Éléments du langage OCAML

Adapté de *Programmation de droite à gauche – et vice-versa* Ed. Paracamplus, 2012

Ce document présente les quelques éléments du langage OCAML qui doivent permettre de lire et comprendre les exemples de code donnés dans le cours. Il va parfois un peu plus loin que le cours. Les éléments sont organisés sous forme de fiches thématiques. Il ne s'agit donc nullement ici d'un manuel de référence exhaustif du langage OCAML, ou d'un tutoriel d'apprentissage de ce langage.

Pour ce dernier aspect, si nous ne donnons que peu d'exemples d'utilisation des aspects du langage dans cette annexe, nous renvoyons le lecteur aux paragraphes du poly de cours où ceux-ci sont développés. Pour le premier aspect, nous renvoyons le lecteur au manuel de référence en ligne :

`http://caml.inria.fr/pub/docs/manual-ocaml/index.html`

Table des matières

1 Écriture de l'application de fonction

Un trait syntaxique perturbant d'OCAML est l'écriture de l'application des fonctions. Il devient néanmoins simple à appréhender si l'on garde en tête que

OCAML est un langage applicatif statiquement typé.

La syntaxe de l'application y est simplifiée à l'extrême : l'écriture

```
f x 0
```

désigne l'application de la fonction `f` aux deux arguments `x` et `0`. Cette expression est admise dans le langage dès que `f` est une fonction qui attend *au moins deux arguments* et que, bien entendu, les types de `x` et de `0` sont compatibles respectivement avec les types de ces deux premiers arguments.

Il n'est pas interdit d'ajouter des parenthèses pour délimiter une application. Ainsi, l'écriture parenthésée

```
(f x 0)
```

est également licite, mais optionnelle.

De manière générale, l'application a la forme

(e e_1 ... e_n)

Lorsque e_1, ... ou e_n sont des expressions complexes il devient *nécessaire de délimiter leur écriture par des parenthèses*. Par exemple, si `g` est une fonction (disons, à un seul argument), on écrit

```
f (g x) 0
```

pour donner à `f` non plus simplement `x` mais le résultat de son application à `g`.

Attention Lorsque l'on débute en OCAML il faut prendre garde à ne pas céder au réflexe de vouloir délimiter la liste des arguments passés à une fonction avec les parenthèses. Par exemple, si l'écriture `(f x 0)` est correcte alors elle n'est pas équivalente à `f (x 0)` où les parenthèses signifient l'application de `x` à `0` et donc `f (x 0)` signifie l'application de `f` à l'unique valeur `(x 0)`.

Les diverses notations de l'application En standard, l'application est *préfixe* : la fonction est devant les arguments. Mais OCAML autorise également l'utilisation de quelques opérateurs binaires en notation *infixe* (le symbole d'opération entre les arguments). Ce sont pour la plupart des symboles, tels ceux des opérations arithmétiques, des connecteurs booléens, etc.

Toutes les constructions syntaxiques, à l'exception des *définitions globales* sont des applications. En particulier, les constructions syntaxiques correspondant à des structures de contrôles, telle l'alternative `if ... then ... else ...` est l'application de l'opérateur *« mixfixe »* `if-then-else` à trois arguments.

« Associativité » de l'application En fait, en OCAML et dans les dialectes de la famille ML, l'application est *associative* à gauche : de façon générale,

les expressions (e e_1 e_2) et ((e e_1) e_2) sont équivalentes.

Elles donnent la même valeur si elles sont correctement typées ; elles sont toutes deux rejetées sinon.

Sur l'application, il est interessant de la mettre en rapport avec les types fonctionnels en OCAML qui sont expliqués ci-dessous, paragraphe 15.

2 Définitions de valeurs ou de fonctions

La forme de base des définitions en OCAML est construite avec le mot clé `let` et le symbole = . Par exemple

```
let x = 3*k + 42
```

associe au nom `x` la valeur de l'expression `3*k + 42`. Il faut naturellement pour que cette définition soit effective que `k` ait été associé à une valeur. La forme générale d'une définition est donc

$$\texttt{let}\ x\ \texttt{=}\ e$$

où x est un identificateur et e une expression. La définition crée une *liaison* entre le nom x et la *valeur* de l'expression e. Cette liaison est conservée dans l'*environnement* d'exécution du programme.

Pour définir une fonction, on utilise la construction `let` en faisant figurer les *paramètres formels* de la fonction entre le nom de celle-ci et le symboble = ; comme dans :

```
let f k = 3*k + 42
```

La forme générale d'une définition de fonction est donc

$$\texttt{let}\ f\ x_1\ \texttt{...}\ \ x_n\ \texttt{=}\ e$$

où f est le nom de la fonction définie et $x_1 \ldots\ x_n$ les noms des paramètres formels utilisés dans l'expression e.

Définitions récursives Lorsque l'on désire donner la définition récursive d'une fonction, il faut l'indiquer avec le mot clé `rec`. La forme générale d'une définition récursive est

$$\texttt{let rec}\ f\ x_1\ \texttt{...}\ \ x_n\ \texttt{=}\ e$$

où f est le nom de la fonction définie et $x_1 \ldots x_n$ les noms des paramètres formels servant dans l'expression e. De plus, si le nom f est utilisé dans e, il sera compris comme celui de la fonction f en cours de définition.

Les définitions mutuellement récursives sont signalées par le mot clé `and` :

$$\begin{array}{c}\texttt{let rec}\ f_1\ x_1\ \texttt{...}x_n\ \texttt{=}\ e_1\\ \vdots\\ \texttt{and}\ f_k\ y_1\ \texttt{...}y_m\ \texttt{=}\ e_k\end{array}$$

dans chacune des expressions e_1, ..., e_k, les occurrences des noms f_1, ..., f_k sont interprétées comme ceux définis mutuellement.

