

Exemple 2 – arbres binaires d'entiers

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.

Exemple 2 — arbres binaires d'entiers

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 **récuratif** suivant :

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

Exemple 2 — arbres binaires d'entiers

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 **récuratif** suivant :

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

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;  
val a1 : arbre_b_int = Noeud (Vide, 1, Vide)
```

Exemple 2 — arbres binaires d'entiers

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 **récuratif** suivant :

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

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;  
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))
```

Exemple 2 — arbres binaires d'entiers

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 **récurif** suivant :

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

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;  
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));;  
  Noeud (Noeud (Noeud (Vide, 1, Vide), 3, Noeud (Vide, 1, Vide)), 5,  
    Noeud (Vide, 1, Vide))
```

Exemple 2 – arbres binaires d'entiers

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.

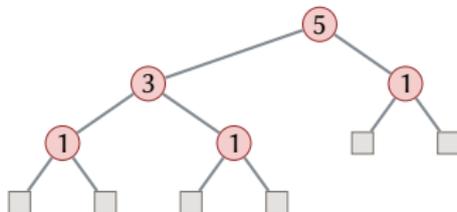
Cette définition se traduit en le type somme **récurif** suivant :

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

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;  
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));;  
  Noeud (Noeud (Noeud (Vide, 1, Vide), 3, Noeud (Vide, 1, Vide)), 5,  
  Noeud (Vide, 1, Vide))
```

Le nom **a3** est lié l'arbre binaire de valeur



Exemple 3 — arbres unaires binaires d'entiers

Un **arbre unaire binaire d'entiers** est

- ▶ l'arbre vide **ou bien**

Exemple 3 — arbres unaires binaires d'entiers

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**

Exemple 3 — arbres unaires binaires d'entiers

Un **arbre unaire binaire d'entiers** est

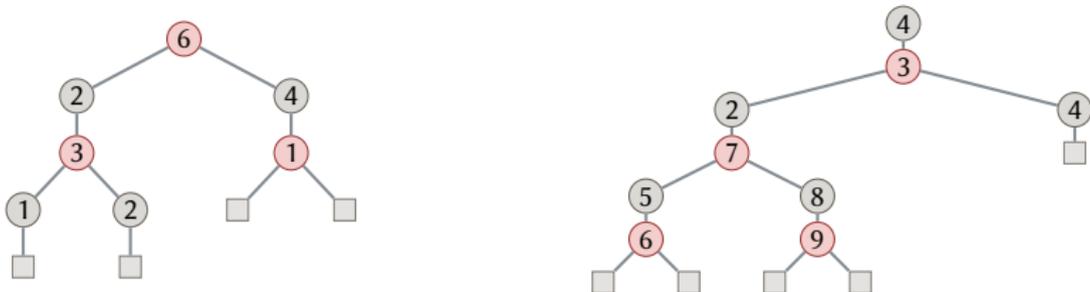
- ▶ l'arbre vide **ou bien**
- ▶ un nœud binaire contenant un entier, attaché à deux arbres unaires binaires dont les racines sont d'arités 1 **ou bien**
- ▶ un nœud unaire contenant un entier, attaché à un arbre unaire binaire dont la racine est d'arité 2.

Exemple 3 — arbres unaires binaires d'entiers

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.

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



Exemple 3 — arbres unaires binaires d'entiers

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

Exemple 3 — arbres unaires binaires d'entiers

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

On commence 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
```

Exemple 3 — arbres unaires binaires d'entiers

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

On commence 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**.

Exemple 3 — arbres unaires binaires d'entiers

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

On commence 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
```

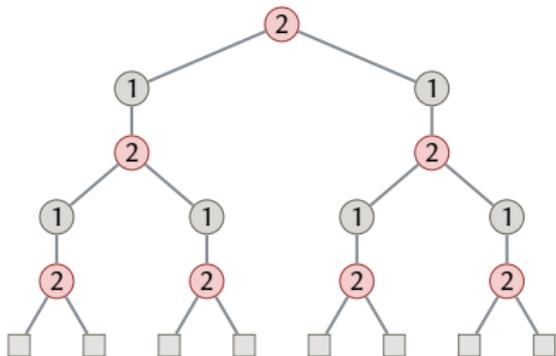
Le mot-clé `and` permet de réaliser des **définitions de types simultanées** : les types `arbre_1` et `arbre_2` sont en effet **mutuellement récursifs**.

