Pascal Lienhardt

L2

Bases en algorithmique et en programmation

Cours et exercices corrigés

ellipses

Chapitre 1

Préliminaires

L'étude des notions présentées dans ce livre suppose que le lecteur dispose de connaissances [1] sur les notions de *type, variable, fonction, conditionnelle, récursivité, itérativité*. Celles-ci sont rappelées succinctement en section 1.1, dans le langage utilisé par la suite. Les exemples d'exécution utilisent l'interpréteur `ocaml` (*cf.* `www.ocaml.org`). La démarche classique de *conception et de développement d'un algorithme* suivie dans ce livre est présentée en section 1.2 page 29. La notion de *type abstrait*, utilisée en particulier dans les chapitres 3, 4 et 5, est explicitée en section 1.3 page 36. Les types et fonctions utilitaires que nous utilisons pour la manipulation des ensembles sont présentés en section 1.4 page 51.

1.1 Notions de base

1.1.1 Types et opérations de base

type	**exemples de valeurs**	**remarques**
`int`	`0`, `-5`, `222`	entiers compris entre -2^{62} et $2^{62}-1$
`float`	`.2`, `-3.14`, `5.2e3`	flottants
`char`	`'a'`, `'?'`, `'1'`	caractères
`string`	`"Hello World!"`	chaînes de caractères
`bool`	`true`, `false`	booléens
`unit`	`()`	valeur unique « vide »

type	**opérations**	**remarques**
`int`	`+`, `-`, `*`, `/`, `mod`	`mod` : reste de la division entière
`float`	`+.`, `-.`, `*.`, `/.`, `**`	`**` : exponentiation
`char`		pas d'opérations

1. Le lecteur voulant acquérir ces connaissances peut consulter par exemple l'ouvrage [7].

`string`	`^`	concaténation
`bool`	`&&`, `||`, `not`	respectivement *et*, *ou* et *non* logiques

Voici quelques exemples d'expressions et de valeurs : la première ligne contient l'expression saisie [1] dans l'interpréteur `ocaml`, la deuxième ligne contient la réponse de l'interpréteur, c.-à-d. la valeur de l'expression [2] :

```
# 5 / 2 ;;
- : int = 2
# 5 mod 2 ;;
- : int = 1
# 3.2 /. 5.0 ;;
- : float = 0.64
# 3.2 ** 2. ;;
- : float = 10.240000000000002
# "Hello World" ^ " : " ^ "Bonjour \n" ;;
- : string = "Hello World : Bonjour \n"
```

Les opérateurs de comparaison sont :

=, <>, >, >=, < et <=

Ces opérateurs permettent de comparer deux valeurs *de même type*, pour tout type de base, comme pour les types construits (*cf.* section 1.1.2).

```
# "alpha" < "alphabetique" ;;
- : bool = true
# "alpha" < "alentour" ;;
- : bool = false
# 2 > 3 && 'c' < 'k'
- : bool = false
# 2 > 3 || 'c' < 'k'
- : bool = true
# not(2 > 3) || ('c' > 'k' && 'c' < 'z')
- : bool = true
```

1.1.2 Types construits

1.1.2.1 Types produits

Les valeurs des *types produits* sont des n-uplets, et la comparaison de deux n-uplets de même type est faite selon l'ordre lexicographique :

```
# (4, 9) ;;
- : int * int = (4, 9)
# ('a', 5, "hello", 7.2) ;;
- : char * int * string * float = ('a', 5, "hello", 7.2)
```

1. Le symbole `#` en début de ligne ne fait pas partie de l'expression : il indique que l'interpréteur `ocaml` attend une « phrase » écrite en langage `OCaml`.

2. Plus précisément, le symbole - en début de ligne indique qu'aucun nom n'est défini ; on trouve ensuite le type de l'expression et sa valeur.

```
# (5, 7, 3) < (6, 1, 2) ;;
- : bool = true
# (5, 7, 3) < (5, 7, 4) ;;
- : bool = true
# (5, 7, 3) < (5, 7, 2) ;;
- : bool = false
```

La déclaration d'un type produit se fait suivant la syntaxe suivante :

`type` *type_name* `=` *type_1* `*` *type_2* `*` $\cdots$ `*` *type_n*

Le mot-clé `type` indique la définition d'un type, ici de nom `type_name`. Toute valeur du type produit ainsi défini est un n-uplet, dont le $i^{ème}$ composant est de type `type_i`, pour tout i compris entre `1` et `n`.

1.1.2.2 Types structurés

Une valeur d'un *type structuré* est similaire à un n-uplet dont les différents composants sont *nommés*. Il est nécessaire de déclarer un type structuré avant de l'utiliser.

`type` *type_name* `= {`*field_1* `:` *type_1* `;` $\cdots$ `;` *field_n* `:` *type_n*`}`

Pour tout i compris entre `1` et `n`, le nom du $i^{ème}$ *champ* est `field_i`, et il est de type `type_i`.

