Il est possible de spécifier les types de certains des paramètres et/ou de la sortie pour gagner en lisibilité ou pour restreindre les types autorisés par

```
let F (P1 : T1) ... (Pn : Tn) : T = EXP
```

où les `T1`, ..., `Tn` et `T` sont des types.

Exemples

Sans restriction :

```
# let pair x y = (x, y);;
val pair : 'a -> 'b -> 'a * 'b = <fun>
```

Avec restriction du type du premier 1er paramètre :

```
# let pair (x : string) y = (x, y);;
val pair : string -> 'a -> string * 'a = <fun>
```

Avec restriction du type de retour :

```
# let pair x y : (int * int) = (x, y);;
val pair : int -> int -> int * int = <fun>
```

7.7. Retardateurs

Considérons la liaison d'un nom à une valeur

```
let f = EXP
```

Rappelons que `f` peut être vue comme la fonction constante qui renvoie la valeur de l'expression `EXP`.

Cette déclaration fait que le calcul demandé pour l´évaluation d'`EXP` est réalisé **avant tout appel** potentiel à `f`.

Comparons ceci avec la liaison

```
let f () = EXP
```

Ici, `f` peut être vue comme la fonction unaire constante qui renvoie la valeur d'`EXP`.

Cette déclaration ne demande pas l'évaluation d'`EXP`. Cette expression n'est évaluée qu'**au moment où `f` est utilisée** via l'appel `f ()`. Cette fonction de type `unit -> T` où `T` est le type d'`EXP` est un *retardateur*.

Cette construction est utile en particulier lorsqu'`EXP` effectue un **effet secondaire** et que celui-ci doit être exécuté à un moment précis.

8. Entrées, sorties et séquences

8.1. Entrées/sorties

Les interpréteurs et leurs boucles d'interaction suffisent pour définir des fonctions et les tester. Les résultats sont affichés par l'interpréteur lui-même.

Cependant, il est important d'avoir à disposition de véritables fonctions d'entrée/sortie lorsqu'il est question de développer des programmes complets.

Il existe pour cela des **fonctions d'entrée/sortie** qui permettent de lire des données sur l'entrée standard et d'en écrire sur la sortie standard.

Les fonctions d'entrée/sortie utilisent le type `unit` et son unique valeur `()`.

Rappel : l'utilisation d'entrées/sorties fait **sortir du paradigme de programmation fonctionnelle pure**. Elles produisent des effets secondaires (impression sur la sortie standard ou bien attente d'une action de l'utilisateur).

En principe, les fonctions d'écriture ne renvoient rien. C'est leur **effet secondaire** qui forme leur intérêt.

En pratique, comme toute fonction doit renvoyer une valeur, par convention, **toute fonction d'écriture renvoie** `()`.

Les fonctions d'écriture principales sont

- `val print_int : int -> unit`
- `val print_float : float -> unit`
- `val print_char : char -> unit`
- `val print_string : string -> unit`
- `val print_newline : unit -> unit`
- `val print_endline : string -> unit`

La fonction d'**impression formatée**

```
val Printf.printf : ('a, out_channel, unit) format -> 'a
```

fonctionne comme le `printf` du langage C.

Exemple

La phrase

```
# Printf.printf "J'ai %d pommes et %d %s" 3 (3 + 1) "bananes";;
```

imprime `J'ai 3 pommes et 4 bananes`.

En principe, les fonctions de lecture ne prennent rien en entrée. C'est ce qu'elles renvoient, c'est-à-dire la **valeur saisie** par l'utilisateur, qui forme leur intérêt.

En pratique, comme toute fonction qui ne possède pas de paramètre est une constante, il n'est pas possible de les implanter ainsi. Pour cette raison, ces fonctions sont bâties comme des **retardateurs**.