On se base sur la définition des arbres unaires binaires pour construire finalement le type recherché :

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

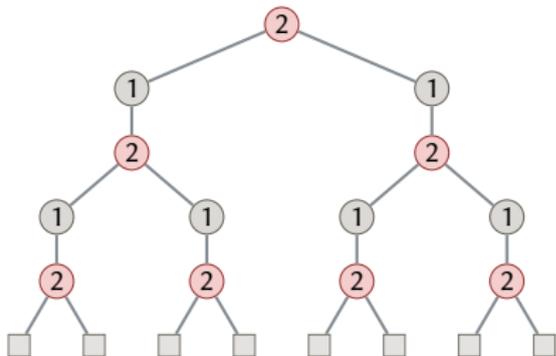
Exemple 3 — arbres unaires binaires d'entiers

Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



Exemple 3 — arbres unaires binaires d'entiers

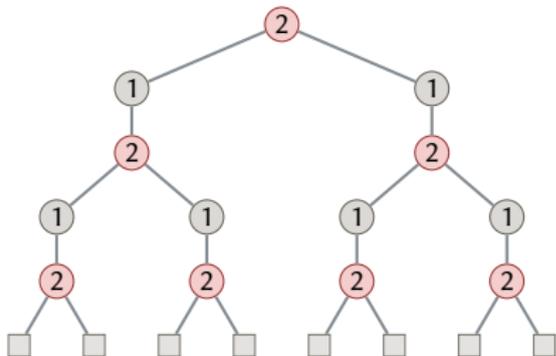
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in  
  let a1 = (Noeud1 (1, a2)) in
```

Exemple 3 — arbres unaires binaires d'entiers

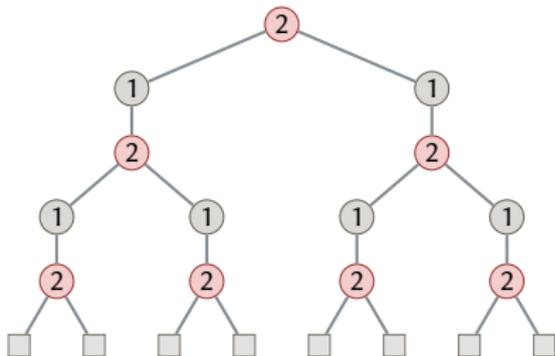
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
  let a1 = (Noeud1 (1, a2)) in
    let a22 = (Noeud2 (a1, 2, a1)) in
```

Exemple 3 — arbres unaires binaires d'entiers

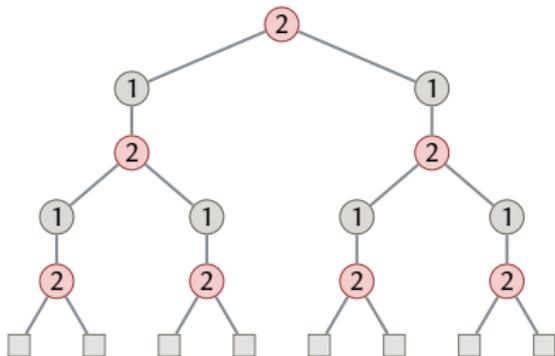
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
  let a1 = (Noeud1 (1, a2)) in
    let a22 = (Noeud2 (a1, 2, a1)) in
      let a11 = (Noeud1 (1, a22)) in
```

Exemple 3 — arbres unaires binaires d'entiers

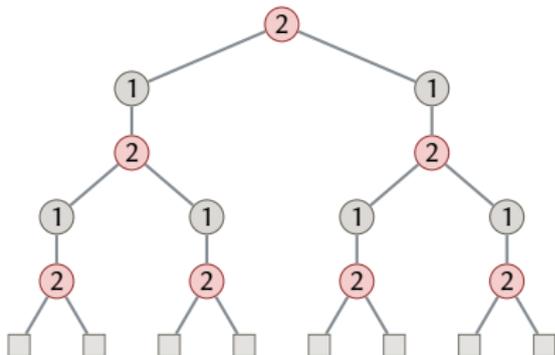
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
  let a1 = (Noeud1 (1, a2)) in
    let a22 = (Noeud2 (a1, 2, a1)) in
      let a11 = (Noeud1 (1, a22)) in
        (Arbre2 (Noeud2 (a11, 2, a11))));;
```

Exemple 3 — arbres unaires binaires d'entiers

Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
  let a1 = (Noeud1 (1, a2)) in
    let a22 = (Noeud2 (a1, 2, a1)) in
      let a11 = (Noeud1 (1, a22)) in
        (Arbre2 (Noeud2 (a11, 2, a11))));;
- : 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)))))
```

Plan

Types

L'algèbre des types

Types produit

Types somme

Types paramétrés

Paramètres dans les types

Tout comme les fonctions qui admettent des paramètres (voués à être substitués par des valeurs), il est possible de définir des types avec des paramètres (voués à être substitués par des types).

Paramètres dans les types

Tout comme les fonctions qui admettent des paramètres (voués à être substitués par des valeurs), il est possible de définir des types avec des paramètres (voués à être substitués par des types).

On parle alors de **types paramétrés**.

Paramètres dans les types

Tout comme les fonctions qui admettent des paramètres (voués à être substitués par des valeurs), il est possible de définir des types avec des paramètres (voués à être substitués par des types).

On parle alors de **types paramétrés**.

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

