin pi) +. square(cos pi);;
- : float = 1.
```

1. Avant la première utilisation d'`opam`, il faut exécuter `opam init` qui a en charge de créer le repertoire `.opam`.
2. La commande `eval ‘opam config env‘` se charge de cette initialisation. Si vous souhaitez l'exécuter pour chaque nouveau terminal, mettez-là dans votre `.profile.perso`.
3. `https://github.com/diml/utop`

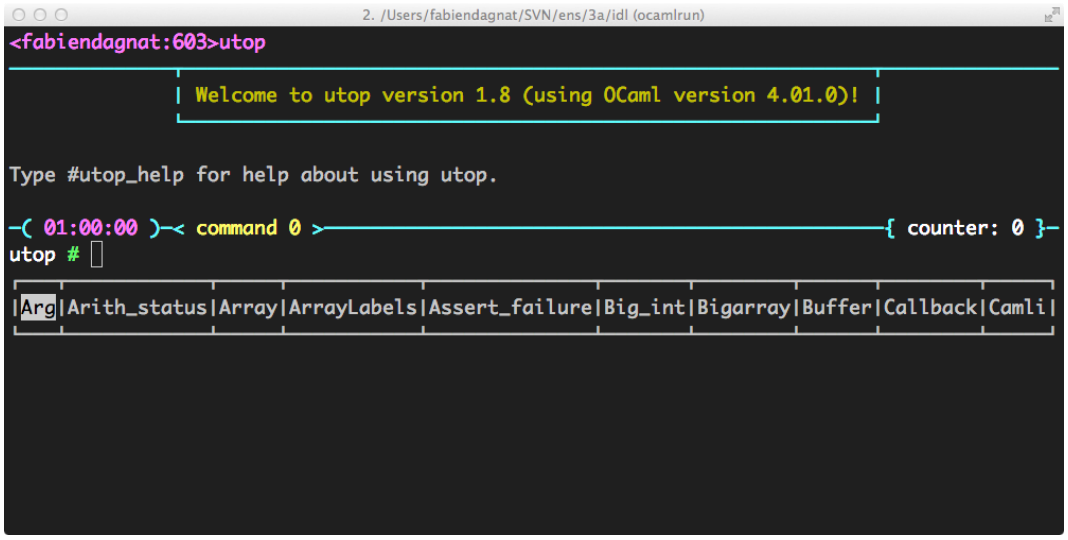

FIGURE 1 – L'interpréteur `utop`

Des *directives* permettent d'interagir avec la boucle d'interaction. Elles sont distinguées des expressions ocaml car elles commencent par un dièse `#`. Par exemple, pour quitter la boucle d'interaction, il faut utiliser la directive `#quit` (suivie de `;;`). De nombreuses directives sont disponibles car `utop` rajoute des directives à celles de l'interpréteur standard. Parmi celles que nous utiliserons :

- `#use` qui charge et interprète le contenu d'un fichier,
- `#directory` pour ajouter un repertoire de recherche de fichiers,
- `#cd` pour changer le repertoire courant de l'interpréteur,
- `#load` pour charger un fichier compilé,
- `#trace` / `untrace` pour obtenir / suspendre des traces des appels à la fonction cible,
- ... voir `http://caml.inria.fr/pub/docs/manual-ocaml/toplevel.html`.

Un mécanisme de gestion des lignes est offert et suit les commandes standards d'emacs [4], on peut ainsi modifier sa ligne et avancer / reculer dans l'historique (les flèches pour monter / descendre). Enfin, `utop` offre un mécanisme de complétion, sa barre inférieur affichant dynamiquement les complétions possibles, on complète en utilisant sa première proposition par la touche tabulation.

2.2 Les fonctions

Exercice 1

La fonction `map` prend en argument une fonction `f` et une liste `l`. Elle doit retourner la liste du résultat des applications de `f` aux éléments de `l`.

▷ **Question 1.1 :**
Sans utiliser l'interpréteur, donner le type de la fonction `map`.

▷ **Question 1.2 :**
Construire la fonction `map` et vérifier son type.

4. Consultez `#utop_bindings` pour connaître les raccourcis d'`utop`.

Exercice 2

Même exercice que précédemment mais avec la fonction `iterate` qui prend en paramètre un entier n et une fonction f et calcule f^n $(f(f(\ldots f())))$.

3 Un peu plus de ocaml

Exercice 3 (*Des arbres*)

▷ **Question 3.1 :**
Construire un type récursif arbre binaire.

▷ **Question 3.2 :**
Utiliser le type ainsi construit pour réaliser un arbre binaire de recherche entier. C'est-à-dire un arbre binaire ayant la propriété suivante : pour tous les nœuds, toutes les valeurs du sous-arbre gauche sont inférieures à celle du nœud, et toutes celles du droit lui sont supérieures.

Pour cela construisez une fonction `ajout` qui ajoute un entier dans un arbre de recherche entier.

▷ **Question 3.3 :**
Utiliser la fonction définie à la fonction précédente pour réaliser une fonction de tri des listes d'entiers.

4 Les expressions

Cette section contient uniquement un problème. Il s'agit de construire un évaluateur d'expression arithmétiques.

Exercice 4 (*Le calcul formel*)

▷ **Question 4.1 :**
Construire un type permettant de décrire des expressions arithmétiques basiques (des réels et les quatre opérateurs de base).

▷ **Question 4.2 :**
Construire une fonction `eval` qui permet d'évaluer une expression.

▷ **Question 4.3 :**
Étendre les expressions pour permettre l'utilisation de variables. Une variable sera une chaîne de caractères qui peut apparaître dans une expression. À l'évaluation, les variables auront une valeur.

▷ **Question 4.4 :**
Modifier la fonction `eval` qui permet d'évaluer une expression dans un environnement. Un environnement est une liste d'association qui associe des valeurs à des variables.

La rencontre d'une variable non définie doit conduire à un cas d'erreur.

