petite référence de CAML

version du document : 2 (rentrée 1996)

Alain BÈGES & Laurent CHÉNO
lycées
CHAMPOLLION (Grenoble) & LOUIS-LE-GRAND (Paris)

Table des matières

1 Types de base

1.1 Unit

Le type `unit` ne contient qu'un élément, noté `()`. On l'utilise pour les fonctions "sans arguments" (en fait, elles en ont un, c'est `()`). Exemple : `quit : unit -> unit` qui s'invoque par `quit() ;;` On l'utilise aussi pour les fonctions qui ne renvoient "rien" (en fait, elles renvoient quelque chose : `()`). Exemple : les fonctions d'impression.

1.2 Booléens

Le type `bool` contient deux valeurs : `true` (vrai) et `false` (faux). Les opérateurs sur les booléens sont : `&&` et `||` qui seront détaillés en 3.2 (ce sont, bien sûr, le ET et le OU), et `not` (la négation). Tous ces opérateurs sont de faible priorité : **on s'astreindra à parenthéser complètement les expressions booléennes.**

1.3 Nombres entiers

Le type `int` correspond aux entiers (signés). Les opérateurs correspondants sont : `+`, `-`, `*`, `/`, `mod`. La division est une division entière ; exemple : `7/3` s'évalue en `2` ; `7 mod 3` donne `1`, c'est le reste de la division précédente. Notons qu'en général on préfèrera, en informatique, ne pas faire trop d'hypothèses sur les choix du compilateur (ou de l'interprète): c'est dire par exemple ici que l'on n'essaiera pas de se rappeler ce qu'en CAML la fonction `mod` renvoie avec un ou deux arguments négatifs.
`succ` et `pred` sont respectivement les fonctions successeur et prédécesseur.
On dispose des comparaisons standards : `< <= = > >= <>` et de la valeur absolue `abs` ainsi que de `min` et `max` qui attendent deux arguments.

1.4 Nombres à virgule ou "flottants"

Le type `float` correspond aux nombres "à virgule", on les reconnaît (paradoxalement) à la présence du point : `1.2e6` correspond à $1{,}2\,10^6$. Attention: `1` est un `int` alors que `1.` est un `float`.
On dispose des opérateurs courants : `+. -. *. /.` et des comparaisons : `<. <=. =. >. >=. <>.`
Les fonctions transcendantes s'orthographient : `sin cos tan asin acos atan sqrt log exp`. Le logarithme (`log`) est népérien et `sqrt` désigne la racine carrée *(square root)*. L'élévation à la puissance s'obtient par `power : float -> float -> float` ; la valeur absolue par `abs_float`.
Enfin, les conversions entiers/flottants sont
`int_of_float : float -> int` et `float_of_int : int -> float`
qui font ce qu'elles peuvent.

1.5 Caractères

Le type `char` correspond aux caractères. On les écrit `'a' 'b'` ou encore `';'` par exemple. Les caractères spéciaux courants sont : `'\\'` et `'\''` et `'\r'` et `'\t'` pour, respectivement, le backslash, le backquote, le retour-chariot et la tabulation.
Les opérateurs de comparaison disponibles sont `=` et `<>`
Pour les conversions on dispose de
`int_of_char : char -> int` et `char_of_int : int -> char`
qui s'appuient sur le codage ASCII des caractères.

1.6 Chaînes de caractères

Le type `string` correspond aux chaînes de caractères. La chaîne `abc"def` est notée `"abc\"def"` ; les caractères spéciaux vus précédemment sont autorisés.
Les opérateurs correspondants sont `^` pour la concaténation et `=` et `<>` pour les comparaisons.

La bibliothèque standard fournit entre autres les fonctions suivantes :

- `string_length : string -> int` renvoie la longueur d'une chaîne ;
- `nth_char : string -> int -> char` renvoie le n-ième caractère d'une chaîne (la numérotation commence à zéro) ; depuis la version 0.7, on dispose heureusement de la notation `s.[i]` qui abrège `nth_char s i` ;
- `set_nth_char : string -> int -> char -> unit` remplace le n-ième caractère d'une chaîne par un autre ; depuis la version 0.7, on dispose heureusement de la notation `s.[i] <- c` qui abrège `set_nth_char s i c` ;
- `sub_string : string -> int -> int -> string` renvoie une nouvelle chaîne : par exemple, l'appel `sub_string "abcdef" 2 3` renvoie `"cde"` qui commence au 2^{e} caractère et est de longueur 3 ;
- `make_string : int -> char -> string` crée une chaîne dont on précise la taille et le caractère de remplissage.