```
type ('P1, ..., 'Pn) ID = OP
```

où `P1, ..., Pn` sont des identificateurs et `OP` fait intervenir des types, des opérateurs de types et `'P1, ..., 'Pn`.

Paramètres dans les types

Tout comme les fonctions qui admettent des paramètres (voués à être substitués par des valeurs), il est possible de définir des types avec des paramètres (voués à être substitués par des types).

On parle alors de **types paramétrés**.

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

```
type ('P1, ..., 'Pn) ID = OP
```

où `P1, ..., Pn` sont des identificateurs et `OP` fait intervenir des types, des opérateurs de types et `'P1, ..., 'Pn`.

Les `'P1, ..., 'Pn` sont des **paramètres de types**.

Paramètres dans les types

Tout comme les fonctions qui admettent des paramètres (voués à être substitués par des valeurs), il est possible de définir des types avec des paramètres (voués à être substitués par des types).

On parle alors de **types paramétrés**.

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

```
type ('P1, ..., 'Pn) ID = OP
```

où `P1, ..., Pn` sont des identificateurs et `OP` fait intervenir des types, des opérateurs de types et `'P1, ..., 'Pn`.

Les `'P1, ..., 'Pn` sont des **paramètres de types**.

Lorsque `n = 1`, la définition se fait simplement (sans parenthèses) par

```
type 'P1 ID = OP
```

Rôle des paramètres de type

Supposons que T soit un type défini par

```
type ('P1, ..., 'Pn) T = OP
```

Rôle des paramètres de type

Supposons que T soit un type défini par

$$\text{type } ('P_1, \dots, 'P_n) T = OP$$

On dit que T est **paramétré** par $'P_1, \dots, 'P_n$.

Rôle des paramètres de type

Supposons que T soit un type défini par

$$\text{type } ('P_1, \dots, 'P_n) T = OP$$

On dit que T **est paramétré** par $'P_1, \dots, 'P_n$.

Dans la définition de T , les occurrences de $'P_i$, $1 \leq i \leq n$, qui y figurent peuvent se penser comme étant **n'importe quel type**.

Rôle des paramètres de type

Supposons que T soit un type défini par

$$\text{type } ('P_1, \dots, 'P_n) T = OP$$

On dit que T **est paramétré** par $'P_1, \dots, 'P_n$.

Dans la définition de T , les occurrences de $'P_i$, $1 \leq i \leq n$, qui y figurent peuvent se penser comme étant **n'importe quel type**.

Un type paramétré désigne ainsi un ensemble de types.

Rôle des paramètres de type

Supposons que T soit un type défini par

$$\text{type } ('P_1, \dots, 'P_n) T = OP$$

On dit que T **est paramétré** par $'P_1, \dots, 'P_n$.

Dans la définition de T , les occurrences de $'P_i$, $1 \leq i \leq n$, qui y figurent peuvent se penser comme étant **n'importe quel type**.

Un type paramétré désigne ainsi un ensemble de types.

Ce mécanisme de définition de types avec des paramètres permet de construire des types faisant intervenir

1. des types scalaires (vus comme **des constantes**);

Rôle des paramètres de type

Supposons que T soit un type défini par

$$\text{type } ('P_1, \dots, 'P_n) T = OP$$

On dit que T **est paramétré** par $'P_1, \dots, 'P_n$.

Dans la définition de T , les occurrences de $'P_i$, $1 \leq i \leq n$, qui y figurent peuvent se penser comme étant **n'importe quel type**.

Un type paramétré désigne ainsi un ensemble de types.

Ce mécanisme de définition de types avec des paramètres permet de construire des types faisant intervenir

1. des types scalaires (vus comme **des constantes**);
2. des paramètres de types (vus comme **des variables**);

Rôle des paramètres de type

Supposons que T soit un type défini par

$$\text{type } ('P_1, \dots, 'P_n) T = OP$$

On dit que T **est paramétré** par $'P_1, \dots, 'P_n$.

Dans la définition de T , les occurrences de $'P_i$, $1 \leq i \leq n$, qui y figurent peuvent se penser comme étant **n'importe quel type**.

Un type paramétré désigne ainsi un ensemble de types.

Ce mécanisme de définition de types avec des paramètres permet de construire des types faisant intervenir

1. des types scalaires (vus comme **des constantes**);
2. des paramètres de types (vus comme **des variables**);
3. des opérateurs de types.

Exemple 1 — listes génériques

Le type à un paramètre