▷ **Question 4.5 :**
Écrire une fonction `string_of_expr`.

▷ **Question 4.6 :**
Construire une fonction `derive` qui dérive une expression par rapport à une variable passée en argument.

▷ **Question 4.7 :**
Construire une fonction `simplifie` qui permet de simplifier une expression en utilisant les règles suivantes :

$$\begin{cases} -0 = 0 \\ \forall e \quad e + 0 = 0 + e = e \\ \forall e \quad e \times 0 = 0 \times e = 0 \\ \forall e \quad e \times 1 = 1 \times e = e \end{cases}$$

5 S'il vous reste du temps ou pour travailler en plus

Exercice 5 (*Tarot*)

▷ **Question 5.1 :**
Définir un type permettant de décrire les cartes d'un jeu de tarot.

▷ **Question 5.2 :**
Définir une fonction `toutes_les_cartes` qui construit l'ensemble des cartes d'une couleur reçue en paramètre.

▷ **Question 5.3 :**
Définir une fonction `string_of_cartes` qui construit une chaîne de caractères décrivant la carte reçue en paramètre.

A Les bases

Les commentaires en ocaml sont contenus entre `(* ... *)`. Les commentaires peuvent être inclus dans des commentaires.

A.1 Types

En ocaml, les types de bases sont :
- **rien** (`unit`) qui est composé d'une seule valeur `()`,
- les **entiers** (`int`) avec les opérations usuelles (`+`,`-`,`*`, `/`, `mod`, `int_of_float`, ...),
- les **réels** (`float`) avec les opérations usuelles (`+.`,`-.`,`*.`, `/.`, `**`, `float_of_int`, ...),
- les **booléens** `false` et `true` (`bool`) ainsi que les opérateurs logique et de comparaison usuels (`=`, `<>`, `<`, `>`, `<=`, `>=`, `not`, `&&`, `||`),
- les **caractères** (`char`) entre `'`,
- les **chaînes de caractères** (`string`) entre `"` avec les caractères spéciaux usuels (`\n`\`t`) et l'opérateur de concaténation `^`; tous les types précédents sont convertibles en chaîne de

caractère par des fonctions de la forme `string_of_`*type* ; on peut accéder au i^e caractères d'une chaîne de caractères par `tab.[i]` et ce caractère est modifiable par `tab.[i]<- e`,
— les **nuplets** (tuples) (`_ * _ * _`) le séparateur est `,`, les couples sont munis des opérateurs `fst` et `snd` ; les tuples plus grands doivent être manipulé par filtrage,
— les **tableaux** (`_ array`) entre `[||]` le séparateur est `;`, on accède à la case `i` par `tab.(i)` et on la modifie par `tab.(i)<- e` (attention à ne pas confondre avec les chaînes de caractères),
— les **listes** (`_ list`) entre `[]` le séparateur est `;`, le constructeur de liste est l'opérateur `::` qui ajoute un élément en tête d'une liste, il existe aussi un opérateur de concaténation `@`.

Pour en savoir plus, consultez la partie librairie (partie IV) de la documentation `http://caml.inria.fr/pub/docs/manual-ocaml`.

A.2 Les structures de contrôle

Les structures de contrôle usuelles existent en ocaml : le choix `if then else`, la séquence `;`, les blocs `begin ...end`, les itérations `for i = e1 to e2 do e3 done` et `while e1 do e2 done`.

La principale structure de contrôle en ocaml est le filtrage de motifs (*pattern matching*). Un motif est une valeur partiellement construite comportant des *trous* (en fait des variables non encore définies). Le filtrage est alors une opération permettant de comparer un motif à une valeur, si les deux entités sont comparables (*i.e.* ont la même forme), les trous (les variables libres) sont comblés par les sous-valeurs correspondantes (elles sont définies). Le procédé peut être rapproché des expressions régulières.

Le filtrage peut échouer, alors, le filtrage suivant est réalisé ou une exception est levée si aucun autre motif n'est disponible (voir exemple 2, ci-dessous). Il est à noter que l'interpréteur (et le compilateur) émet un avertissement si un filtrage de motif est incomplet et peut donc échouer.

Par exemple :