Environnements Les définitions ne produisent pas de valeur. Elles ont pour objet de mémoriser les associations entre des noms et des valeurs. L'ensemble de ces associations constitue un *environnement*. Par exemple, la suite de définitions

```
let a = 3.0
let b = 4.0
let c = 5.0
let s = (a +. b +. c) /. 2.0
let t = sqrt (s *. (s -. a) *. (s -. b) *. (s -. c))
```

construit un environnement que l'on peut représenter par la liste de couples

$$[(\texttt{t}, 6.0); (\texttt{s}, 6.0); (\texttt{c}, 5.0); (\texttt{b}, 4.0); (\texttt{a}, 3.0)]$$

Une définition peut utiliser tous les noms qui ont été définis *avant* elle dans le texte du programme. C'est ce que l'on appelle la *portée lexicale*.

Types et définitions Il n'est pas nécessaire d'indiquer le type attendu des arguments, ni le type du résultat (*type de retour*) dans les définitions en OCAML : le compilateur intègre un mécanisme d'*inférence de type* qui calcule automatiquement, lorsqu'il existe, le type des valeurs et des fonctions. Toutefois, la syntaxe du langage autorise le programmeur à expliciter le type des valeurs manipulées en posant des *containtes de type*. D'ailleurs certaines définitions de classes ne sont pas compilées sans l'aide de telles contraintes.

Nous avons pris le parti dans les différents chapitres de ce livre d'utiliser systématiquement l'indication de types des arguments des fonctions ainsi que leur type de retour, en écrivant nos définitions selon cette forme

$$\texttt{let}\ [\texttt{rec}]\ f\ \texttt{(}x_1\texttt{:}\tau_1\texttt{)}\ \texttt{...(}x_1\texttt{:}\tau_1\texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ e\ ^1$$

où l'écriture $\texttt{(}x_i\texttt{:}\tau_i\texttt{)}$ indique que le $i$ème paramètre doit être de type τ_i et le $\texttt{:}\tau$, avant le signe = indique que τ est le type de la valeur calculée par application de f.

Définition locale

Une définition locale associe une définition à une expression particulière. La forme de base des définitions locales en OCAML est construite à l'aide des mots clés `let`, `in` et du symbole = arrangés de la façon suivante :

$$\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

Attention une définition locale `let-in` construit une **expression**. Sa valeur est celle de e_2 dans laquelle la variable x a la valeur de e_1. La *portée* du nom x est l'expression e_2.

On peut définir localement une fonction :

$$\texttt{let}\ f\ x_1\ \texttt{...}\ \ x_n\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

Si la fonction est récursive, on rajoute le mot clé `rec` :

$$\texttt{let rec}\ f\ x_1\ \texttt{...}\ \ x_n\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

On utilise en général les définitions locales « à l'intérieur » d'une autre définition.

3 Expressions fonctionnelles et fermetures

On trouve parfois en mathématique des définitions de fonctions ainsi posées :

$$\begin{array}{rl} S: & \mathbb{N} \to \mathbb{N} \\ & n \mapsto \frac{n(n+1)}{2} \end{array}$$

où $n \mapsto \frac{n(n+1)}{2}$ signifie que la valeur en n de S est la valeur de $\frac{n(n+1)}{2}$. C'est-à-dire que S est *« la fonction qui à n associe $\frac{n(n+1)}{2}$ »*.

Il existe en OCAML une expression équivalente à $n \mapsto \frac{n(n+1)}{2}$. C'est une *expression fonctionnelle* qui utile le mot clé `fun`, le symbole `->` (« tiret-supérieur ») et qui s'écrit

1. Les crochets autour du `rec` indiquent simplement son caractère optionnel.

```
fun n -> (n * (n+1))/2
```

De manière générales, les expressions fonctionnelles ont la forme suivante :

`fun` x_1 `...` x_n `->` `e`

Remarque : il n'y a pas d'expression fonctionnelle récursive, il n'y a que des *définitions récursives*.

L'usage des expressions fonctionnelles est fréquent en conjonction avec l'application d'un itérateur. Par exemple, pour définir la fonction `list_mult2` du paragraphe **??**, page **??**, il n'est pas utile de nommer la fonction de multiplication par 2, on peut écrire plus directement :

```
let list_mult2 (xs: int list) : int list =
  List.map (fun x -> 2*x) xs
```

La définition des enregistrements fonctionnels du paragraphe **??** donne un autre exemple, moins courant, d'utilisation des expressions fonctionnelles.

Valeur d'une expression fonctionnelle Comme leurs noms l'indiquent, les expressions fonctionnelles sont des *expressions*. Elles ont donc à ce titre une *valeur*. Cette valeur est d'un genre particulier appelé *fermeture*. Pour schématiser, une fermeture est un couple constitué du code d'une fonction et de l'environnement de cette fonction ; c'est-à-dire, la valeur de toutes les variables préalablement définies qu'utilise la fonction.

Puisque les expressions fonctionnelles ont une valeur, on peut également définir une fonction comme toute autre valeur. Ainsi, écrire

```
let f x1 .. xn = e
```

est rigoureusement équivalent à

```
let f = fun x1 .. xn -> e
```

4 L'alternative

est construite avec les mots clé `if`, `then` et `else`. Sa forme générale est

`if` e_1 `then` e_2 `else` e_3

où e_1 est nécessairement de type `bool` et e_2 et e_3 d'un type quelconque, identique pour les deux expressions. On peut dire que le type de l'alternative est `bool -> 'a -> 'a -> 'a`. La valeur de `if` e_1 `then` e_2 `else` e_3 est la valeur de e_2 si la valeur de e_1 est `true` ou la valeur de e_3 sinon. Selon la valeur de e_1, une seule des deux branches de l'alternative est calculée. Le type de l'expression est le type commun à e_2 et e_3.

