Le type `unit` est un type particulier qui **contient une unique valeur**, appelée « vide » et écrite

`()`

Il s'agit d'un *type singleton*.

Exemples

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

```
# let a = () in
  let b = a in
  b;;
- : unit = ()
```

Les opérateurs relationnels sont bien définis sur `unit`.

Dans la suite, nous verrons que ce type sert à rendre les fonctions homogènes au sens où **toute fonction doit renvoyer une valeur**.

En effet, une fonction qui n'est pas censée renvoyer de valeur va renvoyer `()`.

Ce type sert également à introduire des **retardateurs** (notion qui sera abordée aussi dans la suite).

6.5. Expression conditionnelles

Une *expression conditionnelle* est une expression de la forme

`if COND then EXP1 else EXP2`

où `COND` est une expression de type `bool` et `EXP1` et `EXP2` sont deux expressions **toutes deux d'un même type**.

Exemples

```
# if (3 >= 2) || ("aab" <= "aa") then
      "ABC"
  else
      "CDE";;
- : string = "ABC"
```

```
# let x = if 16. <= 2. ** 3. then 21 else 42;;
val x : int = 42
```

Toute expression conditionnelle **possède une valeur** :

1. lorsque `COND` s'évalue en `true`, l'expression conditionnelle a pour valeur celle de `EXP1` ;
2. lorsque `COND` s'évalue en `false`, l'expression conditionnelle a pour valeur celle de `EXP2`.

Une expression conditionnelle étant elle-même une expression, il est possible d'**imbriquer les expressions conditionnelles**.

Exemple

```
# if true then
      if 2 = 3 then
          'A'
      else
          'B'
  else
      'C';;
- : char = 'B'
```

La (bonne) indentation aide à comprendre cette phrase.

Néanmoins, l'interpréteur OCaml la comprend sans ambiguïté.

Il n'a pas besoin de marqueur de fin (comme le `}` de certains langages).

Il est aussi possible de les utiliser au sein d'autres opérations.

Exemple

```
# (if "ab" <= "baa" then 1 else 2) + (if false then 3 else 2);;
  - : int = 3
```

Les valeurs des deux expressions conditionnelles sont additionnées.

L'expression est bien typée car les types des sous-expressions des conditionnelles sont identiques.

Rappellons qu'en programmation fonctionnelle, il n'y a pas d'effet secondaire.

Ainsi, l'**intérêt d'une expression conditionnelle** repose dans la **valeur** qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Exemples

Voici quelques évaluations d'expressions conditionnelles :

- `if 3 >= 1 then 21 else 24` → `if true then 21 else 24` → `21`,
- `if if "ab" = "ab" then 1 = 0 else true then 28 else 21`
  → `if if true then 1 = 0 else true then 28 else 21`
  → `if 1 = 0 then 28 else 21` → `if false then 28 else 21` → `21`.

Grâce au **principe de transparence référentielle**, ces deux expressions peuvent être remplacées par leurs valeurs, l'entier `21`, dans tout programme les employant sans modifier la valeur finale qu'il calcule.

Une *demi-expression conditionnelle* est une expression de la forme

```
if COND then EXP
```

où `COND` est une expression de type `bool` et `EXP` est une expression.

Le système **complète implicitement** et automatiquement cette demi-expression conditionnelle en l'expression conditionnelle

```
if COND then EXP else ()
```

En conséquence, la valeur de `EXP` doit être de type `unit`.

Exemple

```
let f x =
    if x >= 9 then ()
```

est équivalent à

```
let f x =
    if x >= 9 then () else ()
```

Exemple

Voici un exemple plus concret :

```
# let x = read_int () in if x <= 0 then print_int (-x);;
```

Ceci lit un entier et affiche sa valeur absolue s'il est négatif et ne fait rien sinon.

7. Fonctions

Rappelons qu'en programmation fonctionnelle,

1. l'objet de base est la **fonction** ;
2. un programme est une **collection de définitions de fonctions** ;
3. un programme s'exécute en **évaluant sa fonction principale** ;
4. la définition d'une fonction sert à expliquer comment **calculer l'application de cette fonction à des arguments** ;
5. bien entendu, des **définitions de types** peuvent se trouver dans un programme (ce point sera abordé un peu plus loin).