Une valeur de ce type s'écrit :

`{`*field_1* `=` *value_1* `;` $\cdots$ `;` *field_n* `=` *value_n*`}`

où `value_i` est de type `type_i`, pour tout i compris entre `1` et `n`, par exemple :

```
# type t_point = {x : float; y : float} ;;
type t_point = { x : float; y : float; }
# {x = 0.0; y = 0.0} ;;
- : t_point = {x = 0.; y = 0.}
```

1.1.2.3 Types énumérés

Un type *énuméré* est déclaré de la manière suivante :

`type` *type_name* `=` *VALUE_1* `|` *VALUE_2* `|` ... `|` *VALUE_n*

où `VALUE_1`, $\cdots$, `VALUE_n` sont les valeurs du type, par exemple :

```
# type t_mycolor = BLUE | RED | GREEN ;;
type t_mycolor = BLUE | RED | GREEN
# BLUE ;;
- : t_mycolor = BLUE
```

1.1.3 Expression

En `OCaml`, toute *expression* a un type et une valeur, par exemple :

```
# not(3 + 7 > 9) || ((5 - 3 ) = 1) ;;
- : bool = false
# {x = 3.0 *. 7.2 ; y = 1.0 +. 2.0 *. 3.2} ;;
- : t_point = {x = 21.6; y = 7.4}
```

1.1.4 Variables non mutables

Intuitivement, une *variable non mutable* (aussi appelée *constante*) est un nom associé à une valeur [1], qu'il faut déclarer avant de l'utiliser ; la déclaration est caractérisée par le mot-clé `let`.

`let` *var_name* `:` *type_name* `=` *value_of_this_type* `;;`

par exemple :

```
# let x : int = 3 ;;
val x : int = 3
# let y : int = x + 2;;
val y : int = 5
# let col : t_mycolor = BLUE ;;
val col : t_mycolor = BLUE
# let (v_int, v_bool) : int * bool =  (3 mod 2, 4 + 5 > 7) ;;
val v_int : int = 1
val v_bool : bool = true
# let p : t_point = {x = 0.0; y = 1.0} ;;
val p : t_point = {x = 0.; y = 1.}
# p.x ;;
- : float = 0.
# p.y ;;
- : float = 1.
```

Le mot-clé `val` indique qu'un nom est associé à la valeur, et bien sûr à son type ; par exemple `val x : int = 3` est la réponse de l'interpréteur `ocaml` indiquant la déclaration d'une variable de nom `x`, de type `int` et de valeur `3`.

On peut noter qu'il est possible de définir des n-uplets de variables, ce qui revient à définir n variables, comme `(v_int, v_bool)` dans l'exemple ci-dessus. On peut noter aussi que l'accès à un champ d'une variable de type structuré s'écrit `var_name.field_name` : *cf.* `p.x` et `p.y` dans l'exemple ci-dessus.

Il est possible de déclarer une variable *locale* à une expression, voire plusieurs variables locales à une expression :

1. Par la suite, nous utilisons généralement le terme « variable ».

`let` *var_name_1* `:` *type_1* `=` *expression_1* `in` *expression*
`let` *var_name_1* `:` *type_1* `=` *expression_1* `and` *var_name_2* `:` *type_2* `=` *expression_2* `and ...` `and` *var_name_n* `:` *type_n* `=` *expression_n* `in` *expression*

Si l'on souhaite déclarer une variable dont la valeur est définie à partir d'une autre variable, il faut enchaîner les déclarations, par exemple :

```
# let x : int = 3 * 12 ;;
val x : int = 36
# let x : int = 3 and y : int = 2 in
      x * y
  ;;
- : int = 6
# let x : int = 3 in
    let y : int = x * 2 in
      y + x
  ;;
- : int = 9
# x ;;
- : int = 36
```

1.1.5 Fonctions

Une fonction est déclarée en utilisant le mot-clé `let`, et sa signature s'écrit :

fun_name`(`*pf_1*`, ..., `*pf_n*` : `*t_1*` * ...* `*t_n*`) : `*t'_1*` * ... * `*t'_k*

où `(pf_1, ... , pf_n)`, de type `t_1 * ... * t_n`, est le paramètre formel de la fonction, `t'_1 * ... * t'_k` est le type de son résultat, et `t_1 * ... * t_n -> t'_1 * ... * t'_k` est le type de la fonction, par exemple :

```
# let add_diff(a, b : int * int) : int * int =
    (a + b, a - b)
;;
val add_diff : int * int -> int * int = <fun>
# let (sum, diff) : int * int = add_diff(5, 4) ;;
val sum : int = 9
val diff : int = 1
```

Comme pour les variables, le mot-clé `val` indique la déclaration d'un nom, par exemple `val add_diff : int * int -> int * int = <fun>` indique la déclaration de la fonction de nom `add_diff`, de type `int * int -> int * int` et dont la valeur, c'est-à-dire sa définition, est symbolisée par `<fun>`.