```
# let a = ([2;3;4;5],1);;
val a : int list * int = ([2; 3; 4; 5], 1)
# match a with (c,1) -> c;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(_, 0)
- : int list = [2; 3; 4; 5]
# match a with (c,2) -> c;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(_, 0)
Exception: Match_failure ("", 1, 0).
# match a with (2::c,1) -> c;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
([], _)
- : int list = [3; 4; 5]
```

Une variable libre ne peut figurer qu'une seule fois dans un motif. Il existe un motif particulier `_` qui permet de tout filtrer sans définir de variable.

Le filtrage de motifs peut se faire par les instructions `match` ou `function`. Il est également la

structure de base de définition des fonctions. Enfin, les exceptions sont également récupérées par filtrage. Ainsi, par exemple, un tri par insertion peut se programmer comme ci-dessous.

Tout d'abord, il faut une fonction qui permet d'insérer une valeur dans une liste triée en la maintenant triée :

```
let rec insert elt lst =
  match lst with
    [] -> [elt]
  | h::t -> if elt <= h then elt::lst else h::(insert elt t)
;;
```

Elle peut ensuite être utilisée pour trier une liste quelconque :

```
let rec sort lst =
  match lst with
    [] -> []
  | h::t -> insert h (sort t)
;;
```

La valeur retournée par une fonction est la valeur de la dernière expression évaluée avant le retour.

Cette fonctions permet également de démontrer le polymorphisme. En effet, comme elle n'utilise pas la forme des éléments de la liste, elle n'est pas dépendante de leur type. Ainsi, le type de la fonction sort sera : `'a list ->'a list` où `'a` représente une variable de type.

Comme dans tous les langages fonctionnels, l'approche est résolument récursive (définition par des `let rec`), c'est-à-dire les fonctions sont, en général, récursives. Il convient donc d'oublier les structures itératives (même si elles existent). C'est une des exigences qui devra être assurée dans le cadre de cette UV.

Vous noterez également qu'en général, les données ocaml ont une valeur fixe après leur définition. Par exemple, aucune modification n'est possible sur une liste, pour ajouter un élément à une liste, il faut construire une nouvelle liste.

A.3 Les fonctions

Une fonctions est une valeur à part entière en ocaml. On peut donner passer en argument une fonction. Par exemple, il est possible de définir une fonction `iter` qui va prendre en paramètre une autre fonction `f` et une liste `l` et qui va appliquer la fonction `f` à tous les élément de la liste `l`.