Les fonctions de comparaison sont toutes du type `string -> string -> bool` ; elles s'appuient sur l'ordre lexicographique et se notent : `eq_string` (égalité), `neq_string` (inégalité), `ge_string` (supérieur ou égal), `gt_string` (supérieur strict), `le_string` (inférieur ou égal), `lt_string` (inférieur strict).

1.7 Couples et multiplets

Les couples correspondent au produit cartésien de deux types.
Exemple : `(1,"asd")` est du type `int * string`. Bien sûr, on dispose de même des multiplets comme `(false, 3, "asd")` du type `bool * int * string`.
La bibliothèque standard fournit entre autres les fonctions suivantes :

- `fst : 'a * 'b -> 'a` renvoie le premier élément d'un couple ;
- `snd : 'a * 'b -> 'b` renvoie le second ;
- `split : ('a * 'b) list -> 'a list * 'b list` prend une liste de couples et renvoie le couple des listes formées des premiers et des seconds éléments ;
- `combine : 'a list * 'b list -> ('a * 'b) list` effectue l'opération inverse ;
- `map_combine : ('a * 'b -> 'c) -> 'a list * 'b list -> 'c list` fait ce que l'on pense :

```
#map_combine (function (x,y) -> x*y) ([1; 2; 3], [4; 5; 6]) ;;
- : int list = [4; 10; 18]
```

- `do_list_combine : ('a * 'b -> 'c) -> 'a list * 'b list -> unit` fait de même, mais ne renvoie rien : n'a d'intérêt que si la fonction passée en premier argument réalise des effets de bord.

1.8 Listes

Une liste se note `[x_1; x_2; x_3; ...; x_N]` où les objets `x_i` sont tous d'un même type `'a`, le type de la liste est alors `'a list`. La liste vide se note `[]` ; l'opérateur de construction des listes est `::` qu'on lit *quatre points*, ainsi `1 :: 2 :: 3 :: []` ou `1 :: 2 :: [3]` ou `1 :: [2; 3]` valent `[1; 2; 3]`, du type `int list`. On dit que l'opérateur `::` est associatif à droite. Les deux fonctions d'accès fondamentales sont `hd` (pour ***hea**d*) qui renvoie la tête de la liste, et `tl` (pour ***tai**l*) qui renvoie la queue de la liste. Ces deux fonctions déclenchent `Failure "hd"` ou `Failure "tl"` si on tente de les appliquer à la liste vide. La plupart du temps, on leur préfèrera le filtrage.
L'opérateur de concaténation des listes se note `@` ; son exécution ne se fait pas en temps constant mais linéaire en la taille de la première liste.

La bibliothèque standard fournit entre autres les fonctions suivantes :

- `list_length : 'a list -> int` renvoie la longueur de la liste ;
- `rev : 'a list -> 'a list` fabrique une copie renversée de la liste passée en argument, ainsi `rev [1; 2; 3]` vaut `[3; 2; 1]` ;
- `map : ('a -> 'b) -> 'a list -> 'b list` construit la liste des résultats de l'application de la fonction donnée en premier argument à chacun des éléments de la liste passée en second argument ;
- `do_list : ('a -> 'b) -> 'a list -> unit` fait comme `map` mais sans renvoyer aucun résultat : n'est intéressante que pour les effets de bord ;
- `for_all : ('a -> bool) -> 'a list -> bool` dit si le prédicat (fonction à valeur booléenne) passé en premier argument est satisfait par tous les éléments de la liste passée en second argument ;
- `exists : ('a -> bool) -> 'a list -> bool` dit si le prédicat (fonction à valeur booléenne) passé en premier argument est satisfait par au moins un élément de la liste passée en second argument ;
- `mem : 'a -> 'a list -> bool` dit si l'élément passé en premier argument est élément de la liste passée en second argument ;
- `except : 'a -> 'a list -> 'a list` renvoie la liste passée en deuxième argument privée du premier de ses éléments égal à l'objet passé en premier argument (on ne supprime donc au plus qu'un élément) ;
- `substract : 'a list -> 'a list -> 'a list` renvoie la liste des éléments de la première liste qui ne figurent pas dans la seconde ;
- `union : 'a list -> 'a list -> 'a list` renvoie la deuxième liste augmentée des éléments de la première qui n'y figurent pas déjà ;
- `intersect : 'a list -> 'a list -> 'a list` renvoie une copie de la première liste privée des éléments qui ne figurent pas dans la seconde ;
- `index : 'a -> 'a list -> int` renvoie la position de la première occurrence du premier argument dans la liste passée en second argument, la numérotation commençant à 0 ;
- `assoc : 'a -> ('a * 'b) list -> 'b` gère les listes associatives : par exemple

  ```
  assoc 2 [(1,‘a‘); (2,‘b‘) ; (2,‘c‘) ; (3,‘d‘)]
  ```

  vaut `‘b‘`. La fonction `assoc` déclenche l'exception `Not_found` en cas d'échec. La bibliothèque `map` est plus générale et plus efficace, on s'y reportera utilement ;
- `it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` permet des manipulations complexes : l'appel `it_list f a [ b1 ; ... ; bn ]` renvoie `f (... (f (f a b1) b2) ...) bn` ;
- `list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` permet des manipulations complexes : l'appel `list_it f [ a1 ; ... ; an ]` renvoie `f a1 (f a2 (... (f an b) ...))`.