En `OCaml`, il existe de nombreuses fonctions prédéfinies, par exemple des fonctions de conversion de types :

`int_of_float : float → int`	`float_of_int : int → float`
`int_of_char : char → int`	`char_of_int : int → char`
`string_of_int : int → string`	`string_of_float : float → string`

1.1.6 Expression conditionnelle

Une expression conditionnelle est caractérisée par les mots-clés :

`if` *boolean_expression*
`then` *expression_of_type_t*
`else` *expression_of_same_type_t*

par exemple :

```
# let min2int(a, b : int * int) : int =
    if a < b
    then a
    else b
;;
val min2int : int * int -> int = <fun>
# min2int(4, 3) ;;
- : int = 3
```

Gestion des préconditions : lorsqu'une fonction n'est pas définie pour toutes les valeurs de son paramètre, nous utilisons la fonction `failwith`, dont le paramètre est une chaîne de caractères. L'exécution de cette fonction entraîne la levée d'une erreur et l'affichage de la chaîne de caractères, par exemple :

```
# let div_int(a, b : int * int) : int =
    if b = 0
    then failwith("erreur div_int : division par 0")
    else a / b
;;
val div_int : int * int -> int = <fun>
# div_int(3, 0) ;;
Exception: Failure "erreur div_int : division par 0".
```

1.1.7 Composition de fonctions

Le corps d'une fonction peut bien sûr contenir, en plus de déclarations de variables locales, des appels à des fonctions définies antérieurement, par exemple [1] :

```
# let min3int(a, b, c : int * int * int) : int =
    let min_ab : int = min2int(a, b) in
      min2int(min_ab, c)
;;
val min3int : int * int * int -> int = <fun>
```

1. Les symboles `(*` et `*)` délimitent un commentaire en `OCaml`.

```
# min3int(4, 2, 5) ;;
- : int = 2
(* version alternative *)
# let min3int(a, b, c : int * int * int) : int =
    min2int(min2int(a, b), c)
;;
val min3int : int * int * int -> int = <fun>
```

La commande `#trace` permet de suivre des enchaînements d'appels de fonctions. La commande `#untrace` permet de supprimer l'effet de la commande `#trace`, par exemple :

```
# #trace min3int ;;
min3int is now traced.
# #trace min2int ;;
min2int is now traced.
# min3int(4, 2, 5) ;;
min3int <-- (4, 2, 5)
min2int <-- (4, 2)
min2int --> 2
min2int <-- (2, 5)
min2int --> 2
min3int --> 2
- : int = 2
```

Ce sont des commandes de l'interpréteur `ocaml`, et non des fonctionnalités du langage `OCaml`. La syntaxe est :

`#trace` *function_name* `;;`	`#untrace` *function_name* `;;`

À noter qu'il est possible de déclarer des fonctions *locales*[1], de manière similaire à la déclaration de variables locales.

1.1.8 Fonctions récursives

La déclaration d'une fonction récursive est caractérisée par le mot-clé `rec`. Par exemple, la fonction `pow` calcule x^n pour un entier donné `x` et un entier positif `n` :

```
# let rec pow_aux(x, n : int * int) : int =
    if n = 0
    then 1
    else x * pow_aux(x, n-1)
;;
val pow_aux : int * int -> int = <fun>
# let pow(x, n : int * int) : int =
    if (n < 0) || ((x = 0) && (n = 0))
    then failwith("erreur pow : parametres invalides")
```

1. Cette possibilité n'est généralement pas utilisée dans ce livre, la commande `#trace` ne permettant pas de tracer l'exécution de fonctions locales.

```
    else pow_aux(x, n)
;;
val pow : int * int -> int = <fun>
# #trace pow_aux ;;
pow_aux is now traced.
# pow(3, 4) ;;
pow_aux <-- (3, 4)
pow_aux <-- (3, 3)
pow_aux <-- (3, 2)
pow_aux <-- (3, 1)
pow_aux <-- (3, 0)
pow_aux --> 1
pow_aux --> 3
pow_aux --> 9
pow_aux --> 27
pow_aux --> 81
- : int = 81
```

La fonction ci-dessous calcule le même résultat, mais en récursivité *terminale*[1] :

```
# let rec pow_aux(x, n, ind, res : int * int * int * int) : int =
    if ind = n
    then res
    else pow_aux(x, n, ind + 1, x * res)
;;
val pow_aux : int * int * int * int -> int = <fun>
# let pow(x, n : int * int) : int =
    if (n < 0) || ((x = 0) && (n = 0))
    then failwith("erreur pow : parametres invalides")
    else pow_aux(x, n, 0, 1)
;;
val pow : int * int -> int = <fun>
# #trace pow_aux ;;
pow_aux is now traced.
# pow(3, 4) ;;
pow_aux <-- (3, 4, 0, 1)
pow_aux <-- (3, 4, 1, 3)
pow_aux <-- (3, 4, 2, 9)
pow_aux <-- (3, 4, 3, 27)
pow_aux <-- (3, 4, 4, 81)
pow_aux --> 81
pow_aux --> 81
pow_aux --> 81
pow_aux --> 81
```

1. Schématiquement, une fonction est récursive terminale si, quel que soit le cas d'exécution, soit il n'y a aucun appel récursif, soit la fonction s'appelle elle-même une fois et, dans ce cas, il s'agit de la dernière opération effectuée (pour plus de précisions, voir le chapitre 7).