Types de base – Variables – Portée des variables

Lien vers le programme officiel : https://prepas.org/ups.php?document=10
Horaires de l'enseignement : https://prepas.org/index.php?article=56

1 Types prédéfinis[1], non mutables

Les types de base en **Caml**[2], c'est-à-dire dont l'implémentation en mémoire est la plus élémentaire, qui sont aussi « préconstruits » (« *built-in types* »), sont : `int` ; `float` ; `bool` ; `unit`.
Il existe aussi deux types pour représenter les caractères et les chaînes de caractères : `char` et `string`.

Ces types sont non-mutables, c'est-à-dire que la valeur stockée en mémoire n'est pas modifiable, à la différence de ce qui est, par exemple, pour les listes **OCaml**.

C'est l'interpréteur lui-même, qui infère le type des valeurs que l'on définit, dans la console interactive ou dans un script de programme, il n'y a pas de procédure de déclaration de type pour les variables.
Caml infère ce type (à l'instar de ce que fait Python) à partir de la forme de la chaîne de caractères saisie, dans la console, ou écrite, dans le fichier **.ml**.

- Exemples :

```
# 4;;
- : int = 4
```

```
# 0.;;
- : float = 0.
```

```
# "o";;
- : string = "o"
```

```
# true;;
- : bool = true
```

On trouve ici la description des types de base en **OCaml** :
https://fr.wikiversity.org/wiki/Premiers_pas_en_OCaml/Types_primitifs

1.1 Le type `int`

1.1.a. Représentation en mémoire

Les entiers manipulés par défaut en **OCaml** sont des entiers relatifs, codés sur 32 bits ou 64 bits (selon l'architecture de la machine utilisées), en complément à deux.
Le premier bit de cet encodage sert à distinguer les valeurs entières qui codent des nombres entiers que l'on manipule, de celles qui sont des pointeurs et référencent des adresses mémoires (ce qui permet d'optimiser compilation et calculs). Il y a donc sur une architecture 64 bits, 63 bits utilisés pour l'encodage d'un entier, ce qui permet de couvrir la plage de valeurs $[\![-2^{62}\,;2^{62}-1]\!]$.
Ces valeurs extrêmes sont accessibles par les noms suivants :

```
# max_int;;
- : int = 1073741823
```

```
# min_int;;
- : int = -1073741824
```

On rappelle brièvement que le zéro est codé par une suite de 63 bits à zéro, que le plus plus grand entier positif représentable est codé par un premier bit (bit de poids fort, le plus à gauche) égal à zéro, suivi de 62 bits à 1, tandis qu'un entier négatif, $-n$, opposé d'un entier naturel n ($0 \leq n \leq 2^{62}$), est codé par l'écriture en base 2 de l'entier naturel $2^{63}-n$, ce qui assure que l'encodage des négatifs présente toujours un bit le plus à gauche égal à 1 qui distingue leur codage de celui des entiers positifs.

[1] Une référence sur les types prédéfinis se trouve ici : http://caml.inria.fr/pub/docs/manual-ocaml/core.html
[2] Nous dirons souvent, en abrégé, **Caml** au lieu de **OCaml**, le « **O** » valant pour le caractère « orienté-objet » du langage, aspect que nous n'aborderons pas.

L'interpréteur Caml interprète comme un entier toute suite de chiffres (même commençant par 0, contrairement à Python), l'interprétation est correcte, à condition que la valeur saisie soit dans l'intervalle des entiers représentables :

```
# 012345;;
- : int = 12345
# 1073741824;;
- : int = -1073741824
```

❖ **Différence avec l'IPT** : on fera attention au fait que les entiers de Caml étant codés sur un nombre fixé de bits, la place en mémoire pour le stockage d'une valeur entière est connu et invariable, contrairement à ce qui est pour les entiers de type `int` en Python, mais surtout que l'on s'expose en Caml à des possibilités de dépassement de la capacité de représentation (« *overflow* ») qui n'existent pas avec le type `int` de Python.

1.1.b. Opérateurs pour le type `int`

Les opérations arithmétiques sont codées par les symboles usuels `+`, `-`, `*`.
Le quotient et le reste dans la division euclidienne d'un entier a par un entier b s'obtiennent respectivement à l'aide des opérateurs / (division entière) et mod (modulo) :

```
# 23 mod 5;;
- : int = 3
# 23 / 5;;
- : int = 4
#
```

À noter qu'il n'existe pas d'opérateur pour l'exponentiation des entiers, ce qui oblige à définir une fonction.