Les fonctions de lecture principales sont

- `val read_int : unit -> int`
- `val read_float : unit -> float`
- `val read_line : unit -> string`

La fonction de **lecture formatée**

```
val Scanf.scanf : ('a, 'b, 'c, 'd) Scanf.scanner
```

fonctionne comme le `scanf` du C, à la différence près qu'elle ne réalise pas d'affectation des valeurs lues à des variables mais effectue une action selon une fonction.

Exemple

La phrase

```
# Scanf.scanf "%d %d %d" (fun n1 n2 n3 -> print_int (n1 + n2 + n3));;
```

lit trois entiers sur l'entrée standard et renvoie la valeur renvoyée par la fonction anonyme mentionnée appliquée aux valeurs lues. Cette fonction affiche leur somme et renvoie `()`.

8.2. Séquences

L'utilisation des fonctions d'entrée/sortie est parfaitement adaptée à la programmation impérative dans laquelle les exécutions des instructions s'enchaînent les unes après les autres.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre. On utilise pour cela l'*opérateur de séquence* `;`. Il s'utilise de la manière suivante :

```
EXP1; EXP2
```

où `EXP1` est une expression dont la valeur est de type `unit` et `EXP2` est une expression. La **valeur** d'`EXP1; EXP2` est celle d'`EXP2`.

L'opérateur `;` **associe de la gauche vers la droite**. Ainsi,

```
E1; E2; ... ; En
```

est compris comme l'expression totalement parenthésée

```
(...(E1; E2); ...); En
```

La **valeur** de `E1; E2; ... ; En` est ainsi celle de `En`. De ce fait, les expressions `E1`, ..., `En-1` doivent être de type `unit`.

Exemples

```
# let add x y =
      print_string "Appel de add";
      print_int x;
      print_int y;
      x + y;;
val add : int -> int -> int = <fun>
```

Tout appel `add a b` imprime des éléments sur la sortie standard puis renvoie `a + b`.

```
# let test_div n =
      if n mod 2 = 0 then
          print_string "pair\n";
      if n mod 3 = 0 then
          print_string "multiple de 3\n";;
val test_div : int -> unit = <fun>
```

Tout appel `test_div a` imprime une chaîne de caractères si `a` est divisible par 2 et une autre chaîne si `a` est divisible par 3. La valeur `()` est renvoyée dans tous les cas.

Le schéma

```
if COND then
    print_X;
```

où `COND` est une expression de type `bool` et `print_X` est une fonction d'impression est bien adapté à ce contexte. Il permet d'imprimer une information si une condition est vérifiée.

L'utilisation de l'opérateur de séquence `;` est source d'erreurs classiques.

Exemple

```
let div_decr x =
    if x mod 2 = 0 then
        print_string "pair";
        x / 2
    else
        print_string "impair";
        x - 1;;
```

Cette fonction se comprend comme la fonction ci-contre (en améliorant l'indentation) faisant apparaître clairement une **erreur de syntaxe** :

```
let div_decr x =
    if x mod 2 = 0 then
        print_string "pair";
    x / 2
    else
        print_string "impair";
    x - 1;;
```

L'opérateur de séquence est **moins prioritaire** que la conditionnelle.

Pour résoudre ce problème, nous utilisons la notion de bloc. Un *bloc* est une expression de la forme

`begin EXP end`

où `EXP` est une expression. La valeur de `begin EXP end` est celle d'`EXP`.

Exemple

```
let div_decr x =
    if x mod 2 = 0 then begin
        print_string "pair";
        x / 2
    end
    else begin
        print_string "impair";
        x - 1
    end
```

9. Écriture de projets

9.1. Méthode directe

Au lieu d'utiliser l'OCaml uniquement en mode interprété via `ocaml` ou `utop`, il est possible d'écrire de manière classique un programme dans un (ou plusieurs) fichier(s) au sein d'un projet et de le compiler pour **obtenir un exécutable**.

Un *projet* est un ensemble de fichiers `.ml` dont exactement un contient une fonction principale et d'éventuels autres fichiers de configuration.