La raison d'être d'un programme, en programmation fonctionnelle, est de **calculer une valeur**. Il s'agit de la **valeur de retour** calculée par la fonction principale.

Durant cette évaluation, dans un style fonctionnel non pur, des effets secondaires d'impression sur la sortie standard ou sur le système de fichiers ou encore de lecture d'une valeur sur l'entrée standard peuvent avoir lieu.

7.1. Rappels fondamentaux

Voici quelques rappels fondamentaux sur les ensembles, relations binaires et fonctions.

Un *ensemble* est une collection d'éléments.

Étant donné un ensemble E et une entité x, il y a deux possibilités mutuellement exclusives :

- $x \in E$. Nous disons alors que « x appartient à E » ou que « x est un élément de E ».
- $x \notin E$. Nous disons alors que « x n'appartient pas à E ».

Un ensemble E' est un *sous-ensemble* (également appelé *partie*) d'un ensemble E si pour toute entité x, $x \in E'$ implique $x \in E$. Cette propriété est notée $E' \subseteq E$.

Les *opérations ensemblistes* principales sont

- l'**union** $\cup$, telle que $x \in E_1 \cup E_2$ ssi $x \in E_1$ ou $x \in E_2$;
- l'**intersection** $\cap$, telle que $x \in E_1 \cap E_2$ ssi $x \in E_1$ et $x \in E_2$;
- le **produit cartésien** $\times$ tel que $x \in E_1 \times E_2$ ssi $x = (x_1, x_2)$ avec $x_1 \in E_1$ et $x_2 \in E_2$;
- l'**ensemble des parties** $\mathcal{P}$ tel que $\mathcal{P}(E)$ est l'ensemble des ensembles E' tels que $E' \subseteq E$.

Un **type** peut-être vu comme un ensemble. Dire qu'une expression `EXP` est de type `T` revient à dire que `EXP` est un élément de l'ensemble `T`.

Une *relation binaire* entre deux ensembles E et E' est une partie de $\mathcal{P}(E \times E')$.

Ainsi, une relation binaire entre E et E' est un ensemble de couples (x, x') tels que $x \in E$ et $x' \in E'$.

Une *fonction* de *domaine* E et de *codomaine* E' est une relation binaire f entre E et E' telle que pour tout $x \in E$, il y a au plus un $x' \in E'$ tel que $(x, x') \in f$. Cette propriété est notée $f : E \to E'$.

Si pour un $x \in E$, il existe $x' \in E'$ tel que $(x, x') \in f$, x' est l'*image* de x. Nous notons alors par $f(x)$ l'élément x'.

Une fonction à n **entrées** est une fonction $f : (E_1 \times \cdots \times E_n) \to E'$. Nous notons alors par $f(x_1, \ldots, x_n)$ l'élément $f((x_1, \ldots, x_n))$.

Remarque : lorsque $n = 0$, la fonction est de la forme $f : E'$. Il s'agit donc d'une *constante*.

Nous verrons cependant que les fonctions à plusieurs entrées sont conceptualisées de manière beaucoup plus ingénieuse en programmation fonctionnelle.

Au sein de la manipulation des fonctions, il faut faire la distinction entre paramètre et argument :

1. un *paramètre* est un **identificateur** non lié à une valeur et intervenant dans la définition d'une fonction ;
2. un *argument* est une **expression** qui vient se substituer à un paramètre lors de l'appel à une fonction.

Exemple

Soit $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ la fonction définie par $f(x_1, x_2) := x_1 + x_2$.

Les noms x_1 et x_2 sont les **paramètres** de f.

Lors de l'appel $f(f(2,1), 6-3)$, f est appelée avec les **arguments** $f(2,1)$ et $6-3$.

Lors d'un appel à une fonction f à n entrées avec les arguments a_1, ..., a_n, nous disons que l'on *applique* f *à* a_1, ..., a_n.

Exemple

Dans l'expression `max 2 (8 + 1)`, la fonction `max` est appliquée aux arguments `2` et `8 + 1`.