L'écriture `if` e_1 `then` e_2 `else` e_3 est une expression comme une autre. On peut donc composer, c'est-à-dire, imbriquer, des alternatives. Par exemple :

```
if (n < 0) then "strictement négatif"
else if (n = 0) then "null"
else "strictement positif"
```

ou encore

```
if (n <= 0) then
  if (n = 0) then "null"
  else "strictement négatif"
else "strictement positif"
```

On peut également utiliser une expression alternative comme argument d'application de fonction. Par exemple, on peut construire ainsi une expression d'incrément conditionnel :

```
x + (if (x < max) then 1 else -1)
```

qui donne le même résultat que

```
if (x < max) then x+1 else x-1
```

Dans l'écriture d'une expression alternative doivent toujours figurer les deux cas `then` et `else`, à une seule exception prêt : celle où e_2 est de type `unit` et où e_3 est égal à `()`. Dans ce cas, et dans ce cas uniquement, il est autorisé d'écrire : `if` e_1 `then` e_2.

5 Filtrage de motifs

Cette construction est liée à la définition de types sommes dans le langage (voir *infra* 17). Chaque valeur appartenant à un type somme est toujours égale à une expression qui s'écrit sous la *forme* `C (`$\mathtt{e}_1$`, ..., `$\mathtt{e}_n$`)` où `C` est un *constructeur* du type et $\mathtt{e}_1$, ..., $\mathtt{e}_n$ des expressions donnant la valeur de ses arguments. Un type somme est défini par la donnée d'une liste finie de constructeurs, disons $\mathtt{C}_1$, ..., $\mathtt{C}_k$. Les expressions de la forme $\mathtt{C}_1$ `(`$\mathtt{e}_1$`, ..., `$\mathtt{e}_n$`)`, ..., $\mathtt{C}_k$ `(`$\mathtt{e}_1$`, ..., `$\mathtt{e}_m$`)` désignent les valeurs de ce type. Réciproquement, *toute valeur* appartenant à ce type somme est égale à l'une des valeurs que l'on peut écrire sous la forme $\mathtt{C}_1$ `(`$\mathtt{e}_1$`, ..., `$\mathtt{e}_n$`)`, ..., *ou bien* $\mathtt{C}_k$ `(`$\mathtt{e}_1$`, ..., `$\mathtt{e}_m$`)`. Ainsi, pour calculer avec des valeurs appartenant à un type somme, on commence par *reconnaître* à quel *cas de constructeur* il appartient, afin de déterminer le calcul à appliquer dans chaque cas.

La construction de *filtrage* est une structure de contrôle d'analyse de cas, un peu comme le `switch` de C ou de Java, mais utilisé avec des descriptions de cas plus complexes : *les motifs*. L'idée sous-jacente à l'écriture des motifs est que puisque toute valeur appartenant à un type somme est égale à l'application d'un constructeur à quelque chose, en écrivant le motif `C (`$\mathtt{x}_1$`, ..., `$\mathtt{x}n$`)`, où, $\mathtt{x}_1$, ..., $\mathtt{x}_n$ sont des noms de variables, on identifie toutes les valeurs appartenant au cas d'application du constructeur `C`. Et, bien plus, si la valeur considérée est `C (`$\mathtt{v}_1$`, ..., `$\mathtt{v}_n$`)`, on peut également *nommer* respectivement ces valeurs avec $\mathtt{x}_1$, ..., $\mathtt{x}_n$ et utiliser ces noms dans le calcul associé au cas déterminé par `C`.

Syntaxiquement, un motif est
- soit un nom de variable ;
- soit un des constructeurs sans argument du type ;
- soit un constructeur appliqué à un n-uplet de motifs ;
- soit le caractère particulier `_` (souligné) qui est comme une variable sans nom, un motif *anonyme*.

Les motifs sont des expressions *linéaires* en leurs variables : un même nom de variable ne peut apparaître deux fois dans un motif (à l'exception de la *variable sans nom* `_`). Le compilateur rejette les écritures de motifs qui ne respectent pas cette règle avec le message :

```
Error: Variable [...] is bound several times in this matching
```

La structure de filtrage est introduite par les mots clés `match` et `with`. On place entre ces deux mots l'expression de la valeur à traiter. Chaque *cas de filtrage* est un couple constitué d'un motif et d'une expression, séparés par le symbole réservé `->` (« tiret-supérieur »). Les cas de filtrage sont séparés par la barre verticale `|`. La forme générale d'une construction de filtrage est

`match` e `with` C_1 [p_1] `->` e_1 [`|` C_2 [p_2] `->` e_2 ...]

où les crochets indiquent les arguments optionnels.

Par exemple, si un type est défini par les trois constructeurs `C0`, `C1` et `C2` avec, respectivement, 0, 1 et 2 arguments, un filtrage pour les valeurs de ce type s'écrit :

```
match e with
  C0 -> e0
| C1 x -> e1
| C2 (x1, x2) -> e2
```

Extensions du filtrage La possibilité d'utiliser la structure de contrôle du filtrage est étendue en OCAML à d'autres types que les seuls types sommes. Les n-uplets peuvent être des motifs de filtrage (voir **??**, page **??**). Les entiers sont également motifs de filtrage (voir l'exemple de définition de la factorielle en **??**, page **??**). Les booléens peuvent également être motifs de filtrage comme, par exemple dans cette définition du « ou exclusif » :

```
let xor b1 b2 =
  match (b1, b2) with
    (true, true) -> false
  | (true, false) -> true
  | (false, true) -> true
  | (false, false) -> false
```

On écrira une définition plus concise en utilisant le *motif anonyme* :

```
let xor b1 b2 =
  match (b1, b2) with
    (true, false) -> true
  | (false, true) -> true
  | _  -> false
```

Notez que dans ces deux définitions le filtrage est appliqué sur le couple formé des deux arguments de la fonction. Avec un peu de pratique, on se passera des parenthèses autour des couples pour écrire :

```
let xor b1 b2 =
  match b1, b2 with
    true, false -> true
  | false, true -> true
  | _  -> false
```

Les valeurs des types enregistrement sont également motifs de filtrage. Nous renvoyons le lecteur curieux au manuel de référence en ligne du langage sur ce point.