Un projet tenant sur un seul fichier `Program.ml` se compile par la commande

```
ocamlc -o Program Program.ml
```

Ceci invoque le **compilateur bytecode**. Le code produit a l'avantage d'être portable.

Le **compilateur natif** peut être invoqué via la commande

```
ocamlopt -o Prog Prog.ml
```

Le code produit a le désavantage de dépendre de l'architecture de la machine qui l'a compilé mais a l'avantage d'être performant.

Pour **compiler un projet**,

1. on construit un *fichier objet* (`.cmo` ou `.cmx`) pour chaque fichier `F.ml` du projet au moyen de la commande

   ```
   ocamlc -c F.ml
   ```

   ou bien

   ```
   ocamlopt -c F.ml
   ```

2. on appelle l'*éditeur de liens* par la commande

   ```
   ocamlc -o Prog F1.cmo ... Fn.cmo
   ```

   ou bien

   ```
   ocamlopt -o Prog F1.cmx ... Fn.cmx
   ```

   où les `F1.cm*`, ..., `Fn.cm*` sont les fichiers objets du projet.

Dans le cas où un fichier `B.ml` a besoin d'une entité `x` définie dans un fichier `A.ml` du projet, il faut utiliser `x` aux endroits désirés dans `B.ml` en la **préfixant** par `A.`, formant `A.x`.

Exemple

```
(* A.ml *)

let double x = 2 * x
```

```
(* B.ml *)

let quad x = 2 * (A.double x)
```

Dans ce projet, la fonction `double` de `A.ml` s'emploie dans `B.ml` via l'identificateur `A.double`.

Il est possible de réaliser une **ouverture explicite** de `A.ml` dans `B.ml` par

```
open A
```

permettant une utilisation directe (sans préfixe) des entités de `A.ml` dans `B.ml`.

Exemple

```
(* A.ml *)

let double x = 2 * x
```

```
(* B.ml *)

open A

let quad x = 2 * (double x)
```

L'ouverture explicite de `A.ml` dans `B.ml` permet d'appeler la fonction `double` dans `B.ml` sans le préfixe `A.`.

Un *espace de nom* est une fonction (au sens mathématique) qui a un ensemble de symboles (des identificateurs) associe leur définition. Les espaces de noms permettent de **restreindre la visibilité** de certaines définitions.

À chaque fichier `.ml` est associé un espace de noms qui lui est propre.

Il est ainsi possible d'avoir dans un même projet deux fichiers qui définissent des fonctions nommées par un même identificateur.

Exemple

```
(* A.ml *)

let fct x =
    ...
```

```
(* B.ml *)

let fct x y =
    ...
```

```
(* C.ml *)
...
A.fct 1
...
B.fct 'a' 2
...
```

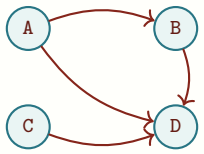
Dans ce projet, deux fonctions nommées `fct` sont définies.
Leur nom absolu n'est en revanche pas le même (`A.fct` et `B.fct`).
Il n'y a ainsi aucune ambiguïté dans `C.ml` qui utilise ces deux fonctions.

À la différence de certains autres langages, il est impossible de construire le fichier objet d'un fichier `X.ml` qui utilise des entités de `Y.ml` avant d'avoir produit celui de `Y.ml`.

Il faut donc compiler le projet dans l'ordre dicté par un **tri topologique** du graphe dual du graphe d'inclusions du projet.

Exemple

Considérons le **graphe d'utilisations** suivant :



Toute flèche 

signifie que le fichier `X.ml` utilise des entités de `Y.ml`.

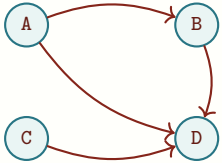
On a les trois ordres suivants possibles :

- ☐ `D.ml`, `C.ml`, `B.ml`, `A.ml` ;
- ☐ `D.ml`, `B.ml`, `C.ml`, `A.ml` ;
- ☐ `D.ml`, `B.ml`, `A.ml`, `C.ml`.