2 Types construits

2.1 Types produit

Comme les multiplets, les types produit sont des produits cartésiens, mais ils permettent en outre de nommer les projections, qu'on appelle des *champs*.

Par exemple
`type individu = { Nom : string ; Prénom : string ; Sécu : int }`
définit un type produit qui comporte trois champs. L'accès aux différents champs (c'est-à-dire les différentes projections) répond à la syntaxe suivante : si `héros` est un objet de type `individu`, on accède à ses projections par `héros.Nom`, `héros.Prénom`, `héros.Sécu`. Un objet de ce type est par exemple
`{ Nom = "Lanturlu" ; Prénom = "Anselme" ; Sécu = 12345 }`.
On notera que le filtrage ne nécessite pas l'explicitation de tous les champs, alors qu'elle est indispensable lors de la définition d'un objet.

2.2 Types somme

Mathématiquement parlant, il s'agit ici de sommes disjointes. Par exemple
`type booléen = Vrai | Faux`
est une définition parfaitement satisfaisante des booléens, `Vrai` et `Faux` sont des *constructeurs* du type. Attention : la définition suivante n'est pas licite : `type nombre = int | float` ; il faut passer par l'usage des constructeurs, en écrivant par exemple
`type nombre = Entier of int | Flottant of float`.
Un objet de ce type se note `Entier(123)` ou `Flottant(3.14159)`. Le filtrage permet de discriminer les valeurs.

2.3 Types paramétrés

Toute définition de type construit peut être paramétrée par des variables de types.
Par exemple `type 'a numéroté = int * 'a`, ou, pour définir un type analogue aux couples,
`type ('a,'b) paire = {Premier : 'a ; Second : 'b}`.
Voici un exemple avec un type somme :
`type ('a,'b) simple_ou_double = Simple of 'a | Double of 'a * 'b`
Ce type permet de décrire quelque chose comme l'union disjointe de A et $A \times B$.

2.4 Types (mutuellement) récursifs

Donnons quelques exemples :
`type 'a liste = Nil | Cons of 'a * 'a liste`
permettrait de définir un type paramétré semblable au type prédéfini Caml `'a list`. Quant aux types

```
type 'a arbre = Nil | Nœud of 'a nœud_interne
 and 'a nœud_interne = { Clé : 'a ; Fils_droit : 'a arbre ; Fils_gauche : 'a arbre }
```

ils fournissent une description d'un arbre binaire.

2.5 Abréviations de type

Caml nous permet de définir des synonymes pour des types simples, par exemple
`type point3d == float * float * float`
permet d'invoquer par la suite cette abréviation pour définir un nouveau type, ou pour une coercion de type. Plus simplement, cela offre une interprétation lisible d'un type complexe.

3 Structures de contrôle

3.1 Séquencement

Une séquence d'expressions est constituée d'une suite d'expressions séparées par des point-virgules. Sa valeur est celle de la dernière expression évaluée. Cela n'a d'intérêt que si les premières expressions réalisent des effets de bord. Le plus souvent, on ensère une séquence par le couple `begin-end`, ou, à la rigueur, par une paire de parenthèses.

3.2 Opérateurs AND et OR

Les opérateurs booléens `&&` et `||` ne peuvent pas être définis en tant que fonctions. En effet, ils n'évaluent leurs arguments qu'au fur et à mesure de leur besoin. Par exemple,
`true || ((3/0) > 0)`
ne déclenche pas l'erreur (exception) `Division_by_zero` mais vaut `true` ; de même
`false && ((3/0) > 0)`
ne déclenche pas d'exception mais vaut `false`.

3.3 Conditionnelles

Une expression conditionnelle est de la forme

`if` *`expression_test`* `then` *`expression_1`* `else` *`expression_2`*

où *`expression_test`* est de type `bool` et où les deux expressions *`expression_1`* et *`expression_2`* doivent être d'un même type, qui sera celui de l'expression conditionnelle.
Dans le seul cas où le type de *`expression_1`* est `unit` la clause `else` peut être omise.