7.2. Définitions et applications de fonctions

Pour **définir une fonction** à `n` paramètres, nous utilisons la construction syntaxique

```
let ID P1 ... Pn = EXP
```

où `ID` est le nom de la fonction, `P1`, ..., `Pn` sont ses paramètres et `EXP` est une expression.

Il s'agit d'une **liaison globale** et donc aussi d'une **déclaration**.

Tout appel à la fonction `ID` possède comme valeur la valeur de `EXP` dans laquelle les occurrences libres de ses paramètres sont remplacées par les arguments.

Exemple

Voici une définition de fonction :

```
# let oppose x = -x;;
val oppose : int -> int = <fun>
```

Voici quelques explications sur la réponse de l'interpréteur :

- `val oppose` informe qu'on a lié au nom `oppose` une valeur ;
- `: int -> int` informe que cette valeur est de type `int -> int` ;
- `= <fun>` est un affichage générique la fonction.

Pour **appliquer une fonction à des arguments**, nous utilisons la construction syntaxique

```
ID A1 ... An
```

Les arguments sont séparés par des **espaces** sans utiliser de parenthèses englobantes. Elles servent uniquement à délimiter un argument qui serait une expression non atomique.

Réaliser un appel par `ID(A1, ..., An)` de manière similaire à ce qu'il se fait dans certaines langages de programmation produira (souvent) une **erreur**. La fonction `ID` serait dans ce cas appliquée à un **unique** `n`**-uplet** `(A1, ..., An)`, ce qui n'est pas forcément recherché.

Exemple

Considérons la fonction

```
# let sum_squares x1 x2 = x1 * x1 + x2 * x2;;
val sum_squares : int -> int -> int = <fun>
```

Lors de l'appel `sum_squares 5 (2 + 1)`, les paramètres `x1` et `x2` sont substitués respecivement dans l'expression définissant la fonction par les arguments, `5` et `2 + 1` (ou leurs valeurs `5` et `3`). L'évaluation donne ainsi

`sum_squares 5 (2 + 1)` → `(5) * (5) + (2 + 1) * (2 + 1)` → `34`

ou

`sum_squares 5 (2 + 1)` → `sum_squares 5 3` → `(5) * (5) + (3) * (3)` → `34`

suivant la **stratégie d'évaluation du langage** (ce point sera vu en détail plus loin).

Une *fonction locale* est une fonction définie à l'intérieure d'une autre. Sa **portée** s'étend à la fonction dans laquelle elle est définie. Elle suit les mêmes règles d'ombrage que celles qui s'appliquent aux liaisons locales de noms.

Exemple

```
# let f u =
      let aux u =
          u ^ "a" ^ u
      in
      (aux u) ^ (aux ("b" ^ u));;
val f : string -> string = <fun>
```

```
# f "";;
- : string = "abab"

# f "cd";;
- : string = "cdacdbcdabcd"
```

Il est également possible de réaliser des définitions de fonctions (locales) **simultanées**.

Exemple

```
# let f x =
      let g y = y - 2 = x
      and h x = 2 * x in
      g (h x);;
val f : int -> bool = <fun>
```

```
# f 1;;
- : bool = false

# f 2;;
- : bool = true
```

7.3. Types fonctionnels

Considérons une fonction de la forme `let f x = EXP`.

Lors de son évaluation, l'interpréteur affiche une réponse de la forme

```
val f : t1 -> t = <fun>
```

Le type associé au nom `f` est ainsi `t1 -> t`. Cela signifie qu'il s'agit d'une fonction acceptant un argument de type `t1` et renvoyant une valeur de type `t`.

Le symbole `->` est un **constructeur de types** : c'est en particulier le constructeur des *types fonctionnels*.

L'application de la fonction `f` à un argument `a` est **correctement typée** si

- l'argument `a` est de type `t1` ;
- l'expression `f a` est utilisée là où une valeur de type `t` est attendue.

Exemples

Considérons la fonction

```
# let f x = if x >= 0 then "positif" else "negatif";;
val f : int -> string = <fun>
```

```
# "Nombre " ^ (f 21);;
- : string = "Nombre positif"
```
Cette expression est bien typée.

```
# 4 + (f 21);;
Error: ... type string but ... expected of type int
```
Cette expression est incorrectement typée.

Considérons une fonction de la forme `let f x1 x2 = EXP`.