L'utilitaire `ocamldep` permet de calculer les dépendances des fichiers d'un projet en un format utilisable par `make`.

Exemple

Reprenons le graphe d'inclusions



La commande `ocamldep *.ml` affiche

```
A.cmo: D.cmo B.cmo
A.cmx: D.cmx B.cmx
B.cmo: D.cmo
B.cmx: D.cmx
C.cmo: D.cmo
C.cmx: D.cmx
D.cmo:
D.cmx:
```

Il est ainsi possible d'écrire des `Makefile` génériques en appelant à l'intérieur `ocamldep` de façon adéquate.

9.2. Utilisation de dune

Une autre façon, plus sophistiquée, de gérer des projets est offerte par l'outil `dune`.

Il permet, entre autres,

- de gérer la compilation d'un projet ;
- de gérer la mise en place de tests ;
- de gérer les dépendances du projet à des bibliothèques externes ;
- de concevoir des bibliothèques.

Pour **créer un nouveau projet** `NAME`, on utilise la commande

```
dune init project NAME
```

Ceci créé un répertoire `NAME` dans le répertoire courant.

Il contient divers fichiers et sous-répertoires. En particulier, il propose un sous-répertoire `bin` contenant un programme `main.ml` doté de la fonction principale.

Il y a aussi divers fichiers `dune`, utilisés pour la configuration de la construction du projet.

En se rendant dans le répertoire du projet (`cd NAME`), on le **construit** via la commande

```
dune build
```

Cela invoque le compilateur OCaml de manière adéquate pour produire un exécutable. Celui-ci a pour chemin relatif `_build/default/bin/main.exe`.

La commande

```
dune exec NAME
```

permet de l'**exécuter**.

Il y a aussi à la racine un fichier `dune-project` contenant des informations sur le projet (auteurs, description, version, *etc.*).

Le répertoire `lib` est propice pour y mettre les différents fichiers du projet, autre que celui qui contient la fonction principale.

Soit `PROJECT` un projet OCaml structuré via `dune`. En étant dans le répertoire `PROJECT` :

- pour **nettoyer le projet** en supprimant tous les fichiers qui peuvent être régénérés par `dune` :

  ```
  dune clean
  ```

- pour **lancer un interpréteur** (et pouvoir utiliser les définitions du projet et les tester) :

  ```
  dune utop
  ```

  Les différentes définitions sont accessibles dans `utop` via le préfixe `PROJECT.MODULE.` où `MODULE` est le nom du module dans lequel figure la définition souhaitée ;

- pour **installer localement** le projet (et pouvoir l'utiliser dans d'autres projets) :

  ```
  opam pin add PROJECT .
  ```

  Il faut alors, dans les fichiers `dune` du projet utilisateur, mentionner la dépendance par l'expression `(libraries ... PROJECT)` ;

- pour **mettre à jour l'installation** du projet après une modification de celui-ci,

  ```
  opam reinstall PROJECT
  ```

- pour imprimer la **liste des projets** installés :

  ```
  opam list
  ```

10. Types

10.1. L'algèbre des types

Rappels :

- en programmation (fonctionnelle), un **type** désigne un **ensemble de valeurs** ;
- dire qu'une expression `EXP` est de type `T` est équivalent à dire qu'`EXP` **est un élément** de l'ensemble `T`.

Il existe deux sortes de types :

1. les *types scalaires*, qui sont des types atomiques et définis à l'avance dans le langage ;
2. les *types construits*, qui sont des constructions faisant intervenir des types scalaires ou construits et des **opérateurs de types**.

Dans certains langages de programmation, les types peuvent avoir des relations entre eux (par exemple, un type `T` peut être un sous-ensemble d'un type `S`).

L'ensemble des types d'un langage et des opérateurs de types est son *algèbre des types*.