3.4 Filtrage

3.4.1 L'expression `match`

Sa syntaxe est :

```
match expression with
  |motif_1 -> expr_1
  |motif_2 -> expr_2
   ...
  |motif_N -> expr_N
```

Caml tente de filtrer *`expression`* avec *`motif_1`* ; en cas de succès, l'expression `match` complète vaut *`expr_1`*. En cas d'échec, les motifs suivants sont essayés dans l'ordre. Si aucun motif ne filtre *`expression`*, l'exception `Match_failure` est déclenchée. Il faut que toutes les expressions *`expr_i`* soient du même type, qui est celui de l'expression `match` toute entière.
Par exemple, le filtrage typique sur les listes est :

```
match liste with
   | [] -> ...
   | tête :: queue -> ...
```

3.4.2 Motifs de filtrage

Les motifs sont définis par :

motif ≡

_	*constructeur-constant*
identificateur	*constructeur-non-constant motif*
constante	(*motif*)
[]	*motif* | *motif*
[*motif* ; ... ; *motif*]	*motif* as *identificateur*
motif :: *motif*	(*motif* : *expr-de-type*)
motif, ..., *motif*	
{ *étiq_1* = *motif_1* ; ... ; *étiq_n* = *motif_n* }	

- `_` filtre tout et n'importe quoi ;
- un identificateur filtre tout mais en effectuant la liaison ;
- une constante ne filtre qu'elle-même ;
- `[]` filtre la liste vide ;
- une liste de motifs filtre, motif par motif, les éléments d'une liste ;
- *motif_1* `::` *motif_2* filtre une liste dont la tête est filtrée par *motif_1* et la queue par *motif_2* ;
- un multiplet de motifs filtre, motif par motif, les éléments d'un multiplet ;
- `{` *étiq_1* `=` *motif_1* `;` `...` `;` *étiq_n* `=` *motif_n* `}` filtre champ par champ une expression de type produit ;
- un constructeur constant ne filtre que lui-même ;
- *constructeur-non-constant motif* filtre une expression de type somme ;
- on peut parenthéser les motifs ;
- *motif_1* `|` *motif_2* filtre ce que filtrent *motif_1* ou *motif_2*, qui n'ont pas le droit de procéder à des liaisons ;
- *motif* `as` *identificateur* filtre ce que filtre *motif* et introduit la liaison de l'expression filtrée avec *identificateur* ;
- `(` *motif* `:` *expr-de-type* `)` filtre une expression filtrée par *motif* de type *expression-de-type*.

3.4.3 Expressions fonctionnelles

Une fonction se définit naturellement par filtrage :

```
function
    | motif_1 -> expr_1
      ...
    | motif_N -> expr_N
```

Si `'a` est le type le plus général des expressions filtrées par les *motif_i* et si `'b` est le type commun aux *expr_i*, le type de cet objet (fonctionnel) se note `'a -> 'b`.
Remarques :

- l'air de rien `function x -> x` fait déjà intervenir un filtrage ;
- l'expression

```
match expression with
  |motif_1 -> expr_1
   ...
  |motif_N -> expr_N
```

  est en fait équivalente à

```
( function |motif_1 -> expr_1
           |motif_2 -> expr_2
            ...
           |motif_N -> expr_N ) ( expression )
```

- inversement,

```
function |motif_1 -> expr_1
         |motif_2 -> expr_2
          ...
         |motif_N -> expr_N
```

peut s'écrire

```
function x -> match x with
  |motif_1 -> expr_1
  |motif_2 -> expr_2
   ...
  |motif_N -> expr_N
```

- La forme syntaxique

  `function` *motif_1* `->` ... `->` `function` *motif_N* `->` *expression*

  peut s'abréger en

  `fun` *motif_1 ... motif_N* `->` *expression*

- `->` est associatif à droite, c'est-à-dire que le type `'a -> 'b -> 'c` doit s'entendre comme étant le type `'a -> ( 'b -> 'c )`
- CAML comprend l'expression `x y z` comme étant `(x y) z` et non pas `x (y z)`.

3.4.4 Expressions `let` locales

L'expression

```
let motif_1 = expr_1
and ...
and motif_N = expr_N
in expression
```

opère des liaisons (par filtrage) qui ne seront que locales à l'évaluation de *expression*. On notera que toutes les *expr_i* sont évaluées avant la première liaison. Elle pourrait s'écrire

`(` `fun` *motif_1 ... motif_N* `->` *expression* `)` *expr_1 ... expr_N*

En écrivant `let rec` au lieu de `let`, on autorise des définitions mutuellement récursives.

3.4.5 Expressions `let` globales : définitions

L'instruction

```
let motif_1 = expr_1
and ...
and motif_N = expr_N
```

opère des liaisons (par filtrage) qui seront globales : on ajoute de nouvelles définitions à l'interprète. On notera que toutes les *expr_i* sont évaluées avant la première liaison.
Ainsi :

```
#let x = 0 and y = 1 ;;
x : int = 0
y : int = 1
#let x = y and y = x ;;
x : int = 1
y : int = 0
```

En écrivant `let rec` au lieu de `let`, on autorise des définitions mutuellement récursives.