```
type 'e liste = Vide | Cellule of 'e * 'e liste
```

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque *e*. On obtient ainsi des **listes génériques**.

Exemple 1 — listes génériques

Le type à un paramètre

```
type 'e liste = Vide | Cellule of 'e * 'e liste
```

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque *e*. On obtient ainsi des **listes génériques**.

Attention : les listes génériques sont **homogènes**, c.-à-d. les éléments de ses cellules sont d'un type quelconque mais tous d'un même type.

Exemple 1 — listes génériques

Le type à un paramètre

```
type 'e liste = Vide | Cellule of 'e * 'e liste
```

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque *e*. On obtient ainsi des **listes génériques**.

Attention : les listes génériques sont **homogènes**, c.-à-d. les éléments de ses cellules sont d'un type quelconque mais tous d'un même type.

Une liste de caractères :

```
# (Cellule ('a', (Cellule ('b', Vide))));;  
- : char liste = Cellule ('a', Cellule ('b', Vide))
```

Exemple 1 — listes génériques

Le type à un paramètre

```
type 'e liste = Vide | Cellule of 'e * 'e liste
```

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque *e*. On obtient ainsi des **listes génériques**.

Attention : les listes génériques sont **homogènes**, c.-à-d. les éléments de ses cellules sont d'un type quelconque mais tous d'un même type.

Une liste de caractères :

```
# (Cellule ('a', (Cellule ('b', Vide))));;  
- : char liste = Cellule ('a', Cellule ('b', Vide))
```

Une liste de listes d'entiers :

```
# (Cellule ((Cellule (1, Vide)), Vide));;  
- : int liste liste = Cellule (Cellule (1, Vide), Vide)
```

Exemple 1 — listes génériques

Le type à un paramètre

```
type 'e liste = Vide | Cellule of 'e * 'e liste
```

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque *e*. On obtient ainsi des **listes génériques**.

Attention : les listes génériques sont **homogènes**, c.-à-d. les éléments de ses cellules sont d'un type quelconque mais tous d'un même type.

Une liste de caractères :

```
# (Cellule ('a', (Cellule ('b', Vide))));;  
- : char liste = Cellule ('a', Cellule ('b', Vide))
```

Une liste de listes d'entiers :

```
# (Cellule ((Cellule (1, Vide)), Vide));;  
- : int liste liste = Cellule (Cellule (1, Vide), Vide)
```

Une liste dont le type des éléments n'est pas déterminé :

```
# Vide;;  
- : 'a liste = Vide
```

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;  
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;  
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

```
type 't couple_autre = {x : 't ; y : 't}
```

dans lequel les deux coordonnées doivent être d'un même type.

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;  
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

```
type 't couple_autre = {x : 't ; y : 't}
```

dans lequel les deux coordonnées doivent être d'un même type.

L'interpréteur donne sa réponse en renommant les paramètres de types en 'a, 'b, ..., (« alpha », « beta », ...) selon leur ordre d'apparition.

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;  
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

```
type 't couple_autre = {x : 't ; y : 't}
```

dans lequel les deux coordonnées doivent être d'un même type.

L'interpréteur donne sa réponse en renommant les paramètres de types en 'a, 'b, ..., (« alpha », « beta », ...) selon leur ordre d'apparition.

```
# let c = {x = 4 ; y = 'd'};;  
val c : (int, char) couple = {x = 4; y = 'd'}
```

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;  
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

```
type 't couple_autre = {x : 't ; y : 't}
```

dans lequel les deux coordonnées doivent être d'un même type.

L'interpréteur donne sa réponse en renommant les paramètres de types en 'a, 'b, ..., (« alpha », « beta », ...) selon leur ordre d'apparition.

```
# let c = {x = 4 ; y = 'd'};;  
val c : (int, char) couple = {x = 4; y = 'd'}  
  
# {c with y = 'e'};;  
- : (int, char) couple = {x = 4; y = 'e'}
```

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;  
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

```
type 't couple_autre = {x : 't ; y : 't}
```

dans lequel les deux coordonnées doivent être d'un même type.

L'interpréteur donne sa réponse en renommant les paramètres de types en 'a, 'b, ..., (« alpha », « beta », ...) selon leur ordre d'apparition.