Lors de son évaluation, l'interpréteur affiche une réponse de la forme

```
val f : t1 -> t2 -> t = <fun>
```

Ce type `t1 -> t2 -> t` doit se comprendre comme étant le type `t1 -> (t2 -> t)` car le constructeur `->` **associe de la droite vers la gauche**.

Ceci suggère que `f` est une fonction qui

- accepte un argument de type `t1` ;
- **renvoie une valeur de type fonctionnel** `t2 -> t`.

Une fonction définie avec deux paramètres est ainsi une **fonction à un paramètre** qui **renvoie une fonction** à un paramètre.

Exemple

Considérons la fonction

```
# let sum_squares x1 x2 = x1 * x1 + x2 * x2;;
val sum_squares : int -> int -> int = <fun>
```

Ceci suggère qu'il est possible d'appeler `sum_squares` avec **un seul argument**, au lieu de deux :

```
# sum_squares 0;;
- : int -> int = <fun>
```

7.4. Applications partielles et fonctions curryfiées

Cette conceptualisation des fonctions à deux paramètres se généralise à $n \in \mathbb{N}$ **paramètres**.

Considérons une fonction de la forme `let f x1 ... xn = EXP`.

Lors de son évaluation, l'interpréteur affiche une réponse de la forme

```
val f : t1 -> ... -> tn -> t = <fun>
```

Ce type est compris comme le type totalement parenthésé

```
t1 -> (t2 -> (t3 -> ... -> (tn -> t)...))
```

Comme cas particulier, quand $n = 0$, le type de `f` est simplement `t`.

Lors de l'application `f a1 ... an` de `f` à $k \in \mathbb{N}$ arguments tel que $k \leqslant n$, nous obtenons une expression de type

```
tk+1 -> ... -> tn -> t
```

Lorsque $k < n$, il s'agit d'une *application partielle*.

Conséquence 1 : appliquer partiellement une fonction à des arguments **crée une nouvelle fonction**.

Conséquence 2 : l'application d'une fonction à un argument **associe de la gauche vers la droite**. Ainsi, `f a1 a2 ...ak` est compris comme l'application totalement parenthésée `(...((f a1) a2) ...) ak`.

Exemple

Considérons la fonction

```
# let surround pref suff fact = pref ^ fact ^ suff;;
val surround : string -> string -> string -> string = <fun>
```

Cette fonction possède **exactement trois paramètres**. Voici quelques applications possibles :

```
# surround "a" "b" "c";;
- : string = "acb"
```

Il s'agit ici d'une application avec $k = 3$ selon les notations de la page précédente.

```
# surround "((" "))";;
- : string -> string = <fun>
```

Il s'agit ici d'une application avec $k = 2$ et donc d'une **application partielle**.

Il y a plusieurs façons d'utiliser cette dernière fonction obtenue par application partielle :

1. directement par `(surround "((" "))") "abc"`, qui est équivalent à `surround "((" "))" "abc"` ;
2. via un alias :

```
# let surround' fact = surround "((" "))" fact;;
val surround' : string -> string = <fun>
# surround' "abc";;
- : string = "((abc))"
```

Ainsi, `surround "((" "))"` est une fonction acceptant une chaîne de caractère et qui la renvoie avec `((` au début et `))` à la fin.

En OCaml (ainsi que dans beaucoup d'autres langages fonctionnels), les fonctions sont *curryfiées*.

Ceci signifie qu'une fonction à plusieurs entrées est conceptualisée comme une **fonction à une seule entrée et une seule sortie** (pouvant elle-même être d'un type fonctionnel).

Exemples

- La fonction `(+)` est de type `int -> int -> int`, équivalent au type `int -> (int -> int)`.
  L'appel `(+) 1` est donc de type `int -> int` et est une fonction qui accepte un entier et qui renvoie son successeur.
- La fonction `List.cons` est de type `'a -> 'a list -> 'a list`, équivalent au type `'a -> ('a list -> 'a list)`.
  L'appel `List.cons 0` est donc de type `int list -> int list` et est une fonction qui accepte une liste d'entiers `lst` et qui renvoie la liste dont la tête est `0` et la queue est `lst`.
- La fonction `String.sub` est de type `string -> int -> int -> string`, équivalent aux types `string -> int -> (int -> string)` et `string -> (int -> (int -> string))`.
  Nous pouvons donc la voir de manière équivalence comme une fonction à trois entrées et qui renvoie une chaîne de caractères, comme une fonction à deux entrées et qui renvoie une fonction, ou encore comme une fonction à une entrée et qui renvoie une fonction.
  Par exemple, `String.sub "abcdefg" 2` est une expression de type `int -> string`.