La **définition d'un nouveau type** `ID` se fait par

```
type ID = OP
```

où `OP` fait intervenir des types et des opérateurs de types.

Rappelons que les **types scalaires** dont nous disposons sont

`int`, `float`, `char`, `string`, `bool`, `unit`.

Voici les *opérateurs de types* principaux que nous allons considérer (l'opérateur `->` a déjà été vu) :

Opérateur	Arité	Nom
`->`	2	Flèche
`*`	2	Produit cartésien binaire
`*`	$\geqslant 2$	Produit cartésien multiple
`{ }`	$\geqslant 1$	Produit nommé
`|`	$\geqslant 1$	Somme

10.2. Types produit

Étant donnés deux types `T1` et `T2`,

`T1 * T2`

désigne le type *produit cartésien binaire* de `T1` et `T2`.

Il contient pour valeurs les *couples* `(e1, e2)` où `e1` (resp. `e2`) est de type `T1` (resp. `T2`).

Exemple

Par la définition `type point = int * int` le type `point` contient tous les couples à coordonnées entières.

Un couple s'écrit par `(e1, e2)`. Les parenthèses sont facultatives.

Exemples

```
# (3.5, 21);;
- : float * int = (3.5, 21)
```

```
# 3.5, 21;;
- : float * int = (3.5, 21)
```

```
# (1, (2, 3));;
- : int * (int * int) = (1, (2, 3))
```

```
# ((1, 2), 3);;
- : (int * int) * int = ((1, 2), 3))
```

Si `c` est un couple, nous accèdons à sa 1re coordonnée par `fst c` et à sa 2e coordonnée par `snd c`.

Exemples

```
# let add c =
      (fst c) + (snd c);;
val add : int * int -> int = <fun>
```

```
# add (2, 4);;
- : int = 6
```

L'opérateur de types `*` est prioritaire face à `->`.

Il existe un moyen plus élégant (et plus général) pour accéder aux coordonnées de `c` au moyen de

```
let (c1, c2) = c in ...
```

Exemples

```
# let add c =
      let (c1, c2) = c in
      c1 + c2;;
val add : int * int -> int = <fun>
```

```
# let first' c =
      let (c1, _) = c in
      c1;;
val first' : 'a * 'b -> 'a = <fun>
```

L'opérateur de types `*` est **non associatif** : si `T1`, `T2` et `T3` sont trois types,

`(T1 * T2) * T3`

est un type différent de

`T1 * (T2 * T3)`

Exemple

```
# let conc_21 x =
      let (x1, x2) = x in
      let (x11, x12) = x1 in
      x11 ^ x2 ^ x12;;
val conc_21 : (string * string) * string -> string
    = <fun>
```

```
# conc_21 (("a", "b"), "c");;
- : string = "acb"
```

```
# let conc_12 x =
      let (x1, x2) = x in
      let (x21, x22) = x2 in
      x21 ^ x1 ^ x22;;
val conc_12 : string * (string * string) -> string
    = <fun>
```

```
# conc_12 ("a", ("b", "c"));;
- : string = "bac"
```

Les types `(string * string) * string` et `string * (string * string)` sont bien différents.

Pour tout $n \geqslant 3$, étant donnés des types `T1`, ..., `Tn`,

```
T1 * ... * Tn
```

désigne le type *produit cartésien multiple* de `T1`, ..., `Tn`.

Il contient pour valeurs les *n-uplets* `(e1, ..., en)` où pour tout $1 \leqslant i \leqslant n$, `ei` est de type `Ti`.

Exemple

```
type tr = int * unit * (int -> char)
```

Le type `tr` contient les triplets `(x1, x2, x3)` où `x1` est de type `int`, `x2` de type `unit` et `x3` est de type `int -> char`.

Un n-uplet s'écrit

```
(e1, ..., en)
```

Les parenthèses sont facultatives.