```
# let c = {x = 4 ; y = 'd'};;  
val c : (int, char) couple = {x = 4; y = 'd'}  
  
# {c with y = 'e'};;  
- : (int, char) couple = {x = 4; y = 'e'}  
  
# {c with y = 2.2};;  
- : (int, float) couple = {x = 4; y = 2.2}
```

Exemple 3 – arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- ▶ les nœud unaires contiennent tous des valeurs d'un même type;

Exemple 3 — arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- ▶ les nœud unaires contiennent tous des valeurs d'un même type;
- ▶ les nœuds binaires contiennent tous des couples dont les coordonnées peuvent être de types différents.

Exemple 3 — arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- ▶ les nœud unaires contiennent tous des valeurs d'un même type;
- ▶ les nœuds binaires contiennent tous des couples dont les coordonnées peuvent être de types différents.

Nous avons ainsi besoin de trois paramètres de types : 'u pour le type des valeurs des nœuds unaires, et 'x et 'y pour ceux des nœuds binaires.

Exemple 3 — arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- ▶ les nœuds unaires contiennent tous des valeurs d'un même type;
- ▶ les nœuds binaires contiennent tous des couples dont les coordonnées peuvent être de types différents.

Nous avons ainsi besoin de trois paramètres de types : 'u pour le type des valeurs des nœuds unaires, et 'x et 'y pour ceux des nœuds binaires.

```
type ('u, 'x, 'y) arbre_1 =  
  Vide1 |  
  Noeud1 of 'u * ('u, 'x, 'y) arbre_2
```

Exemple 3 – arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- ▶ les nœuds unaires contiennent tous des valeurs d'un même type;
- ▶ les nœuds binaires contiennent tous des couples dont les coordonnées peuvent être de types différents.

Nous avons ainsi besoin de trois paramètres de types : 'u pour le type des valeurs des nœuds unaires, et 'x et 'y pour ceux des nœuds binaires.

```
type ('u, 'x, 'y) arbre_1 =  
  Vide1 |  
  Noeud1 of 'u * ('u, 'x, 'y) arbre_2  
and ('u, 'x, 'y) arbre_2 =  
  Vide2 |  
  Noeud2 of ('u, 'x, 'y) arbre_1 * ('x * 'y) * ('u, 'x, 'y) arbre_1
```

Exemple 3 – arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- ▶ les nœuds unaires contiennent tous des valeurs d'un même type;
- ▶ les nœuds binaires contiennent tous des couples dont les coordonnées peuvent être de types différents.

Nous avons ainsi besoin de trois paramètres de types : 'u pour le type des valeurs des nœuds unaires, et 'x et 'y pour ceux des nœuds binaires.

```
type ('u, 'x, 'y) arbre_1 =  
  Vide1 |  
  Noeud1 of 'u * ('u, 'x, 'y) arbre_2  
and ('u, 'x, 'y) arbre_2 =  
  Vide2 |  
  Noeud2 of ('u, 'x, 'y) arbre_1 * ('x * 'y) * ('u, 'x, 'y) arbre_1  
  
type ('u, 'x, 'y) arbre_12 =  
  Vide |  
  Arbre1 of ('u, 'x, 'y) arbre_1 |  
  Arbre2 of ('u, 'x, 'y) arbre_2
```

Exemple 4 — Mots

On cherche à représenter des **mots** sur un alphabet quelconque.

Exemple 4 — Mots

On cherche à représenter des **mots** sur un alphabet quelconque.

Pour rappel, un mot est une suite finie de lettres qui appartiennent à un alphabet donné.

Exemple 4 — Mots

On cherche à représenter des **mots** sur un alphabet quelconque.

Pour rappel, un mot est une suite finie de lettres qui appartiennent à un alphabet donné.

Les lettres d'un mot u sont indicées de 1 à sa longueur $|u|$ et pour tout $1 \leq i \leq |u|$, u_i désigne la i^{e} lettre de u .

Par exemple, le mot $u := \text{baacba}$ vérifie $u_1 = \text{b}$ et $u_4 = \text{c}$.

Exemple 4 — Mots

On cherche à représenter des **mots** sur un alphabet quelconque.

Pour rappel, un mot est une suite finie de lettres qui appartiennent à un alphabet donné.

Les lettres d'un mot u sont indicées de 1 à sa longueur $|u|$ et pour tout $1 \leq i \leq |u|$, u_i désigne la i^{e} lettre de u .

Par exemple, le mot $u := \text{baacba}$ vérifie $u_1 = \text{b}$ et $u_4 = \text{c}$.

Usuellement, en **programmation impérative** (et donc dans un style mutable), un mot u est représenté par un tableau `tab` qui contient ses lettres ($u_i = \text{tab}[i - 1]$). Ceci supporte les opérations de lecture et modification de lettres, ainsi que de concaténation.

Exemple 4 — Mots

On cherche à représenter des **mots** sur un alphabet quelconque.

Pour rappel, un mot est une suite finie de lettres qui appartiennent à un alphabet donné.

Les lettres d'un mot u sont indicées de 1 à sa longueur $|u|$ et pour tout $1 \leq i \leq |u|$, u_i désigne la i^{e} lettre de u .

Par exemple, le mot $u := \text{baacba}$ vérifie $u_1 = \text{b}$ et $u_4 = \text{c}$.

Usuellement, en **programmation impérative** (et donc dans un style mutable), un mot u est représenté par un tableau `tab` qui contient ses lettres ($u_i = \text{tab}[i - 1]$). Ceci supporte les opérations de lecture et modification de lettres, ainsi que de concaténation.

En **programmation fonctionnelle** (et donc dans un style non mutable), il est possible d'utiliser une liste simplement chaînée pour représenter les lettres du mot. Il est cependant possible de faire mieux.

Exemple 4 — Mots

Idée principale : pour connaître un mot u , il suffit de connaître pour toute position i la lettre u_i de u .

Exemple 4 — Mots

Idée principale : pour connaître un mot u , il suffit de connaître pour toute position i la lettre u_i de u .

Un mot est donc une **fonction** qui envoie chaque position i sur une lettre.

Exemple 4 — Mots

Idée principale : pour connaître un mot u , il suffit de connaître pour toute position i la lettre u_i de u .

Un mot est donc une **fonction** qui envoie chaque position i sur une lettre.

On obtient donc le type à un paramètre suivant