Voir aussi Malgré l'écriture spéciale de ses constructeurs, le type des listes est un type somme. Ses valeurs sont donc motifs de filtrage. L'utilisation du filtrage avec les listes est expliquée au paragraphe **??**.

6 Exceptions

Les exceptions sont des valeurs du langage qui appartiennent au type `exn`. C'est un type très particulier que l'on peut qualifier de « type somme *ouvert* ». La bibliothèque standard de OCAML fournit un certain nombre d'exceptions prédéfinies :

`Not_found, Division_by_zero, End_of_file`
`Invalid_argument, Failure`, etc.

Les deux dernières sont des constructeurs fonctionnels du type `exn` qui s'appliquent à un argument de type `string`. Par exemple, l'exception

`(Invalid_argument "index out of bounds")`

est le résultat d'une tentative d'accès à un élément de tableau en dehors de son intervalle d'indices.

Définir une exception Le programmeur peut définir de nouvelles exceptions (c'est-à-dire de nouveaux constructeurs du type `exn`) avec le mot clé `exception`. La syntaxe de définition est

`exception` *Ident* [`of` τ]

où *Ident* est le nom de la nouvelle exception, dont l'initiale est **obligatoirement majuscule** puisque c'est un constructeur de type somme, et où le type de l'argument, s'il y en a, est τ que l'on place après le mot clé `of`.

Il faut noter que le paramètre d'une exception est toujours *monomorphe*, c'est-à-dire, sans variable de type. Une définition d'exception comme le tente l'exemple ci-dessous n'est pas acceptable :

```
exception Wrong_list of ('a list)
```

Déclencher une exception Le statut particulier des exceptions tient à leur utilisation dans les programmes : signaler une erreur ou l'impossibilité d'un calcul (par exemple, `Division_by_zero`). Dans ces situations, le programme doit s'interrompre. Pour interrompre le déroulement ordinaire d'un programme en indentifiant la cause de cette interruption, on *déclenche* une exception avec la fonction primitive `raise` de type `exn -> 'a`. Par exemple, une fonction de recherche s'achèvera par `(raise Not_found)` en cas d'échec de la recherche.

La fonction prédéfinie `failwith` appliquée à une chaîne de caractères déclenche l'exception `Failure` avec le message passé en argument. Elle est de type `string -> 'a`.

Rattraper une exception Lorsqu'une portion de code est susceptible de déclencher une exception, le programmeur peut prévoir cette éventualité et y associer une poursuite de calcul *ad hoc* avec la structure de contrôle `try-with`. L'expression à contrôler est écrite entre le `try` et le `with`; le ou les traitements à appliquer en cas de déclenchement d'une expression sont écrits après le `with`, à la manière des cas de motifs dans un filtrage. La syntaxe générale est la suivante :

`try` e `with` Ex_1 [v_1] `->` e_1 [`|` Ex_2 [v_2] `->` e_2 ...]

où les crochets indiquent les arguments optionnels.

7 Modules et nom qualifié

Le code qui référence les entités (types, fonctions, etc.) exportées par un module `M` utilise en général leur *nom qualifié* ; c'est-à-dire le nom tel que défini dans l'interface préfixé par le nom du module dont l'**initiale est obligatoirement majuscule**. Les noms du module et de l'entité sont séparés par un point. Par exemple, `List.length` est le nom complet (*qualifié*) de la fonction `length` définie dans le module `List` de bibliothèque standard du langage.

On peut éviter l'usage de la notation qualifiée en « ouvrant » le module `M` avec la directive `#open M`. Dans une large application ayant recours à plusieurs modules, il faut naturellement prendre garde au risque d'ambiguïté de noms. Par exemple, beaucoup de modules de la bibliothèque standard utilisent le nom `create`.

8 Le type `unit`

Le type `unit` est un type prédéfini à une seule valeur notée `()`. Il est, peu ou prou, l'analogue du type `void` de C/Java. Lorsque la valeur d'une expression est de type `unit`, c'est l'indice que celle-ci appartient au monde *impératif*. Par exemple, les fonctions d'écriture ou d'affichage, qui procèdent par *effet de bord* ont

`unit` comme type de retour ; les opérations d'affectation également ; et encore, les expressions construites avec les boucles `for` ou `while` ont pour valeur la valeur `()` du type `unit`.

Cette valeur est également utilisée pour définir des fonctions qui n'ont pas besoin d'argument, c'est-à-dire, des fonctions dont la valeur ne dépend que de celle de leur environnement au moment de leur invocation. Pour invoquer de telles fonctions, on les applique simplement à la valeur `()`.

9 Booléens : type `bool`

Les deux constantes booléennes, valeurs du type `bool`, sont `true` et `false`. La négation s'écrit `not` b où b est une valeur de type `bool`. La conjonction et la disjonction sont des opérateurs infixes notés respectivement b_1 `&&` b_2 et b_1 `||` b_2, où b_1 et b_2 sont de type `bool`. Ce sont des opérateurs *séquentiels*. C'est-à-dire que si b_1 est `false` dans b_1 `&&` b_2 alors b_2 n'est pas évalué et le résultat est `false` ; si b_1 est `true` dans b_1 `||` b_2 alors b_2 n'est pas évalué et le résultat est `true`.

Les opérateurs de comparaisons de base sont *polymorphes* (de type `'a -> 'a -> bool`). On a les opérateurs infixes

- e_1 `=` e_2 pour l'égalité structurelle (même représentation mémoire) ;
- e_1 `<>` e_2 la négation de e_1 `=` e_2 ;
- e_1 `>` e_2 le test de supériorité stricte ;
- e_1 `<` e_2 le test d'infériorité stricte ;
- e_1 `>=` e_2 le test de supériorité au sens large (supérieur ou égal) ;
- e_1 `<=` e_2 le test d'infériorité au sens large (inférieur ou égal) ;
- e_1 `==` e_2 pour l'égalité physique (même emplacement mémoire) ;
- e_1 `!=` e_2 la négation de e_1 `==` e_2 ;

Les tests `< > <= >=` ont le comportement attendu sur les types de base `int`, `float` et `char`. L'ordre sur les chaînes de caractères, les listes et les tableaux est l'ordre lexicographique. L'utilisation des tests d'ordre sur les autres types donnera toujours un résultat, mais leur usage est déconseillé.