```
let rec iter f l =
  match l with
  | [] -> ()
  | h::t -> f h; iter f t
;;
```

Cette fonction a pour type `('a -> 'b) -> 'a list ->unit`.

On peut alors utiliser cette fonction `iter` pour afficher les éléments d'une liste de chaînes de caractères. La fonction d'affichage est `print_string`. Ainsi :

```
# iter print_string ["a";"b";"c";"\n"];;
abc
- : unit = ()
```

En ocaml, il est possible d'appliquer partiellement une fonction, c'est-à-dire, de lui fournir moins d'arguments que ce qu'elle attend. On obtient alors une nouvelle fonction qui attend les arguments non fournis. Ainsi :

```
# let print_list = iter print_string;;
val print_list : string list -> unit = <fun>
# print_list ["a";"b";"c";"\n"];;
abc
- : unit = ()
```

B Un peu plus de ocaml

B.1 La compilation

Les programmes ocaml peuvent aussi être compilés. Pour cela, dans un fichiers, on collecte des définitions et des expressions (le `;;` n'est alors plus nécessaire). Ce fichier est alors compilé par la commande `ocamlc` (ou `ocamlopt`). L'opération se déroule alors en deux étapes :

1. compilation des tous les modules `ocamlc -c ...`
2. éditions des liens de tous ces fichiers et production d'un exécutable `ocamlc -o prog ...`

Les fichiers (ou modules) compilés utilisent l'extension `cmo` (ou `cmx`). Le résultat de l'édition de liens par `ocamlc` est un fichier *bytecode* exécutable par la machine virtuelle (`ocamlrun`). Par défaut, ce fichier exécutable contient la ligne `#!/usr/bin/ocamlrun` [5] et est donc exécutable sur la plupart des systèmes sans lancer explicitement `ocamlrun`.

La distribution d'ocaml contient un outil de compilation automatique `ocamlbuild`. Celui-ci vise une cible que l'utilisateur lui fournit. Il analyse cette cible pour trouver toutes les opérations de compilation nécessaires. Il compile alors tous les fichiers dans un repertoire `_build` dans le repertoire courant. Il rale s'il trouve des fichiers compilés en dehors du repertoire `_build` ! Il faut alors les éliminer. Il existe deux types de cible selon que l'on souhaite utiliser `ocamlc` (`.byte`) ou `ocamlopt` (`.native`). Ainsi par exemple :

```
ocamlbuild -libs unix main.native
```

compile le fichier `main.ml` et toutes ses dépendances avec `ocamlopt`. Il liera le programme à la bibliothèque `unix` et produira un executable de nom `main.native`. Il créera enfin un lien dans le repertoire courant vers cet exécutable.

L'outil `ocamlbuild` peut également lancer l'exécution si on ajoute `--` suivi des arguments de la ligne de commande. Ainsi,

```
ocamlbuild main.byte -- file.txt
```

lance toutes les étapes de compilation puis exécute le programme `main.byte` en lui passant en paramètre `file.txt`.

Un mécanisme de fichier d'option permet un contrôle plus fin (fichier `_tags`) et on peut également écrire des *plugins* en ocaml. Le lecteur qui souhaite des compléments est redirigé vers le chapitre du manuel qui lui est consacré `https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc`.

5. le chemin peut varier !

B.2 Les modules

Un module est un ensemble de définitions (des types, des valeurs, des fonctions, ...). Il possède une *signature* et une *structure*. La signature définit l'interface (extérieure) du module, c'est-à-dire les entités publiques introduites par le modules que les autres programmes peuvent utiliser. La structure doit au moins définir les entités déclarées dans la signature (avec des types compatibles). Les entités de la structure qui ne sont pas déclarées dans la signature ne sont pas utilisables par les autres programmes. Cette possibilité permet de rendre des types abstraits (*i.e.* non manipulables voir section B.4).

Dans sa réalisation la plus simple un module est un fichier contenant les déclarations du module (extension `ml`). Sa signature doit alors être dans un fichier de même nom et d'extension `mli`. Dans ce cas, le nom du module correspond au nom du fichier.

Lors de la compilation, la signature est compilée dans un fichier d'extension `cmi`. Si aucune signature n'est fournit toutes les entités du module (*i.e.* le fichier) sont disponibles (à travers un fichier `cmi` généré automatiquement).

Dans un programme utiliser une valeur `toto` d'un autre module `tutu` consiste à la préfixée par le nom du module : `tutu.toto`.

En fait, les modules sont beaucoup plus puissants que présenté ci-dessus, vous pouvez consulter `http://caml.inria.fr/pub/docs/manual-ocaml/moduleexamples.html` pour en savoir plus.

B.3 Les types construits

En ocaml, on peut déclarer de nouveaux types (syntaxe `type truc =`). Ces types peuvent être paramétriques (syntaxe `type 'var truc =`). Ces nouveaux types sont généralement des alias, c'est-à-dire que les valeurs d'un tel nouveau type est également de l'ancien type.

Parmi les types que l'on peut définir la forme la plus utilisée est le type somme. Un tel type est définit par une définition de la forme :