3.4.6 Du sucre syntaxique

La forme syntaxique
`let f` *`motif1 ... motifN`* `=` *`expression`*
est comprise par Caml comme
`let f = fun` *`motif1 ... motifN`* `->` *`expression`*.

3.5 Boucles

On dispose de deux types de boucles. La boucle `while` obéit à la syntaxe suivante

`while` *`expression_test`* `do` *`expression`* `done`

La boucle `for` obéit à la syntaxe suivante

`for` *`identificateur`* `=` *`expr_1`* `to` *`expr_2`* `do` *`expression`* `done`

où *`expr_1`* et *`expr_2`* sont du type `int`. L'identificateur est implicitement local. Caml connaît aussi la version `downto`.

3.6 Exceptions

Il y a deux façons de définir des exceptions :
`exception` *`identificateur`* et `exception` *`identificateur`* `of` *`expression-de-type`*.
Les définitions d'exceptions ajoutent de nouveaux constructeurs au type somme prédéfini `exn` des valeurs "exceptionnelles".
On déclenche une exception en invoquant la fonction `raise : exn -> 'a`. On rattrape une exception par la structure de contrôle `try`, dont la syntaxe est :

```
try expression with|motif_1 -> expr_1
                   |...
                   |motif_N -> expr_N
```

L'évaluation de *`expression`* peut déclencher une exception, celle-ci est éventuellement interceptée par le filtrage du `try` (sans quoi elle se propage), l'évaluation de *`expression`* se poursuit alors par l'évaluation du *`expr_i`* adéquat.
Il peut être intéressant d'utiliser des exceptions non constantes pour récupérer l'état du calcul au moment du déclenchement de l'exception.
Par exemple :

```
#exception Trouvé of int ;;
Exception Trouvé defined.
#let trouve_indice prédicat =
    let rec trouve n = function
        | [] -> raise ( Failure "absent" )
        | a :: q -> if prédicat(a) then raise (Trouvé n)
                    else trouve (n+1) q
```

```
    in trouve 0 ;;
trouve_indice : ('a -> bool) -> 'a list -> 'b = <fun>
#trouve_indice (function x -> (x mod 2) = 0) [ 3 ; 5 ; 8 ; 7 ] ;;
Uncaught exception: Trouvé 2
```

L'exception `Failure of string` est prédéfinie.
`failwith` *chaîne* est équivalent à `raise ( Failure` *chaîne* `)`.

4 Effets de bord : types mutables

4.1 Types mutables

Caml permet de définir des valeurs mutables de type produit en introduisant le mot-clef `mutable` devant un (ou plusieurs) champ d'un type produit ; on dispose alors de l'opération d'affectation, qui permet de modifier physiquement les valeurs correspondantes. Exemple :

```
#type point2d = {mutable X : int; mutable Y : int} ;;
Type point2d defined.
#let p = {X=3; Y=3};;
p : point2d = {X=3; Y=3}
#p.X ;;
- : int = 3
#p.X <- 14 ;;
- : unit = ()
#p;;
- : point2d = {X=14; Y=3}
```

4.2 Autre type mutable : les références

Si `'a` est un type, le type `'a ref` est le type des références de ce type. On dispose des deux opérateurs `!` de déréférencement, et `:=` d'affectation. Voici une petite session Caml pour en montrer l'usage :

```
#let a = ref 1 ;;
a : int ref = ref 1

#a := ((!a) + 1) ;;
- : unit = ()

#a ;;
- : int ref = ref 2

#!a ;;
- : int = 2
```

Voici comment pourraient être définies les références :

```
#type 'a référence = { mutable Référence : 'a } ;;
Type référence defined.

#let référence x = { Référence = x } ;;
référence : 'a -> 'a référence = <fun>

#let déréf {Référence = x} = x ;;
déréf : 'a référence -> 'a = <fun>
```

```
#let reçoit r x = r.Réference <- x ;;
reçoit : 'a réference -> 'a -> unit = <fun>

##infix "reçoit" ;;

#let a = référence 1 ;;
a : int réference = {Réference=1}

#a reçoit ((déréf a) + 1) ;;
- : unit = ()

#a ;;
- : int réference = {Réference=2}

#déréf a ;;
- : int = 2
```

4.3 Autre type mutable : les vecteurs

Un vecteur se note `[| x_1; x_2; x_3; ...; x_N |]` où les objets `x_i` sont tous d'un même type `'a`, le type du vecteur est alors `'a vect`. Si `v : 'a vect` est un vecteur, on accède à son `i`-ème élément par `v.(i)`, la numérotation commençant à 0 ; la structure est mutable : on modifie en place le `i`-ème élément en écrivant `v.(i) <-` *`expression`*. La bibliothèque standard fournit entre autres les fonctions suivantes :

- `vect_length : 'a vect -> int` renvoie la taille du vecteur ;
- `make_vect : int -> 'a -> 'a vect` fabrique un nouveau vecteur dont la taille est fixée par le premier argument, le second initialisant chacun de ses éléments ;
- `list_of_vect : 'a vect -> 'a list` construit la liste des éléments du vecteur ;
- `vect_of_list : 'a list -> 'a vect` réalise l'opération inverse.