10 Entiers : type `int`

Les constantes entières peuvent être désignées en notation *décimale* par des suites des dix chiffres en base 10 : `0 ... 9`. La suite est précédée du symbole `-` (« tiret ») si l'on veut désigner une valeur entière négative.

On peut également utiliser la notation *hexadécimale* en combinant les dix chiffres `0 ...9` et les six lettres `A ...F`, ou `a ...f`. Dans ce cas, la suite est préfixée par les symboles `0x` (ou `0X`) ; eux-mêmes précédés du symbole `-` (« tiret ») si l'on veut désigner une valeur entière négative.

On peut enfin utiliser les notations binaires (base 2) ou octal (base 8) pour lesquelles les préfixes sont respectivement `0b` (ou `0B`) et `0o` (ou `0O`).

L'ensemble des entiers représentables en mémoire est fini :

- `max_int` désigne la plus grande valeur entière ;
- `min_int` désigne la plus petite valeur entière.

Leur valeur dépend de l'architecture machine : 32 *bits* ou 64 *bits*. Seuls, 31 ou 63 bits sont utilisés pour coder les entiers. Le *bit* inutilisé est réservé à l'usage du gestionnaire automatique de mémoire.

Les opérations arithmétiques binaires prédéfinies adoptent une notation infixe. Elles sont de type `int -> int -> int` :

- e_1 `+` e_2 addition ;
- e_1 `-` e_2 soustraction ;

— e_1 `*` e_2 multiplication ;
— e_1 `/` e_2 division entière ;
— e_1 `mod` e_2 *modulo*.

Les calculs sont effectués modulo 2^{31} ou 2^{63}, selon l'architecture. Il n'y a donc *pas de débordement*. La division par 0 provoque le déclenchement de l'exception `Division_by_zero`.

Les opérations logiques *« bit-à-bit »* binaires prédéfinies sont également en notation infixe et de type `int -> int -> int` :
— e_1 `land` e_2 conjonction *« bit-à-bit »* ;
— e_1 `lor` e_2 disjonction *« bit-à-bit »* ;
— e_1 `lsr` e_2 décalage à droite ;
— e_1 `lsl` e_2 décalage à gauche ;
— e_1 `asr` e_2 décalage arithmétique à droite (le signe est conservé).

Les opérations unaires prédéfinies de type `int -> int` sont :
— `succ` la fonction *successeur* ;
— `pred` la fonction *prédécesseur* ;
— `abs` la valeur absolue ;
— `lnot` la négation logique *« bit-à-bit »*.

Attention : le symbole `-` (le « tiret ») sert à la fois de symbole unaire pour les valeurs négatives et de symbole binaire pour la soustraction. Il peut parfois provoquer des ambiguïtés d'analyse des expressions. Par exemple, si `f` est de type `int -> int -> int`, il faut parenthéser ses arguments négatifs : on écrira `f 1 (-1)` car `f 1 -1` est interprété comme `(f 1) - (1)`.

11 Flottants : type `float`

Les flottants sont le plus souvent notés comme des « nombres à virgule ». La virgule est notée par le *point décimal*, comme dans `3.1416`. La partie droite peut être omise lorsqu'elle vaut 0 : `5.0` et `5.` donnent la même valeur ; mais le point ne peut pas être omis : `5.` est de type `float` alors que `5` (sans le point) est de type `int`. En revanche, une partie gauche nulle ne peut être omise : `.5` est incorrect, il faut écrire `0.5`.

Quatre constantes symboliques sont prédéfinies pour le type `float`
— `max_float` pour la plus grande valeur *positive finie* ;
— `min_float` pour la plus petite valeur *positive non nulle* ;
— `infinity` pour la valeur infinie positive, par exemple, valeur d'une division par 0 d'un nombre positif de type `float` ;
— `neg_infinity` pour la valeur infinie négative (division par 0 d'un négatif) ;
— nan pour *« not a number »* (division de 0 par 0).

Les trois dernières constantes sont requises par le standard IEEE 754 auquel obéit l'implémentation des valeurs de type `float` en OCAML. Les opérations sur les flottants ne déclenchent pas d'exception sur les divisions par 0 ou les débordements, mais utilisent ces valeurs.

Les opérations binaires de base sur les flottants sont
— e_1 `+.` e_2 addition ;
— e_1 `-.` e_2 soustraction ;
— e_1 `*.` e_2 multiplication ;
— e_1 `/.` e_2 division ;
— e_1 `**` e_2 élévation à la puissance.

Notez que les opérations artithmétiques utilisent des symboles différents de ceux pour les entiers.

Les conversions entre entiers et flottants doivent être explicites. Les fonctions de conversions prédéfinies sont :
— `float_of_int` de type `int -> float` convertit son argument entier en flottant ;

— `int_of_float` de type `float -> int` renvoie la partie entière de son argument (le résultat est indéterminé pour `nan` ou les nombres trop grand).

La valeur absolue, les fonctions trigonométriques et d'autres sont également prédéfinies. Nous renvoyons le lecteur au manuel de référence en ligne du langage.

12 Caractères : type `char`

Les constantes de caractères sont écrites entre apostrophes, comme : `'A' 'a' '0' '=' '?'`. Certains caractères ont une notation particulière :

— `'\n'` pour la fin de ligne (*linefeed*) ;
— `'\t'` pour la tabulation ;
— `'\''` pour l'apostrophe ;
— `'\"'` pour le guillemet anglais (*double quote*) ;
— `'\\'` pour le *backslash*.

OCAML ne connaît que le codage ASCII des caractères. On peut également désigner un caractère par son code ASCII. On écrit alors `'\ddd'` où `ddd` est une suite de 3 chiffres désignant un entier compris entre 0 et 255. Il faut écrire `'\065'` pour désigner `'A'` et non `'\65'` On peut également utiliser une notation hexadécimale du code ASCII de la forme `'\xhh'` où `h` est un chiffre hexadécimal.