```
type 'a mot = {  
    lettres : int -> 'a;  
    longueur : int  
}
```

Exemple 4 — Mots

Idée principale : pour connaître un mot u , il suffit de connaître pour toute position i la lettre u_i de u .

Un mot est donc une **fonction** qui envoie chaque position i sur une lettre.

On obtient donc le type à un paramètre suivant

```
type 'a mot = {  
    lettres : int -> 'a;  
    longueur : int  
}
```

Quelques explications :

- ▶ le paramètre de type `'a` renseigne sur le type des lettres du mot;

Exemple 4 — Mots

Idée principale : pour connaître un mot u , il suffit de connaître pour toute position i la lettre u_i de u .

Un mot est donc une **fonction** qui envoie chaque position i sur une lettre.

On obtient donc le type à un paramètre suivant

```
type 'a mot = {  
    lettres : int -> 'a;  
    longueur : int  
}
```

Quelques explications :

- ▶ le paramètre de type `'a` renseigne sur le type des lettres du mot ;
- ▶ le champ `lettres` est la fonction dont le rôle a été décrit plus haut ;

Exemple 4 — Mots

Idée principale : pour connaître un mot u , il suffit de connaître pour toute position i la lettre u_i de u .

Un mot est donc une **fonction** qui envoie chaque position i sur une lettre.

On obtient donc le type à un paramètre suivant

```
type 'a mot = {  
    lettres : int -> 'a;  
    longueur : int  
}
```

Quelques explications :

- ▶ le paramètre de type `'a` renseigne sur le type des lettres du mot ;
- ▶ le champ `lettres` est la fonction dont le rôle a été décrit plus haut ;
- ▶ le champ `longueur` contient la longueur du mot (ceci est nécessaire car on ne peut pas déduire la longueur du mot uniquement depuis la fonction `lettres`).

Exemple 4 — Mots

```
# let mot_1 = {  
  lettres =  
  (fun i ->  
    if i = 2 || i = 3 then  
      'a'  
    else  
      'b'  
  );  
  longueur = 5  
};;  
val mot_1 : char mot = {lettres = <fun>; longueur = 5}
```

Ceci lie au nom `mot_1` le mot `baabb`.

Exemple 4 — Mots

```
# let mot_1 = {
  lettres =
    (fun i ->
      if i = 2 || i = 3 then
        'a'
      else
        'b'
    );
  longueur = 5
};;
val mot_1 : char mot = {lettres = <fun>; longueur = 5}
```

Ceci lie au nom `mot_1` le mot `baabb`.

```
# let mot_2 = {
  lettres = (fun i -> i mod 2 = 0);
  longueur = 4294967296
};;
val mot_2 : bool mot = {lettres = <fun>; longueur = 4294967296}
```

Ceci lie au nom `mot_2` le mot `FTFTF...` de longueur 4294967296 (où F désigne `false` et T désigne `true`).

La quantité de mémoire utilisée est négligeable devant la longueur du mot.

Exemple 5 — Images

Cette idée de représenter les mots par des fonctions peut s'appliquer à la représentation d'images.

Exemple 5 — Images

Cette idée de représenter les mots par des fonctions peut s'appliquer à la **représentation d'images**.

Une image (dans le plan) peut se voir comme un tableau à deux dimensions de pixels.

Exemple 5 – Images

Cette idée de représenter les mots par des fonctions peut s'appliquer à la **représentation d'images**.

Une image (dans le plan) peut se voir comme un tableau à deux dimensions de pixels.

Mieux : une image peut aussi se voir comme une **fonction** qui envoie chaque point sur une couleur. Ceci permet de représenter des images de très hautes définitions avec peu de mémoire.

Exemple 5 – Images

Cette idée de représenter les mots par des fonctions peut s'appliquer à la **représentation d'images**.

Une image (dans le plan) peut se voir comme un tableau à deux dimensions de pixels.

Mieux : une image peut aussi se voir comme une **fonction** qui envoie chaque point sur une couleur. Ceci permet de représenter des images de très hautes définitions avec peu de mémoire.

On obtient les types