Exemple

```
# (0., 1, "abc", 'v');;
- : float * int * string * char = (0., 1, "abc", 'v')
```

Ceci construit un quadruplet.
Notons qu'il n'a pas été nécessaire de définir ce type au préalable.

Il est impossible d'utiliser `fst` et `snd` pour accéder aux coordonnées d'un n-uplet lorsque $n \neq 2$.

Si `p` est un tel n-uplet, nous accédons à ses coordonnées au moyen de

```
let (e1, ..., en) = p in ...
```

Exemple

```
# let p = (2.5, 3.4, -1.2) in
      let (x, y, z) = p in
          x +. z;;
- : float = 1.3
```

Il est possible de ne recueillir que la `k`$^{\text{e}}$ de ses coordonnées par

```
let (_, ..., _, ek, _, ..., _) = p in ...
```

Exemple

```
# let (_, _, x, _, _) = (1, 2, 3, 4, 5) in x + 10;;
- : int = 13
```

Le symbole `_` est le *joker*. Il ne peut pas être lié à une valeur mais accepte néanmoins syntaxiquement d'être défini.

Pour tout $n \geqslant 1$, étant donnés des types `T1`, ..., `Tn` et des identificateurs `ID1`, ..., `IDn`,

```
{ID1 : T1 ; ... ; IDn : Tn}
```

désigne le type *produit nommé* de `T1`, ..., `Tn`.

Il contient pour valeurs les *enregistrements* dont les *champs* sont `ID1`, ..., `IDn` de types respectifs `T1`, ..., `Tn`.

Exemple

```
type personne = {nom : string ; age : int}
```

Un enregistrement s'écrit

```
{ID1 = EXP1 ; ... ; IDn = EXPn}
```

où `EXP1`, ..., `EXPn` sont des expressions de types respectifs `T1`, ..., `Tn`.

Exemple

```
# {nom = "Haskell Curry" ; age = 81};;
- : personne = {nom = "Haskell Curry"; age = 81}
```

Il est possible d'accèder au champ `c` d'un enregistrement `e` par

`e.c`

Exemple

En reprenant l'exemple précédent,

```
# let p = {nom = "Alan Turing" ; age = 41} in p.nom;;
- : string = "Alan Turing"
```

est une phrase qui s'évalue en le champ `nom` de la valeur `p` de type `personne` localement liée.

Voici une fonction qui accepte deux valeurs de type `personne` et qui renvoie le nom de la personne la plus âgée (et la chaîne de caractères vide en cas d'égalité) :

```
# let nom_du_plus_age p1 p2 =
      if p1.age > p2.age then
          p1.nom
      else if p1.age < p2.age then
          p2.nom
      else
          "";;
val nom_du_plus_age : personne -> personne -> string = <fun>
```

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié. Il est seulement possible, via l'ombrage de nom, de lier un **nouveau** nom `x` à une nouvelle valeur.

Pour « modifier » un enregistrement, il faut en **reconstruire** un en prenant compte de la modification.

Si `e` est un enregistrement possédant (entre autres) des champs `c1`, ..., `cn`, l'expression

```
{e with c1 = v1 ; ... ; cn = vn}
```

est un enregistrement dont les valeurs des champs sont les mêmes que celles de ceux de `e`, sauf pour les champs `c1`, ..., `cn` dont les valeurs sont respectivement égales à `v1`, ..., `vn`.

Exemples

```
type point = {a : int ; b : int ; c : int}
```

```
# let p1 = {a = 1 ; b = 2 ; c = 3};;
val p1 : point = {a = 1; b = 2; c = 3}
```

```
# let p2 = {p1 with a = -1};;
val p2 : point = {a = -1; b = 2; c = 3}
```

```
let p3 = {p1 with b = 3 ; c = 4};;
val p3 : point = {a = 1; b = 3; c = 4}
```

Il est incorrect de définir des types enregistrements qui ont un même identificateur de champ.