Les fonctions sur les caractères sont fournies par le module `Char`. Citons :

— `Char.code` de type `char -> int` qui donne le code ASCII de son argument ;
— `Char.chr` de type `int -> char` qui donne le caractère dont le code ASCII est passé en argument ou l'exception `Invalid_argument "Char.chr"` si l'entier passé en argument est en dehors de l'intervalle 0. . .255.
— `Char.escaped` de type `char -> string` qui convertit le caractère passé en argument en la chaîne qui contient ce seul caractère.

13 Chaînes de caractères : type `string`

Les chaînes de caractères s'écrivent entre deux `"` (guillemets anglais), comme l'inévitable `"Hello world!"`. On peut utiliser les écritures spéciales de caractères : dans `"Hello\tworld!\n"`, une tabulation sépare les deux mots et le caractère de fin de ligne est inséré en fin de chaîne (ce qui commande un passage à la ligne à l'affichage). La chaîne vide, qui ne contient aucun caractère s'écrit `""`.

La concaténation de chaînes de caractères peut s'écrire avec le symbole `^` (« accent circonflexe ») en notation infixe : `"Hello"^" "^"world"^"!"` est égal à notre inévitable et fameux `"Hello world!"`.

Les chaînes de caractères sont comme des « tableaux » de caractères. On peut accéder à chacun de leurs éléments individuellement en utilisant sa position dans la suite : son *indice*. Le premier indice est 0. Pour obtenir la valeur du premier caractère de notre chaîne préférée, on écrit `"Hello world!".[0]`. La valeur obtenue est le caractère `'H'`. De façon générale, on écrit s`.[`i`]` pour obtenir le caractère en $i$ème position dans la chaîne s. On obtient l'exception `Invalid_argument "index out of bounds"` si i est une valeur négative ou trop grande. On peut remplacer un caractère par un autre à une position donnée dans une chaîne. On écrit s`.[`i`] <- c` pour remplacer la caractère en position i dans s par c. Le résultat de cette opération n'est pas la chaîne modifiée, mais la valeur `()` du type `unit`. Chaque caractère d'une chaîne est donc *modifiable*. Ici également, on obtient l'exception `Invalid_argument "index out of bounds"` si i est une valeur négative ou trop grande.

La longueur maximale d'une chaîne de caractères dépend du système utilisé. Elle est donnée par la constante `Sys.max_string_length` fournie par le module `Sys`.

Les fonctions prédéfinies pour les chaînes de caractères sont fournies par le module `String`. Citons :

- `String.length` de type `string -> int` qui donne la longueur (nombre de caractères) de son argument. La dernière position dans une chaîne non vide `s` est `(String.length s) - 1` ;
- `String.create` de type `int -> string` qui crée une chaîne de la longueur passée en argument. Son contenu est totalement arbitraire. On obtient l'exception `Invalid_argument "String.create"` si l'argument est plus grand que la longueur maximale d'une chaîne ou négatif ;
- `String.index` de type `string -> char -> int` telle que `(String.index s c)` donne la position la plus à gauche du caractère `c` dans la chaîne `s` ou l'exception `Not_found` si `c` n'a pas d'occurrence dans `s`.
- `String.sub` de type `string -> int -> int -> string` telle que `(String.sub s i len)` donne une *copie* de la sous-chaîne de `s` de longueur `len` qui commence en `i` ou l'exception `Invalid_argument "String.sub"` si `i` et `len` ne définissent pas une sous-chaîne de `s`.

Nous invitons le lecteur à se référer à la documentation en ligne du langage pour en savoir plus sur les chaînes de caractères.

14 Listes : type `'a list`

On peut directement écrire des valeurs du type `list` en plaçant entre crochets ouvrant (`[`) et fermant (`]`) ses éléments séparés par un point virgule (`;`). Par exemple : `[ "Hello"; " "; "world"; "!" ]` est une liste de quatre chaînes de caractères. Son type s'écrit `string list`. Comme les tableaux, les listes sont *homogènes* : un seul type peut remplacer le paramètre du type `list`. L'écriture `[ 1; true ]` est mal typée, ce n'est pas une valeur licite du langage. On écrit `[]` la liste vide qui ne contient aucun élément. C'est une *valeur immédiate* du langage.

Deux opérations binaires pour les listes sont utilisées en notation infixe :

- `::` de type `'a -> 'a list -> 'a list` telle que `x::xs` construit la liste dont le premier élément est `x` et la suite `xs`. Le type de `x` doit être cohérent avec le type des éléments de `xs`. Un nouvel emplacement mémoire est alloué pour contenir `x` et `xs` reste à sa place. Les valeurs écrites `[ 1; 2; 3 ]` ou `1::2::3::[]` sont égales. Le symbole `::` est un *constructeur* du type des listes, tout comme la constante `[]`. On peut les utiliser pour écrire des motifs de filtrage ;
- `@` de type `'a list -> 'a list -> 'a list` donne la liste résultant de la concaténation de ses deux arguments. Dans le résultat de `(xs1 @ xs2)`, la liste `xs1` est copiée alors que `xs2` reste en place.

Ne confondez pas `::` et `@`. Le type de `::` indique clairement que ses deux arguments ne sont pas de même nature alors qu'ils sont tous deux des listes pour la fonction `@`.

Ainsi, pour ajouter `x` devant la liste `xs`, on écrit `x::xs` et non `x@xs` qui serait mal typé. Si l'on tient absolument à utiliser `@`, on peut écrire `[x]@xs`, mais cela entraîne des calculs supplémentaires inutiles.

Également, il est vain d'espérer obtenir un programme qui ajoute un élément `x` en fin d'une liste `xs` en écrivant `xs::x`. Et tout aussi vain de tenter de construire la liste des deux éléments `x1`, `x2` en écrivant `x1::x2`.