```
type point = int * int
```

Exemple 5 – Images

Cette idée de représenter les mots par des fonctions peut s'appliquer à la **représentation d'images**.

Une image (dans le plan) peut se voir comme un tableau à deux dimensions de pixels.

Mieux : une image peut aussi se voir comme une **fonction** qui envoie chaque point sur une couleur. Ceci permet de représenter des images de très hautes définitions avec peu de mémoire.

On obtient les types

```
type point = int * int

type 'a image = {
  contenu_pixels : point -> 'a;
  largeur : int;
  hauteur : int
}
```

Exemple 5 – Images

Avec de plus la définition du type

```
type couleur = {rouge : int; vert : int; bleu : int}
```

toute valeur de type `couleur` `image` fait le travail.

Exemple 5 — Images

Avec de plus la définition du type

```
type couleur = {rouge : int; vert : int; bleu : int}
```

toute valeur de type `couleur image` fait le travail.

Par exemple,

```
# let im_1 = {
  contenus_pixels =
    (fun p ->
      let (x, y) = p in
      if x = y then
        {rouge = 127; vert = 127; bleu = 127}
      else
        {rouge = 0; vert = 0; bleu = 0}
    );
  largeur = 16;
  hauteur = 16
}
val im_1 : couleur image = {contenus_pixels = <fun>;
  largeur = 16; hauteur = 16}
```

Ceci lie au nom `im_1` une image 16x16 dont les pixels sur la diagonale sont gris et les autres noirs.

Exemple 5 — Images

Voici un autre exemple :

```
# let im_2 = {
  contenu_pixels =
  (fun p ->
    let (x, y) = p in
    if (sqrt ((float_of_int x) ** 2.
              +. (float_of_int y) ** 2.)) <= 1024. then
      {rouge = 0; vert = 0; bleu = 0}
    else
      {rouge = 255; vert = 255; bleu = 255}
  );
  largeur = 1048576;
  hauteur = 1048576
}
val im_2 : couleur image = {contenu_pixels = <fun>;
  largeur = 1048576; hauteur = 1048576}
```

Ceci lie au nom `im_2` une image (de très haute définition) d'un disque noir sur un fond blanc.

Axe 3 : concepts avancés

Notions

Listes

λ -calcul

Plan

Notions

Récurtivité

Filtrage

Fonctions d'ordre supérieur

Polymorphisme

Stratégies d'évaluation

Plan

Notions

Récurtivité

Filtrage

Fonctions d'ordre supérieur

Polymorphisme

Stratégies d'évaluation

Définitions récursives

```
# let fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;  
Error: Unbound value fact
```

Cette définition pose problème : l'identificateur `fact` n'est lié à aucune valeur lorsque l'on fait appel à sa valeur en l. 5.

Définitions récursives

```
# let fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Error: Unbound value fact

Cette définition pose problème : l'identificateur `fact` n'est lié à aucune valeur lorsque l'on fait appel à sa valeur en l. 5.

Pour pouvoir réaliser des **définitions récursives** (c.-à-d. lier des valeurs à un nom en faisant référence au nom lui-même), on utilise la construction

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

où `ID` est un identificateur, `P1, ..., Pn` sont ses paramètres et `EXP` est une expression.

Définitions récursives

```
# let fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Error: Unbound value fact

Cette définition pose problème : l'identificateur `fact` n'est lié à aucune valeur lorsque l'on fait appel à sa valeur en l. 5.

Pour pouvoir réaliser des **définitions récursives** (c.-à-d. lier des valeurs à un nom en faisant référence au nom lui-même), on utilise la construction

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

où `ID` est un identificateur, `P1, ..., Pn` sont ses paramètres et `EXP` est une expression.

```
# let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;  
val fact : int -> int = <fun>
```

Ceci définit bien la fonction factorielle.