```
type nom =
  | Nom1 of t1
  | Nom2 of t2
  | ...
  | Nomk of tk
;;
```

Il regroupe toutes les valeurs construites à l'aide des **constructeurs** `Nom1` à `Nomk`[6]. Construire une valeur de ce type est réalisé par, par exemple, `Nom1(toto)` si `toto` est une valeur de type `t1`.

Ces types sont alors manipulés par filtrage de motifs.

Les types sommes peuvent également être récursifs, c'est-à-dire qu'un de ces sous-éléments l'utilise. Le type liste est un exemple de type récursif, il est, en effet, définit par :

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

Un autre type somme est définit par défaut en ocaml, le type `option` :

6. Attention à la majuscule !

```
type 'a option =
  | None
  | Some of 'a
```

On peut également définir des types enregistrements :

```
type ratio = {num: int; denum: int}

let add r1 r2 =
   {num = r1.num * r2.denum + r2.num * r1.denum;
    denum = r1.denum * r2.denum}

add {num=1; denum=3} {num=2; denum=5}
```

B.4 L'abstraction de type

Un des intérêts de la notion de modules est la possibilité d'abstraire un type. En effet, il est possible de déclarer un nouveau type (dans une signature) sans montrer (et rendre accessible) sa réalisation. Par exemple, le type précédent `ratio` peut être abstrait par la signature suivante (fichier `Ratio.mli`) :

```
type ratio
val add : ratio -> ratio -> ratio
val num : ratio -> int
val denum : ratio -> int
val create : int -> int -> ratio
val print : ratio -> unit
```

La réalisation figure alors dans le fichier `Ratio.ml` :

```
type ratio = {num: int; denum: int}

let add r1 r2 =
   {num = r1.num * r2.denum + r2.num * r1.denum;
    denum = r1.denum * r2.denum}

let num r = r.num
let denum r = r.denum

let create n d = {num = n; denum = d}

let print r =
  print_int r.num;
  print_string "/";
  print_int r.denum;
```

La forme du type `ratio` n'est plus accessible, le code suivant :

```
let r = Ratio.create 1 1 in
  print_int r.num
```

provoque l'erreur :

```
File "UseRatioError.ml", line 2, characters 12-17:
Unbound record field label num
```

Pour utiliser le type, il faut alors passer par les fonctions fournies dans le modules :

```
let r1 = Ratio.create 1 1
and r2 = Ratio.create 1 2 in
  Ratio.print (Ratio.add r1 r2)
```

B.5 Exceptions

OCaml permet l'utilisation d'exceptions. Celles-ci sont déclarées par la construction `exception`, sont levées par l'opérateur `raise` et sont rattrapées par la construction `try ... with`.

Par exemple, la fonction `head` qui renvoie la tête d'une liste peut lever une exception la liste reçue en argument est vide :

```
exception Empty_list

let head l =
  match l with
  | [] -> raise Empty_list
  | hd :: tl -> hd
```

Dans la librairie ocaml, on rencontre des fonctions permettant de manipuler la notion de dictionnaire (appelé une liste d'associations). Le module `List` contient ainsi une fonction `assoc` qui prend une clé et une liste d'association et renvoie la valeur associée à la clé dans la liste d'association. Si la clé ne figure pas dans la liste d'association une exception `Not_found` est levée. Ainsi, écrire une fonction `name_of_digit` qui converti un chiffre (pas un nombre) en chaîne de caractères peut s'écrire :

```
let name_of_digit digit =
  try
    List.assoc digit [0, "zéro"; 1, "un"; 2, "deux"; 3, "trois"; 4, "quatre";
                      5, "cinq"; 6, "six"; 7, "sept"; 8, "huit"; 9, "neuf"]
  with Not_found ->
    "not a binary digit";;
```

La partie `with` contient un filtrage de motifs quelconque et les exceptions peuvent contenir des données.