1.1.c. Fonctions pour le type `int`

Il existe peu de fonctions prédéfinies pour le type `int`, mais il convient de connaître les fonctions `abs` (valeur absolue), `pred` (prédécesseur) et `succ` (successeur) :

```
# abs(-3);;
- : int = 3
```

```
# pred(0);;
- : int = -1
```

```
# succ(10);;
- : int = 11
```

1.2 Le type `float`

1.2.a. Représentation en mémoire

Les flottants manipulés par défaut en **OCaml** sont des flottants codés sur 64 bits selon la norme IEEE754.
Le bit de poids fort (le plus à gauche donne le signe), tandis que les 11 bits suivants définissent l'exposant, e, et que les 52 bits de poids faible (les plus à droite), codent la mantisse m. Comme en Python, seuls les réels de la forme $\pm\, n/2^q$, $(n, q) \in \mathbb{N} \times \mathbb{Z}$ peuvent avoir une représentation exacte en mémoire (encore faut-il que les valeurs n et q ne soient pas trop grandes), et les calculs mènent à des arrondis s'accumulant au fil des opérations.
Notons, sans nous y attarder ici, qu'il existe un codage spécifique sur 64 bits au sein du type `float`, pour représenter les infinis (valeurs flottantes supérieure ou inférieure à toute autre flottant représentable) et les résultats d'évaluation au format `float` non interprétables comme des valeurs flottantes (`nan` : « not a number ») :

```
# 1. /. 0.;;
- : float = infinity
```

```
# (-1.) /. 0.;;
- : float = neg_infinity
```

```
# acos 2.;;
- : float = nan
```

Comme en Python, sont interprétées comme des flottants les suites de chiffres au sein desquelles figure une virgule explicitée par un point « `.` », ou un « `e` » renvoyant à une écriture scientifique en base 10, précédée éventuellement d'un signe :

```
# +2.;;
- : float = 2.
```

```
# -2.3;;
- : float = -2.3
```

```
# 2e-3;;
- : float = 0.002
```

1.2.b. Opérateurs pour le type `float`

En **Ocaml**, les opérateurs sont chacun dédiés à opérer sur un type donné de valeurs, aussi les opérateurs arithmétiques `+`, `-`, `*` ne sont pas utilisables avec les flottants.
Il faut utiliser les opérateurs « pointés » `+.`, `-.` et `*.`, et ne pas les faire opérer, comme `+`, `-`, `*`. sur des opérandes qui ne seraient pas de même type, sous peine de froisser le compilateur :

```
# 2. + 3.;;
Characters 0-2:
  2. + 3.;;
  ^^
Error: This expression has type float but an expression was expected of
type
         int
# 1. +. 0;;
Characters 6-7:
  1. +. 0;;
       ^
Error: This expression has type int but an expression was expected of type
         float
```

En conséquence, et même si leur utilisation doit être limitée au strict nécessaire, il est nécessaire de faire dans ces situations un appel explicite aux fonctions de conversion de type, comme `float_of int` ou `int_of_float`.

❖ **Différence avec l'IPT** : il n'existe aucun mécanisme de conversion de type en Caml, contrairement à ce qui se passe en Python, où l'opération `2 + 3.` est acceptée et donne un résultat de type `float`. On dit que la « surcharge » des opérateurs n'est pas de mise en Caml.

Il existe un opérateur d'exponentiation pour les flottants, noté `**`, qui n'est pas pointé car son homologue n'existe pas pour le type `int`, mais qui requiert deux opérandes de type `float` :

```
# 2. ** 3.;;
- : float = 8.
#
```

Ceci étant, il ne s'agit pas d'y recourir si l'on traite un problème dans lequel les valeurs sont de type entier (il s'agit en effet peu ou prou d'un appel à une fonction $x \mapsto exp(\alpha . \ln x)$ qui ne garantit pas l'exactitude des calculs).