C'est précisément le fait que le fonctions sont **curryfiées** qui donne accès au mécanisme d'**application partielle**.

Ce procédé est extrêmement puissant et utile pour avoir du **code concis et lisible** (ceci sera illustré plus tard lors de l'étude des fonctions d'ordre supérieur).

Supposons que nous disposions d'une fonction `f` de type `t1 -> t2 -> t`.

Nous savons alors que `f` possède **au moins deux entrées** (il se peut que `t` soit fonctionnel et ainsi que l'on puisse voir `f` comme une fonction à strictement plus de deux entrées).

Une question légitime se pose : comment appliquer `f` partiellement de sorte à accéder uniquement à son 2e paramètre (sans toucher à son 1er comme c'est le cas via une application partielle) ?

Réponse : utiliser la fonction `Fun.flip` de la bibliothèque standard. Son type est `('a -> 'b -> 'c) -> 'b -> 'a -> 'c` et sera expliqué plus tard. L'appel `Fun.flip f a2` est la fonction de type `t1 -> t` qui se comporte comme `f` dans le cas où son 2e argument est **spécialisé** à `a2`.

Exemple

```
# let f' = Fun.flip (-) 1;;
val f' : int -> int = <fun>
```

```
# f' 2;;
- : int = 1
```

7.5. Fonctions anonymes

Une fonction n'a pas nécessairement besoin d'avoir un nom pour exister.

Une fonction bien définie mais qui **n'a pas de nom** est une *fonction anonyme*.

La construction syntaxique

```
fun P1 ... Pn -> EXP
```

où `P1`, ..., `Pn` sont des paramètres et `EXP` est une expression permet de définir une fonction anonyme.

Exemples

```
# (fun a b -> (a + b) * a) 4 3;;
- : int = 28
```

Définit une fonction anonyme appliquée aux arguments `4` et `3`.

```
# let produit k =
      fun x -> x * k;;
val produit : int -> int -> int = <fun>
```

L'expression `produit 10` a pour valeur une fonction de type `int -> int` qui multiplie par `10` son argument.

La véritable manière de définir une fonction en OCaml se fait par

```
function P -> EXP
```

Cette expression est une fonction anonyme à un paramètre `P`.

Les deux autres manières de définir les fonctions sont du sucre syntaxique :

1. la construction

   ```
   fun P1 P2 ... Pn -> EXP
   ```

   est équivalente à

   ```
   function P1 -> function P2 -> ... -> function Pn -> EXP
   ```

   Ceci utilise le fait que les fonctions sont curryfiées.

2. La construction

   ```
   let F P1 ... Pn = EXP
   ```

   est équivalente à

   ```
   let F = fun P1 ... Pn -> EXP
   ```

7.6. Inférence des types des fonctions

Le **mécanisme d'inférence des type** de l'OCaml agit statiquement et implicitement.

Il se charge de **deviner le type** d'une fonction en analysant son code.

Exemples

1.
```
let mystere x y z =
    if x && (y 1) then z
```
Cette fonction est de type
```
bool -> (int -> bool) -> unit -> unit
```

2.
```
let etrange x y =
    "a" ^ ((y 1) ((x 'a') + 1)) ^ "b"
```
Cette fonction est de type
```
(char -> int) -> (int -> int -> string)
    -> string
```

3.
```
let saugrenu x y z t =
    1. +. (x ((y (not z) (t - 1)) ^ "ab") y)
```
Cette fonction est de type
```
(string -> (bool -> int -> string) -> float)
    -> (bool -> int -> string) -> bool -> int
    -> float
```

4.
```
let bizarre y =
    ((fun x -> x + 1) 2) + (y (string_of_int 3))
```
Cette fonction est de type
```
(string -> int) -> int
```