5 Effets de bord : entrées / sorties

5.1 Entrées/sorties clavier/écran

La bibliothèque standard fournit entre autres les fonctions suivantes d'affichage à l'écran :

- `print_char : char -> unit` affiche un caractère à l'écran ;
- `print_string : string -> unit` affiche une chaîne de caractères à l'écran ;
- `print_int : int -> unit` affiche la représentation décimale d'un entier à l'écran ;
- `print_float : float -> unit` affiche la représentation décimale d'un flottant à l'écran ;
- `print_newline : unit -> unit` affiche un saut de ligne à l'écran.

Signalons au passage l'existence de quatre fonctions de conversion : `string_of_int : int -> string` et `string_of_float : float -> string`, et leurs inverses
`int_of_string : string -> int` et `float_of_string : string -> float`.
La bibliothèque standard fournit entre autres les fonctions suivantes d'entrée au clavier :

- `read_line : unit -> string` attend l'entrée au clavier d'une chaîne de caractères limitée par un saut de ligne, renvoie cette chaîne sans le saut de ligne ;

- `read_int : unit -> int` est la composée de `int_of_string` sur `read_line` ;
- `read_float : unit -> float` est la composée de `float_of_string` sur `read_line` ;

5.2 Fichiers texte

Les fichiers texte utilisent le format ASCII standard et sont lisibles par tous les éditeurs de texte.
La bibliothèque standard fournit entre autres les fonctions d'entrée suivantes :

- `open_in : string -> in_channel` prend le nom du fichier à ouvrir et fournit un objet de type `in_channel` via lequel se feront toutes les opérations d'entrée sur ce fichier ;
- `input_char : in_channel -> char` lit un caractère sur le fichier ;
- `input_line : in_channel -> string` lit sur le fichier une chaîne de caractères limitée par un saut de ligne, renvoie cette chaîne sans le saut de ligne ;
- `close_in : in_channel -> unit` ferme le fichier en lecture.

Toute lecture qui tente d'aller au delà de la fin du fichier déclenche l'exception `End_of_file`.
La bibliothèque standard fournit entre autres les fonctions de sortie suivantes :

- `open_out : string -> out_channel` prend le nom du fichier à ouvrir et fournit un objet de type `in_channel` via lequel se feront toutes les opérations de sortie sur ce fichier ;
- `output_char : out_channel -> char -> unit` écrit un caractère sur le fichier ;
- `output_string : out_channel -> string -> unit` écrit une chaîne de caractères sur le fichier ;
- `close_out : out_channel -> unit` ferme le fichier en écriture.

5.3 Fichiers binaires

Les fichiers binaires utilisent un format particulier à Caml pour représenter les valeurs du langage, et ne sont lisibles que par des programmes Caml.
La bibliothèque standard fournit entre autres les fonctions d'entrée suivantes :

- `open_in_bin : string -> in_channel` prend le nom du fichier binaire à ouvrir et fournit un objet de type `in_channel` via lequel se feront toutes les opérations d'entrée sur ce fichier ;
- `input_value : in_channel -> 'a` lit une valeur Caml sur le fichier ;
- `close_in : in_channel -> unit` ferme le fichier en lecture.

L'exception `End_of_file` est déclenchée dans les mêmes conditions que précédemment.
La bibliothèque standard fournit entre autres les fonctions de sortie suivantes :

- `open_out_bin : string -> out_channel` prend le nom du fichier binaire à ouvrir et fournit un objet de type `in_channel` via lequel se feront toutes les opérations de sortie sur ce fichier ;
- `output_value : out_channel -> 'a -> unit` écrit une valeur Caml sur le fichier ;
- `close_out : out_channel -> unit` ferme le fichier en écriture.