1.2.c. Fonctions prédéfinies pour le type `float`

Il s'agit de toutes les fonctions usuelles, `abs_float`, `sqrt`, `exp`, `log`, `log10`, trigonométriques, `cos`, `sin`, `tan` et trigonométriques inverses, `acos`, `asin`, `atan`, ainsi que les fonctions `floor` et `ceil`.
On notera que, pour ces deux dernières aussi, la valeur retournée est de type `float` (contrairement à ce qu'il en est en Python).
On signale au passage que la constante `pi` n'est pas prédéfinie et on l'obtient usuellement en calculant l'expression `4. *. atan 1.`, dont l'évaluation renvoie le flottant le plus proche de la valeur exacte de π dans la norme I3E754 (*cf.* Xavier Leroy)

```
# 4. *. atan 1.;;
- : float = 3.1415926535897931
```

La liste exhaustive de ces fonctions est ici (rubrique « Floating-point arithmetics »).

1.3 Le type `bool`

1.3.a. Nommage des valeurs booléennes

Les deux valeurs booléennes sont notées en Caml `true` et `false`.

```
# true;;
- : bool = true
```

```
# false;;
- : bool = false
```

1.3.b. Les opérateurs de comparaison

Comme en Python, comme toujours pourrait-on dire, les valeurs booléennes adviennent dans les programmes par l'évaluation d'expressions correspondant à la comparaison de deux valeurs.

Les opérateurs de comparaison en **Caml**, sont `=` (égalité), `<>` (différent de) et les usuels `<`, `<=`, `>`, `>=`.

Ces opérateurs sont utilisables entre valeurs de même type exclusivement !

Attention ! : la « *double comparaison* » (par exemple `0 < x < 2`) n'est pas autorisée.

Note : on prêtera attention au fait qu'il existe aussi un opérateur testant l'adresse mémoire de deux objets est la même (==) ou non (!=) : on dit que ces deux opérateurs testent l' ***égalité physique*** de deux objets.

La nuance entre `=` et `==` (ou entre `<>` et `!=`) est discutée ici, pour l'instant, on retiendra que les opérateurs introduits précédemment sont adaptés pour tester l'égalité de valeurs de types non-mutables comme les types traités dans ce premier cours, et opèrent sur le même mode que les opérateurs `<`, `<=`, `>`, `>=`.

☛ La référence sur ce sujet (à examiner ultérieurement), se trouve ici :
http://caml.inria.fr/pub/docs/manual-ocaml/libref/Stdlib.html

1.3.c. Les opérateurs booléens

Les opérateurs booléens sont

- `not` (« *négation* », analogue du `not` de Python) ;
- `||` (« *disjonction* », analogue du `or` de Python) ;
- `&&` (« *conjonction* », analogue du `and` de Python).

Comme en Python, une expression composée d'opérateurs booléens est évaluée « paresseusement », de gauche à droite, *i.e.* l'évaluation de l'expression s'interrompt dès que la valeur du résultat est acquise, ce qui explique les résultats d'évaluation suivants :

```
# (2 = 3) && (1 / 0 = 1);;
- : bool = false
```

```
# (2 = 3) || (1 / 0 = 1);;
Exception: Division_by_zero.
```

Ces opérateurs sont « associatifs à droite » (notion sur laquelle nous reviendrons), en attendant, on veillera prudemment à **parenthéser explicitement les expressions booléennes** afin d'éviter toute ambiguïté.

1.4 Le type `unit`

Le type `unit` (comparable au `NoneType` de Python) comporte une seule valeur notée `()` (analogue du `None` de Python). Son utilité est de permettre de définir des fonctions ne prenant aucun argument, ou ne renvoyant aucune valeur (fonctions agissant par « effet de bord », appelées parfois « procédures » plutôt que fonctions), tout en conservant le format propre à toute fonction, qui est de prendre une valeur en argument et de renvoyer une valeur. Le type **`unit`** est aussi de façon générale le type des expressions produisant uniquement des effets de bord, ce qui sera discuté plus loin.

```
# print_int 2;;
2- : unit = ()
```

```
# ();;
- : unit = ()
```

1.5 Le type `char`

Le type `char` est très spécifique (il n'a pas d'analogue en Python par exemple), il comporte 256 valeurs, dont les 128 premières correspondent aux caractères codés dans la table ASCII. Les 128 autres valeurs correspondent à une extension de la table ASCII dans laquelle l'espace de représentation est sur 8 bits.

```
# for i = 0 to 255 do print_char (Char.chr i) done;;
 ^@^A^B^C^D^E^F
 ^N^O^P^Q^R^S^T^U^V^W^X^Y^Z^[^\^]^^^_ !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmn
opqrstuvwxyz{|}~^?\200\201\202\203\204\205\206\207\210\211\212\213\214\215\216\217\220\221\222\223\224\225\226\227\
230\231\232\233\234\235\236\237_¡¢£¤¥¦§¨©ª«¬-®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòó
ôõö÷øùúûüýþÿ- : unit = ()
```