Exemple

```
# type t1 = {a : int ; b : int}
type t1 = {a : int ; b : int; }
# type t2 = {a : int ; c : char}
type t2 = { a : int; c : char; }

# let f x = x.a
val f : t2 -> int = <fun>
```

Problème : le type de la fonction `f` ne peut pas être inféré correctement.

En effet, le champ `a` étant commun aux types `t1` et `t2`, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces phrases mais il néglige la définition du type `t1` (qui est plus ancienne).

Ceci est correct :

```
# {a = 2 ; c = 'y'};;
- : t2 = {a = 2; c = 'y'}
```

Ceci provoque une erreur :

```
# {a = 2 ; b = 3};;
Error: The record field label b belongs to the type t1
       but is mixed here with labels of type t2
```

Il est ainsi non recommandé de définir, dans un même espace de noms, des types produit nommés ayant des noms de champs communs.

Cette contrainte **ne s'applique plus** si les types sont définis dans des **fichiers différents** car ils appartiennent à des **espaces de noms différents**.

Nous accèdons alors aux noms des champs d'un enregistrement en les préfixant par `A.` s'ils sont d'un type défini dans un fichier nommé `A.ml`.

Exemple

```
(* A.ml *)

type a = {a : int ; b : int}
```

```
(* B.ml *)

type b = {a : int ; c : char}
```

```
(* C.ml *)

let c1 = {B.a = 2 ; B.c = 'y'}
and c2 = {A.a = 2 ; A.b = 3}
```

10.3. Types somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les 1res lettres sont des majuscules,

`Id1 | ... | Idn`

désigne le type *somme* de `Id1`, ..., `Idn`.

Il contient exactement `n` **valeurs** : `Id1`, ..., `Idn`.

Ces valeurs sont appelées *constructeurs*.

Exemple

```
type numero = Un | Deux | Trois
```

Une *valeur* d'un type somme s'écrit via son constructeur.

Exemples

```
# Deux;;
- : numero = Deux

# let plusieurs n =
      n = Deux || n = Trois;;
val plusieurs : numero -> bool = <fun>
```

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

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

Certains constructeurs d'un type somme peuvent avoir un argument. Un *argument* est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, ..., `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ...| Idk of T | ...| Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

Exemple

Soit le type `nombre` défini par

```
type nombre = Entier of int | Rationnel of int * int | Infini
```

Les constructeurs `Entier` et `Rationnel` possèdent des arguments.

Une meilleure indentation :

```
type nombre =
    |Entier of int
    |Rationnel of int * int
    |Infini
```

Une valeur d'un type somme s'écrit par son constructeur suivi de son argument (s'il en a un).

Exemples

```
# Entier 13;;
- : nombre = Entier 13
```

```
# Rationnel (2, 3);;
- : nombre = Rationnel (2, 3)
```

```
# Infini;;
- : nombre = Infini
```

Une *liste d'entiers* est la liste vide **ou bien** une cellule contenant un entier qui est attachée à une liste d'entiers.

Cette définition se traduit en le type somme **récursif** ci-contre :

```
type liste_int =
    |Vide
    |Cellule of int * liste_int
```

Exemple

Nous construisons des listes d'entiers de la manière suivante :

```
# let e = Vide;;
val e : liste_int = Vide

# let e = Cellule (1, e);;
val e : liste_int = Cellule (1, Vide)

# let e = Cellule (2, e);;
val e : liste_int = Cellule (2, Cellule (1, Vide))

# let e = Cellule (3, e);;
val e : liste_int = Cellule (3, Cellule (2, Cellule (1, Vide)))
```

Le nom `e` est finalement lié à la liste $(3, 2, 1)$.