Pour ajouter un élément en fin de liste, on utilisera l'opération de concaténation `@`. Mais attention, cette opération attend deux listes, on écrira donc `xs@[x]` pour ajouter `x` en fin de `xs`, c'est-à-dire, pour concaténer la liste `xs` avec *la liste* qui ne contient que `x` et que l'on peut écrire `[x]`. Écrire `xs@x` serait encore ici incorrect.

Module `List` Les fonctions prédéfinies sur les listes sont fournies par le module `List`. Citons :

- `List.hd` qui donne le premier élément d'une liste non vide : `(List.hd (x::xs))` est égal à `x`. On obtient l'exception `Failure "hd"` si l'argument est la liste vide ;
- `List.tl` qui donne la « suite » d'une liste non vide : `(List.tl (x::xs))` est égal à `xs`. On obtient l'exception `Failure "hd"` si l'argument est la liste vide ;

— `List.length` de type `'a list -> int` donne la longueur de son argument. La liste vide est de longueur 0, mais il est déconseillé d'utiliser la valeur de l'application de `List.length` pour savoir si une liste est vide ou non. Utilisez plutôt la comparaison avec `[]` (toutes les listes vides sont égales entre elles) ou mieux, le *filtrage* ;
— `List.rev` de type `'a list -> 'a list` qui donne une liste dont les éléments sont dans l'ordre inverse de celui qu'ils ont dans son argument (liste *miroir*) ;
— `List.mem` de type `'a -> 'a list -> bool` telle que `(List.mem x xs)` vaut `true` si `x` a une occurrence dans `xs` et `false` sinon ;
— `List.map` de type `('a -> 'b) -> 'a list -> 'b list` telle que `(List.map f xs)` donne la liste dont les éléments sont le résultat de l'application de `f` à chaque élément de `xs`, schématiquement : `(List.map f [ x1; ...; xn ])` est égal à la nouvelle liste `[ (f x1); ...; (f xn) ]` ;
— `List.filter` de type `('a -> bool) -> 'a list -> 'a list` telle que `(List.filter p xs)` donne la liste de tous les éléments `e` de `xs` tels que `(p e)` vaut `true`. L'ordre des éléments est préservé ;
— `List.sort` de type `('a -> 'a -> int) -> 'a list -> 'a list` est une *fonction de tri* pour les listes telle que `(List.sort c xs)` est une liste dont les éléments sont tous ceux de `xs` placés dans l'ordre croissant déterminé par la fonction de comparaison `c`. Cette fonction de comparaison est analogue à celle utilisée par la fonction `Array.sort` (voir *supra* **??**). Le résultat est une nouvelle liste.

Il existe encore beaucoup de fonctions prédéfinies dans le module `List`. Consultez la documentation en ligne du langage pour en savoir plus.

15 Langage de types

Une propriété intéressante des types en OCAML, partagée d'ailleurs avec l'ensemble des langages de la famille ML, est l'existence de *types à paramètres*. C'est ce qui ouvre la possibilté de définir des fonctions polymorphes. En particulier, le type des fonctions est un type à paramètres. Leurs paramètres sont les types des arguments et le type du résultat. Certains paramètres d'un type peuvent être indéterminés (c'est le polymorphisme), on utilise pour les désigner des *variables de type*.

On peut classer les types de OCAML en : types *primitifs*, types *n-uplets*, types *fonctionnels*, types *sommes* et types *enregistrement*.

Chaque type selon qu'il est paramétré ou non s'écrit par un simple nom ou par l'application de son nom aux expressions du langage de type désignant ses paramètres. L'écriture de l'application est en général postfixée, sauf pour l'écriture des types n-uplets et fonctionnels où elle est infixée. Voici la *grammaire* du langage des types de OCAML :

τ	::=	*ident*	type simple
	|	'*ident*	variable de type
	|	τ *ident*	type à un paramètre
	|	(τ,...) *ident*	type à plusieurs paramètres
	|	τ * τ	type n-uplet
	|	τ -> τ	type fonctionnel
	|	(τ)	

Les types primitifs ou prédéfinis Les types de base simples, prédéfinis du langage sont

`bool char int float string unit`

pour les booléens, les caractères (ASCII - ISO 8859-1), les entiers, les nombres à virgule (standard IEEE 754), les chaînes de caractères. Le type `unit` possède une seule valeur écrite `()`.

Le lecteur est invité à consulter les paragraphes de cette annexe consacrés à chacun d'eux.

Les types à paramètres τ `array` et τ `list` sont respectivement le type des tableaux de valeurs de type τ et le type des listes de valeurs de type τ. Les paragraphes **??** et 14 de cette annexe et le chapitre **??** de ce livre sont consacrés aux valeurs de ces types.

Le type τ `option` est uilisé pour les valeurs optionnelles (voir les chapitres **??**, page **??** et **??**, page **??** pour des exemples d'utilisation de valeurs de ce type).

Le type τ `ref` est le type des valeurs *modifiables* (voir les chapitres **??**, paragraphe **??** ainsi que le paragraphe **??** de cette annexe).

Les types des n-uplets sont notés τ_1 `*` τ_2 pour le type des couples de valeurs de type τ_1 et τ_2 ; τ_1 `*` τ_2 `*` τ_3 le type des triplets ; etc. La notation infixe des types n-uplets est stricte : τ_1 `*` τ_2 `*` τ_3 n'est pas égal à τ_1 `*` `(`τ_2 `*` τ_3`)` ni à `(`τ_1 `*` τ_2`)` `*` τ_3.

Les types fonctionnels sont notés τ_1 `->` τ_2. La notation infixe des types fonctionnels est associative à droite : τ_1 `->` `(`τ_2 `->` τ_3`)` est égal à τ_1 `->` τ_2 `->` τ_3 mais différent de `(`τ_1 `->` τ_2`)` `->` τ_3 qui est le type dont l'argument est une fonction de type τ_1 `->` τ_2.