Le type `char` est dénoté par l'emploi de guillemets simples (apostrophes). Il existe un module `Char`, dans lequel on trouve les fonctions `code` et `chr` permettant d'aller et venir entre code et caractère codé :

```
# 'e';;
- : char = 'e'
# "e";;
- : string = "e"
```

```
# 'ef';;
Characters 0-1:
     'ef';;
     ^
     Error: Syntax error
```

```
# Char.code('e');;
- : int = 101
# Char.chr(101);;
- : char = 'e'
```

1.6 Le type `string`

Le type `string` est un type non-mutable pour les chaînes de caractères. Une valeur de ce type est une chaîne de longueur donnée, composée de caractères codés sur un byte (un octet, 8 bits).
Une seule opération est disponible nativement, la concaténation, dénotée par « `^` » :

```
# "ocaml"^" n'est pas "^"Python";;
- : string = "ocaml n'est pas Python"
```

Pour les autres opérations, il faut les invoquer au travers d'un appel au module String.
La longueur d'une chaîne est obtenue par un appel à la fonction `length` du module `String`. La numérotation des positions des caractères composant une chaîne se fait à partir d'un premier indice égal à zéro ; l'accès au $n^{\text{ème}}$ caractère d'une chaîne se fait avec la syntaxe « `.[<position>]` » ou par appel explicite à `String.get`.

```
# String.length "ocaml";;
- : int = 5
```

```
# "ocaml".[0];;
- : char = 'o'
```

```
# String.get "ocaml" 0;;
- : char = 'o'
```

1.7 Le type `exn` (exceptions)

Il s'agit d'un type que l'on reprendra lors de l'étude des fonctions, dont les valeurs correspondent à des situations mettant en défaut l'exécution d'une instruction ou l'évaluation d'une expression. Lorsque cette situation est rencontrée, l'exécution du programme est interrompue et s'affiche un message caractérisant la nature la situation anormale :

```
# 1 / 0 ;;
Exception: Division_by_zero.
```

```
# Division_by_zero;;
- : exn = Division_by_zero
```

Les exceptions prédéfinies sont listées ici, sous le chapeau « Exceptions ».

1.8 Fonctions utilitaires pour la gestion des types

Vous en trouverez une liste et une description assez exhaustive ici.

1.8.a. Les fonctions de conversion de type

La conversion de type implicite n'étant pas possible en Caml, si l'on souhaite ou plutôt si l'on doit changer le type d'une valeur, il est nécessaire de faire appel à une fonction de conversion de type, qui renverra une nouvelle valeur que l'on pourra utiliser.
Les fonctions de conversion de type ont des dénominations toutes sur le même modèle, ce qui permet de les mémoriser assez facilement, voici quelques exemples :

```
# int_of_float 2.;;
- : int = 2
# float_of_int 2;;
- : float = 2.
```

```
# string_of_int 12;;
- : string = "12"
# int_of_char 'a';;
- : int = 97
```

On notera cependant que le recours à ces fonctions doit être limité au strict nécessaire.

1.8.b. Fonctions d'affichage sur la sortie standard

Si l'on souhaite afficher des valeurs sur la sortie standard (valeurs intermédiaires obtenues au fil de l'exécution d'un algorithme par exemple, à des fins de débogage), il faut choisir pour chaque type la fonction d'affichage adaptée au type de la valeur :

`print_int` ; `print_float` ; `print_char` ; `print string` ; `print_bytes`.

On notera qu'il n'existe pas de fonction « print_bool », et à l'inverse qu'il existe une utile fonction `print_newline ()` afin de distiller les affichages sur plusieurs lignes.

1.8.c. Fonctions de lecture sur l'entrée standard

Les données lues sur l'entrée standard afin qu'elle puisse être interprétées comme des valeurs d'un type donné, doivent être lues en faisant appel à la fonction adaptée au type attendu :

- `read_int ()` pour lire une chaîne interprétable comme une valeur entière ;
- `read_float ()` pour lire une chaîne interprétable comme une valeur flottante ;
- `read_line ()` pour lire du texte.