6 Effets de bord : la bibliothèque graphique

La bibliothèque graphique n'est accessible qu'après l'incantation `#open "Graphics" ;;` On ouvre la fenêtre graphique par l'incantation `open_graph "highest"` qui renvoie `()`.
On dispose alors entre autres des fonctions et types suivants :

- le type `color` est prédéfini comme un synonyme de `int`. La fonction

  `rgb : int -> int -> int -> color`

  permet de décrire une couleur par son codage RVB. Les couleurs `black white red green blue yellow cyan magenta` sont prédéfinies ;
- `set_color : color -> unit` fixe la couleur courante des tracés ;
- `clear_graph : unit -> unit` efface la fenêtre graphique ;
- `size_x : unit -> int` et `size_y : unit -> int` renvoient la taille de la fenêtre graphique ;
- `plot : int -> int -> unit` affiche un point ;
- `moveto : int -> int -> unit` déplace le point courant sans dessiner ;
- `lineto : int -> int -> unit` trace un segment entre le point courant et le point spécifié qui devient le nouveau point courant ;
- `current_point : unit -> int * int` renvoie la position du point courant ;
- `draw_circle : int -> int -> int -> unit` attend les coordonnées du centre et le rayon du cercle à tracer ;
- `fill_rect x y largeur hauteur` peint un rectangle ;
- `fill_circle x y rayon` peint un disque ;
- `wait_next_event [ Button_down ; Key_pressed ]` attend sans rien faire que l'utilisateur déclenche un événement clavier ou souris, peu importe la valeur renvoyée ;
- `close_graph : unit -> unit` supprime la fenêtre graphique.

Annexes

A Piles

CAML offre une gestion de pile grâce à la bibliothèque `stack`.
On pourra, au choix, pour utiliser une fonction `f` de cette bibliothèque l'appeler directement en la nommant `stack__f`, ou bien rendre accessibles toutes les fonctions de la bibliothèque en une fois, par l'incantation `#open "stack" ;;` qui permet d'omettre la référence au nom de la bibliothèque : l'appel de la fonction est alors simplement `f`.
La bibliothèque offre les type et fonctions suivantes :

- `type 'a t` est le type des piles contenant des éléments du type `'a` ;
- `exception Empty` est déclenchée quand on tente d'appliquer `pop` à une pile vide ;
- `new : unit -> 'a t` crée une nouvelle pile, vide au départ ;
- `push : 'a -> 'a t -> unit` empile ;
- `pop : 'a t -> 'a` dépile ;
- `clear : 'a t -> unit` vide la pile ;
- `length : 'a t -> int` renvoie la taille de la pile ;
- `iter : ('a -> 'b) -> 'a t -> unit` applique tour à tour son premier argument (une fonction) à chacun des éléments présents dans la pile, sans la modifier.

B Files d'attente

CAML offre une gestion de file d'attente grâce à la bibliothèque `queue`.
On pourra, au choix, pour utiliser une fonction `f` de cette bibliothèque l'appeler directement en la nommant `queue__f`, ou bien rendre accessibles toutes les fonctions de la bibliothèque en une fois, par l'incantation `#open "queue" ;;` qui permet d'omettre la référence au nom de la bibliothèque : l'appel de la fonction est alors simplement `f`.
La bibliothèque offre les type et fonctions suivantes :

- `type 'a t` est le type des queues contenant des éléments du type `'a` ;
- `exception Empty` est déclenchée quand on tente d'appliquer `take` à une queue vide ;
- `new : unit -> 'a t` crée une nouvelle queue, vide au départ ;
- `add : 'a -> 'a t -> unit` ajoute un élément à une queue ;
- `take : 'a t -> 'a` retire un élément de la queue ;
- `peek : 'a t -> 'a` ne modifie pas la queue mais renvoie l'élément que renverrait `take` ;
- `clear : 'a t -> unit` vide la queue ;
- `length : 'a t -> int` renvoie la taille de la queue ;
- `iter : ('a -> 'b) -> 'a t -> unit` applique tour à tour son premier argument (une fonction) à chacun des éléments présents dans la queue, sans la modifier.

C Tables d'association, ou dictionnaires

CAML offre une gestion de dictionnaire grâce à la bibliothèque `map`.
On pourra, au choix, pour utiliser une fonction `f` de cette bibliothèque l'appeler directement en la nommant `map__f`, ou bien rendre accessibles toutes les fonctions de la bibliothèque en une fois, par l'incantation `#open "map" ;;` qui permet d'omettre la référence au nom de la bibliothèque : l'appel de la fonction est alors simplement `f`.
Une façon simple de gérer les dictionnaires est d'utiliser des listes associatives, listes d'éléments de la forme `(clé,valeur)`. On écrit alors la recherche grâce à la fonction `assoc` définie par

```
let rec assoc clé table = match table with
    | (c,v) :: q -> if c = clé then v else assoc clé q
    | [] -> raise Not_found ;;
```

La représentation choisie par CAML assure une recherche en temps logarithmique plutôt que linéaire. La bibliothèque offre les type et fonctions suivantes :