Un *arbre binaire d'entiers* est l'arbre vide ou bien un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

Cette définition se traduit en le type somme récursif ci-contre :

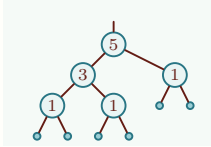
```
type arbre_b_int =
    |Vide
    |Noeud of arbre_b_int * int * arbre_b_int
```

Exemple

Nous construisons des arbres binaires d'entiers de la manière suivante :

```
# let a0 = Vide;;
val a0 : arbre_b_int = Vide
# let a1 = (Noeud (a0, 1, a0));;
val a1 : arbre_b_int = Noeud (Vide, 1, Vide)
# let a2 = (Noeud (a1, 3, a1));;
val a2 : arbre_b_int = Noeud (Noeud (Vide, 1, Vide), 3, Noeud (Vide, 1, Vide))
# let a3 = (Noeud (a2, 5, a1));;
val a3 : arbre_b_int = Noeud (Noeud (Noeud (Vide,1,Vide), 3, Noeud (Vide,1,Vide)), 5, Noeud (Vide,1,Vide))
```

Le nom `a3` est lié l'arbre binaire

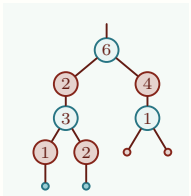


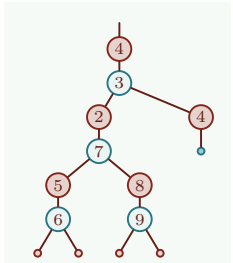
Un *arbre unaire binaire d'entiers* est

- l'arbre vide **ou bien**
- un nœud binaire contenant un entier, attaché à deux arbres unaires binaires dont les racines sont d'arités 1 **ou bien**
- un nœud unaire contenant un entier, attaché à un arbre unaire binaire dont la racine est d'arité 2.

Exemples

Voici deux arbres unaires binaires, respectivement dont la racine est d'arité 2 et dont la racine est d'arité 1 :





La définition de ces objets se traduit en les définitions de types suivantes.

Nous commençons par définir deux types pour représenter les arbres unaires binaires en séparant les cas en fonction de l'arité de la racine :

```
type arbre_1 =
    |Vide1
    |Noeud1 of int * arbre_2

and arbre_2 =
    |Vide2
    |Noeud2 of arbre_1 * int * arbre_1
```

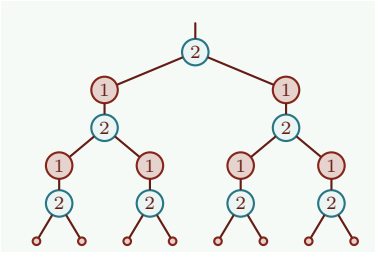
Le mot-clé `and` permet de réaliser des **définitions de types simultanées** : les types `arbre_1` et `arbre_2` sont en effet **mutuellement récursifs**.

Étant donné qu'un arbre unaire binaire peut être vide, avoir une racine unaire ou bien une racine binaire, nous définissons le type final par la somme (c'est-à-dire l'**union disjointe**) de ces trois possibilités :

```
type arbre_12 =
    |Vide
    |Arbre1 of arbre_1
    |Arbre2 of arbre_2
```

Exemple

Écrivons une expression de type `arbre_12` qui représente l'arbre




```
# let a = Vide1 in
  let a = Noeud2 (a, 2, a) in
  let a = Noeud1 (1, a) in
  let a = Noeud2 (a, 2, a) in
  let a = Noeud1 (1, a) in
  Arbre2 (Noeud2 (a, 2, a));;
```

```
- : arbre_12 =
Arbre2
 (Noeud2
   (Noeud1 (1,
     Noeud2 (Noeud1 (1, Noeud2 (Vide1, 2, Vide1)), 2,
      Noeud1 (1, Noeud2 (Vide1, 2, Vide1)))),
   2,
   Noeud1 (1,
    Noeud2 (Noeud1 (1, Noeud2 (Vide1, 2, Vide1)), 2,
     Noeud1 (1, Noeud2 (Vide1, 2, Vide1))))))
```