2 Les « variables »

2.1 Préliminaire

Les « variables » dont il est question ici ne sont pas à proprement parler des variables, mais des **noms**, attachés à des valeurs ou à des expressions.
Les noms acceptables **commencent par une** lettre **minuscule** parmi les vingt-six lettres de l'alphabet latin, nécessairement, tandis que les caractères suivants doivent être, soit des minuscules ou des majuscules parmi les vingt-six lettres de l'alphabet latin, soit des chiffres ; les espaces et les tirets sont exclus, mais le « tiret du bas », « _ » est accepté.
On doit distinguer les noms globaux, valides durant toute l'exécution d'un programme, des noms locaux, dont la *portée* limite leur validité à l'évaluation d'une expression.
On pourra lire une description de la nuance entre noms globaux et locaux ici.

2.2 Définition d'une variable globale (nom de portée globale)

Comme en Python, une variable est définie à partir du moment où l'on associe un nom à une valeur.
En **Caml**, plutôt que de dire que l'on affecte une valeur à une variable, on dira que l'on établit une liaison entre un nom et une valeur.

Les variables (globales) se définissent par la syntaxe suivante :

`let <nom de la variable> = <valeur ou expression>.`

C'est **au moment de la définition de la variable** que **son type est inféré** par l'interpréteur (lorsque l'on travaille dans la console ou dans Jupyter) ou le compilateur (lorsque l'on compile un programme pour en créer une version exécutable par une machine).

```
# let a = 2;;
val a : int = 2
```

```
# let b = true;;
val b : bool = true
```

```
# let c = -1e7;;
val c : float = -10000000.
```

Cette syntaxe définit une liaison entre un identificateur (le nom choisi pour la variable) et une valeur.

Si c'est une expression qui suit le symbole « = », cette expression est évaluée et l'identificateur renvoie à la valeur résultant de l'évaluation de l'expression (note : il en est de même en Python).

C'est ce qui explique que la valeur de la variable `b` n'est pas modifiée dans ce qui suit par la redéfinition de la variable `a` (il en est de même en Python : il ne reste aucune trace de l'expression qui a été évaluée et dont le résultat d'évaluation a servi pour donner sa valeur à `b`) :

```
# let a = 1;;
val a : int = 1
# let b = a+1;;
val b : int = 2
# let a = 10;;
val a : int = 10
# b;;
- : int = 2
```

❑ Remarque : comme avec Python, dans la console interactive, l'exécution d'une instruction qui n'est constituée que d'une expression, conduit à l'évaluation de cette expression et la valeur résultant est affichée. Ce n'est pas le cas si une telle expression se trouve dans un script, dans ce cas, il faut forcer l'affiche à l'aide de la fonction adaptée au type de l'expression :

Console interactive :	**Script prog1.ml**	**Éxécution du script prog1.ml**

`# let a = 2;;`
`val a : int = 2`
`# a;;`
`- : int = 2`
`# a + 2;;`
`- : int = 4`
	`let a = 2 ;;`
	`print_int (a + 2);;`
	`a;; (*la valeur de a ne s'affiche pas*)`
		`val a : int = 2`
		`# 4- : unit = ()`
		`# - : int = 2`
		`#`

2.3 Définition conjointe de plusieurs variables

Il est possible de définir en une instruction deux variables ou plus (analogue de l'affectation simultanée de Python : `a, b = 1, 2`), on utilise alors le mot-clé `and`[3] :

```
# let a = 1 and b = 2 and c = 3;;
val a : int = 1
val b : int = 2
val c : int = 3
```

Mais dans ce cas, il n'est pas possible de faire référence à l'une des variables à définir au sein d'une expression à évaluer avant que la définition n'ait été finalisée.

- Exemple :

```
# let n = 1 and m = n + 2;;
Characters 18-19:
  let n = 1 and m = n + 2;;
                    ^
Error: Unbound value n
#
```

❑ Remarque : on pourra utilement méditer sur l'exemple ci-dessous (à ne pas suivre)

```
# let x = 1;;
val x : int = 1
# let x = 10 and y = 1 + x;;
val x : int = 10
val y : int = 2
```

On comprend que la valeur référencée par le nom `x` lors de la définition de `y` est la valeur précédemment définie et non la valeur que l'on s'apprête à attacher au nom `x`, mais la situation est exemplaire d'un code peu lisible (il aurait fallu sensément attacher deux noms différents aux valeurs 1 et 10).