- `type ('a,'b) t` est le type des dictionnaires dont les clés sont du type `'a` et les valeurs du type `'b` ;
- `empty : ('a -> 'a -> int) -> ('a,'b) t` crée un dictionnaire vide, la fonction passée en argument permet de définir un ordre sur l'ensemble des clés, puisque, appliquée à deux clés c_1 et c_2 elle renverra 0 si $c_1 = c_2$, un entier négatif si $c_1 < c_2$ et un entier positif si $c_1 > c_2$. Pour des clés entières on choisira souvent `empty (prefix -)`, pour le cas général, on pourra utiliser `empty eq__compare` ;
- `add : 'a -> 'b -> ('a,'b) t -> ('a,'b) t` ajoute un couple `(clé,valeur)` à un dictionnaire ;
- `find : 'a -> ('a,'b) t -> 'b` renvoie la valeur associée à la clé, ou déclenche `Not_found` s'il n'y en a pas ;
- `remove : : 'a -> ('a,'b) t -> ('a,'b) t` supprime un élément du dictionnaire ;
- `iter : ('a -> 'b -> 'c) -> ('a,'b) t -> unit` applique tour à tour la fonction à chacun des couples `(clé,valeur)` présents dans le dictionnaire, qui n'est pas modifié.

D Ensembles

CAML offre une gestion des ensembles totalement ordonnés grâce à la bibliothèque `set`.
On pourra, au choix, pour utiliser une fonction `f` de cette bibliothèque l'appeler directement en la nommant `set__f`, ou bien rendre accessibles toutes les fonctions de la bibliothèque en une fois, par l'incantation `#open "set" ;;` qui permet d'omettre la référence au nom de la bibliothèque : l'appel de la fonction est alors simplement `f`.
La définition d'une structure de données efficace pour toutes les opérations habituelles sur les ensembles totalement ordonnés est un problème difficile, pour lequel on ne connaît pas de réponse tout à fait satisfaisante.
Il faudra bien sûr expliciter à CAML l'ordre utilisé : on fournira donc toujours une *fonction d'ordre*, de type `'a -> 'a -> int`, qui, pour deux arguments x et y, rendra 0 si (et seulement si) $x = y$, un entier positif (strictement) si $x > y$ et négatif (strictement) si $x < y$.
Les concepteurs de CAML nous proposent, par la bibliothèque `set`, une implémentation relativement efficace qui se décline grâce au jeu suivant de fonctions (qui n'est pas ici décrit de façon exhaustive) :

- `'a t` est le type des ensembles d'éléments de type `'a` ;
- `empty : ('a -> 'a -> int) -> 'a t` crée un ensemble vide ordonné par la fonction d'ordre passée en argument ;
- `is_empty : 'a t -> bool` dit si un ensemble est vide ;

- `mem : 'a -> 'a t -> bool` est le test d'appartenance ;
- `add : 'a -> 'a t -> 'a t` permet d'ajouter un élément à un ensemble ;
- `remove : 'a -> 'a t -> 'a t` permet d'ôter un élément d'un ensemble ;
- `choose : 'a t -> 'a` renvoie un élément de l'ensemble, ou déclenche l'exception `Not_found` si l'ensemble est vide ;
- `union`, `inter`, `diff`, `equal` réalisent les fonctions standard sur les ensembles ;
- `elements` renvoie la liste des éléments d'un ensemble ;
- `fold : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b` permet de combiner les résultats d'une fonction sur les éléments d'un ensemble. Ainsi, `fold f s a` calcule `(f xN ... (f x2 (f x1 a)) ...)`, où `x1 ... xN` sont les éléments de l'ensemble `s`, dans un ordre non précisé ;
- `iter : ('a -> 'b) -> 'a t -> unit` applique son premier argument (qui est une fonction) à chacun des éléments de l'ensemble désigné par son second argument ; ne renvoie jamais que `()`.

E Calculs en multi-précision

CAML offre une bibliothèque de fonctions pour les calculs en multi-précision: il s'agit de la bibliothèque `num`, qu'on ouvrira par l'incantation `#open "num" ;;`.
La bibliothèque offre les types, opérateurs et fonctions suivantes (liste non exhaustive):

— `type num` est le type des nombres en multi-précision: ce sont des entiers ou des rationnels ;

— `+/ -/ */ // **/` sont les opérateurs qui réalisent l'addition, la soustraction, la multiplication, la division et l'élévation à la puissance;

— `=/ </ <=/ >/ >=/ <>/` sont les opérateurs qui réalisent les comparaisons standard;

— `minus_num` calcule l'opposé;

— `quo_num` et `mod_num` calculent quotient et reste dans la division euclidienne;

— `max_num min_num abs_num floor_num ceiling_num` calculent ce qu'elles disent;

— `string_of_num num_of_string int_of_num num_of_int float_of_num` réalisent les conversions nécessaires (voire suffisantes).