Stricto sensu toute fonction en OCAML est unaire : puisqu'une fonction de type
τ_1 `->` τ_2 `->` τ_3 est une fonction de type τ_1 `->` `(`τ_2 `->` τ_3`)`, c'est-à-dire, une fonction qui a toute valeur de type τ_1 associe une *fonction* de type τ_2 `->` τ_3. L'associativité à droite des types fonctionnels est le pendant de l'associativité à gauche de l'application fonctionnelle :

- si e est de type τ_1 `->` τ_2 `->` τ et e_1 est de type τ_1 alors `(`e e_1`)` est de type τ_2 `->` τ ;
- si e_2 est de type τ_2, puisque `(`e e_1`)` est de type τ_2 `->` τ alors `((`e e_1`)`e_2`)` est de type τ ; c'est-à-dire, en omettant les parenthèses, `(`e e_1 e_2`)` est de type τ.

Ce que résume *l'arbre de typage* suivant :

$$\dfrac{\dfrac{e : \tau_1 \texttt{ -> } \tau_2 \texttt{ -> } \tau \quad e_1 : \tau_1}{\texttt{(}e\ e_1\texttt{)} : \tau_2 \texttt{ -> } \tau} \qquad e_2 : \tau_2}{\texttt{(}e\ e_1\ e_2\texttt{)} : \tau}$$

16 Définitions de type

Les définitions de type sont introduites par le mot clé `type`. On peut définir des *types produits* ou pour des *types sommes* (voir ci-dessous 17). Une définition de type peut faire figurer des *paramètres de type*. Ceux-ci sont notés comme des variables de type : `'a`, `'b`, etc. (voir 15).

Une définition d'un type somme introduit dans les programmes, outre le nom du type, des noms qui serviront à manipuler les valeurs construites pour ce type : noms des *constructeurs* pour les types sommes.

Exemple de définition avec des types produits :

```
type bit = bool
type duet = bit * bit
type quartet = bit * bit * bit * bit
type bitseq = bit list
```

Avec paramètre de type :

```
type 'a triplet = 'a * 'a * 'a
```

Exemple de définition de type somme (arbres binaires) :

```
type 'a btree =
  Empty
| Node of 'a btree * 'a * 'a btree
```

17 Types sommes

Les types sommes sont également appelés types *union* ou *variants*. On les a même parfois appelés *types algébriques* car ils définissent l'ensemble de leurs valeurs comme l'application des *constructeurs* de valeur du type aux expressions donnant la valeur de leurs arguments. Les constructeurs de valeur d'un type somme sont interprétés par le mécanisme d'évaluation des programmes comme des *fonctions d'allocation* qui réservent une place mémoire pour y ranger les valeurs de leurs arguments. La définition d'un type somme consiste à donner la liste des constructeurs possibles des valeurs du type accompagné chacun du type des arguments attendus. En OCAML, le nom d'un constructeur pour un type somme **doit impérativement commencer par une majuscule**. Dans la définition de type, le nom d'un constructeur et les types des arguments attendus sont séparés par le mot clef `of`. Les couples (constructeur,type) de la définition de type sont séparés par une barre verticale (caractère `|`). Un constructeur de type peut être une *valeur atomique*, une constante qui n'attend aucun argument. Lorsqu'un constructeur attend plusieurs arguments, il faut réunir ceux-ci dans un n-uplet. Et, bien entendu, un type somme peut être paramétré. On peut écrire ainsi la forme générale de la définition d'un type somme :

$$\texttt{type (}\alpha\ \texttt{...) }t\texttt{ = }C_1\ [\texttt{of }\tau_1]\ \texttt{...}\ \ \texttt{|}\ C_n\ [\texttt{of }\tau_n]$$

où les crochets indiquent le caractère optionnel de la définition du type des arguments des constructeurs.

Les types sommes sont commodément utilisés pour définir de multiples types de valeurs : les types énumérés (ensemble de constantes) ; les type `union` au sens de C (valeurs des différents types réunis en un seul) ; les structures arborescentes, dont le cas le plus simple est la liste, etc. En OCAML, les valeurs des types sommes se manipulent avec le mécanisme de filtrage décrit au paragraphe 5 de cette annexe.

Le type `'a option` est un type somme paramétré à deux constructeurs prédéfini dans la bibliothèque standard de la façon suivante :

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

Le type `'a list` est un type somme prédéfini avec deux constructeurs adoptant une syntaxe particulière : le constructeur de liste vide s'écrit `[]` ; le constructeur d'ajout d'un élément s'écrit `::` et s'utilise en notation infixe.

18 Valeurs des types somme

Toutes les valeurs de type somme peuvent être exprimées en utilisant simplement les constructeurs définis pour le type. À l'exception notable du type `list`, l'application d'un constructeur, lorsqu'il attend un argument, est toujours préfixe. Une valeur d'un type somme s'écrit donc soit simplement `C0`, lorsque `C0` est un constructeur constant (sans argument) ; soit `(C1 e)` où `e` donne la valeur de l'argument attendu par le constructeur `C1`. Si l'on désire qu'un constructeur utilise plusieurs valeurs pour fabriquer une valeur du type somme, celles-ci sont encloses dans un n-uplet. Par exemple, si `C1` fabrique une valeur du type somme à partir de deux autres valeurs, on écrira `(C1 (`e_1`,`e_2`))`.

Une expression de la forme `(C1 `e_1 e_2`)` où `C1` est un constructeur, ne peut jamais être correcte.

Comme partout en OCAML, on peut alléger l'écriture en omettant quelques parenthèses. Il faut néanmoins être prudent et garder à l'esprit la nature syntaxique particulière de l'application en OCAML (voir 1). Prenons par exemple le type (récursif)

```
type t = Z | S of t
```

Les écritures `Z` et `S Z` sont correctes, mais pas `S S Z` : il faut écrire `S (S Z)`. Et si l'on veut appliquer une fonction `f` à cette dernière expression, il faut l'enclore entre parenthèses et écrire `f (S (S Z))`.