2.4 Définitions locales

❖ En IPT, la notion de portée d'une variable a été abordée au travers du prisme des fonctions : dans un abrégé très grossier, on peut dire que l'on y a distingué les variables globales, définies en dehors des fonctions, et les variables locales dont les noms sont locaux, parce que ce sont les noms donnés aux paramètres de la fonction ou parce ce que les variables sont définies par une affectation de valeur inscrite dans le code de la fonction.

En **Caml**, hors du cadre de la définition de fonction, il est possible de définir un nom local utilisé seulement dans le temps de l'évaluation d'une expression et même d'imbriquer ces définitions locales de noms de variable.

La définition d'une variable dont le nom est local se fait au travers de la syntaxe :

`let <nom de la variable> = <valeur1 ou expression1> in <expression2 à évaluer>`.

Le nom de la variable est alors inconnu en dehors du contexte de l'évaluation de l'expression « 2 ».

```
# let a = 2 in a + 3;;
- : int = 5
# a;;
Characters 0-1:
  a;;
  ^
Error: Unbound value a
```

Sauf bien sûr s'il existe une variable de même nom définie globalement (situation que l'on essaiera d'éviter pour la lisibilité du code) :

```
val a : int = 100
# let a = 2 in a + 3;;
- : int = 5
# a;;
- : int = 100
```

[3] Note : la syntaxe abrégée suivante, équivalente, est néanmoins acceptée :

```
# let a, b, c = (1, 2, 3);;
```

2.5 Définitions locales imbriquées

Il est possible d'imbriquer les définitions locales de variables :

```
# let a = 1 in let b = a + 1 in let c = a + b + 1 in a + b + c;;
- : int = 7
```

L'exemple qui précède est un peu gratuit, mais on retiendra que **l'on veillera à définir les variables avec une portée adaptée à l'utilisation qui en est faite**, c'est-à-dire que l'on ne définira pas une variable globale si la variable en question n'est destinée qu'à être utilisée qu'une fois, dans l'évaluation d'une seule expression.
Afin de « visualiser » la portée de chaque variable dans l'expression précédente, on peut parenthéser l'expression. La portée de chacune des variables, en couleur, est ci-dessous limitée à l'expression parenthésée avec la même couleur :

```
let a = 1 in (let b = a + 1 (in let c = a + b + 1 in (a + b + c)));;
```

2.6 Les « vraies » variables : les références

Dans ce qui précède, il a été dit que nous ne définissions pas avec une instruction `let` une variable au vrai sens du terme, à savoir un objet dont on peut changer la valeur.
La nuance n'est pas encore sensible ici car on peut avoir l'impression lorsque l'on exécute la séquence de deux instructions suivantes :

```
# let a = 1;;
val a : int = 1
# let a = 2;;
val a : int = 2
```

que l'on a changé la valeur d'une même variable `a`, de même ci-dessous pour une variable locale `a` :

```
# let a = 1 in let a = 2 in a * a;;
- : int = 4
```

Or ce n'est pas le cas, en fait on a, dans les deux cas utilisé le même nom pour désigner deux valeurs distinctes en mémoire (auxquelles ont accède par deux adresses différentes en mémoire).
La nuance deviendra sensible lorsque l'on abordera les instructions à caractère impératif (instructions conditionnelles et boucles).

On définit une « vraie » variable lorsqu'un nom fait référence à une adresse mémoire qui ne change pas au fil de l'exécution du programme et qui, elle, permet d'accéder à des valeurs qu'il est possible de modifier au fil de l'exécution du programme.

Il existe un type générique pour de telles variables : le type **référence**. Ce type se décline, suivant le type de valeur référencée, en **référence d'entier**, **référence de flottant**, ***etc.***

La syntaxe pour définir une variable qui est une référence est la suivante :

```
# let a = ref 0;;
val a : int ref = {contents = 0}
```

Une référence peut être vue comme un **contenant** (c'est à ce contenant que l'on affecté le nom `a`), permettant d'accéder à un **contenu**, qui est la valeur référencée :

- pour accéder à la valeur référencée, la syntaxe est la suivante :

```
# !a;;
- : int = 0
```

`!` est appelé opérateur de déréférencement.

- pour modifier la valeur référencée, la syntaxe est la suivante :

```
# a := 1;; (* expression de type unit *)
```

`:=` est appelé opérateur d'affectation.
On aura l'occasion d'y revenir, mais on peut noter que l'expression `a := 1` est de type `unit`.