```
# (fact 7);;  
- : int = 5040
```

Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
```

```
# let rec x =
  let x = 20 in
    x + x;;
```

```
# let x = 10 in
  (let x = 20 in
    x)
  + x;;
```

```
# let rec x =
  (let x = 20 in
    x)
  + x;;
```

Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
Warning 26: unused variable x.
- : int = 40
```

```
# let rec x =
  let x = 20 in
    x + x;;
```

```
# let x = 10 in
  (let x = 20 in
    x)
  + x;;
```

```
# let rec x =
  (let x = 20 in
    x)
  + x;;
```

Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
Warning 26: unused variable x.
- : int = 40
```

```
# let rec x =
  let x = 20 in
    x + x;;
```

```
# let x = 10 in
  (let x = 20 in
    x)
  + x;;
- : int = 30
```

```
# let rec x =
  (let x = 20 in
    x)
  + x;;
```

Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
Warning 26: unused variable x.
- : int = 40
```

```
# let rec x =
  let x = 20 in
    x + x;;
val x : int = 40
```

```
# let x = 10 in
  (let x = 20 in
    x)
  + x;;
- : int = 30
```

```
# let rec x =
  (let x = 20 in
   x)
  + x;;
```

Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
Warning 26: unused variable x.
- : int = 40
```

```
# let x = 10 in
  (let x = 20 in
    x)
  + x;;
- : int = 30
```

```
# let rec x =
  let x = 20 in
    x + x;;
val x : int = 40
```

```
# let rec x =
  (let x = 20 in
    x)
  + x;;
Error: This kind of expression is
not allowed as right-hand side of
'let rec'
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Error: Unbound value f

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
Stack overflow during evaluation  
(looping recursion?).
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
Stack overflow during evaluation  
(looping recursion?).
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
Stack overflow during evaluation  
(looping recursion?).
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;
```

Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
Stack overflow during evaluation  
(looping recursion?).
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
Stack overflow during evaluation  
(looping recursion?).
```

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Suite d'appels pour le calcul de `(zero 4)` :

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Suite d'appels pour le calcul de
`(zero 4)` :
`(zero 4)`

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Suite d'appels pour le calcul de `(zero 4)` :
`(zero 4) → (un 3)`

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Suite d'appels pour le calcul de `(zero 4)` :

`(zero 4) → (un 3) → (deux 2)`

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Suite d'appels pour le calcul de

`(zero 4)` :

$$\begin{aligned} (\text{zero } 4) &\rightarrow (\text{un } 3) \rightarrow (\text{deux } 2) \\ &\rightarrow (\text{zero } 1) \end{aligned}$$

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Suite d'appels pour le calcul de

`(zero 4)` :

$$\begin{aligned} &(\text{zero } 4) \rightarrow (\text{un } 3) \rightarrow (\text{deux } 2) \\ &\qquad\qquad\qquad \rightarrow (\text{zero } 1) \rightarrow (\text{un } 0) \end{aligned}$$

Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Suite d'appels pour le calcul de

`(zero 4)` :

$$\begin{aligned} (\text{zero } 4) &\rightarrow (\text{un } 3) \rightarrow (\text{deux } 2) \\ &\rightarrow (\text{zero } 1) \rightarrow (\text{un } 0) \\ &\rightarrow \text{"un"}. \end{aligned}$$

Simulation des instructions de boucle

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

Simulation des instructions de boucle

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

En effet, l'effet d'une suite d'instructions (en pseudo-code) utilisant une boucle « tant que » se traduit au moyen d'une définition d'une fonction récursive utilisant une conditionnelle et d'un appel à cette fonction :

Ici, C est une condition et I est une expression.

Simulation des instructions de boucle

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

En effet, l'effet d'une suite d'instructions (en pseudo-code) utilisant une boucle « tant que » se traduit au moyen d'une définition d'une fonction récursive utilisant une conditionnelle et d'un appel à cette fonction :

```
Tant que  $C$  :  
     $I$   
Fin
```

Ici, C est une condition et I est une expression.

Simulation des instructions de boucle

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

En effet, l'effet d'une suite d'instructions (en pseudo-code) utilisant une boucle « tant que » se traduit au moyen d'une définition d'une fonction récursive utilisant une conditionnelle et d'un appel à cette fonction :

```
Tant que  $C$  :  
     $I$   
Fin
```

```
Fonction rec f :  
    Si  $C$  :  
         $I$   
        Appel à  $f$   
    Fin  
Fin  
Appel à  $f$ 
```

Ici, C est une condition et I est une expression.

Simulation des instructions de boucle — exemples

Boucle `while` simple :

Fonction C

```
int triangle(int n) {
    int i, res;
    res = 0;
    i = n;
    while (i >= 1) {
        res += i;
        i -= 1;
    }
    return res;
}
```

Fonction CAML

```
let triangle n =
  let rec aux i =
    if i >= 1 then
      i + (aux (i - 1))
    else
      0
  in
  (aux n)
```

Simulation des instructions de boucle — exemples

Boucle `while` simple :

Fonction C

```
int triangle(int n) {
    int i, res;
    res = 0;
    i = n;
    while (i >= 1) {
        res += i;
        i -= 1;
    }
    return res;
}
```

Fonction CAML

```
let triangle n =
  let rec aux i =
    if i >= 1 then
      i + (aux (i - 1))
    else
      0
  in
  (aux n)
```

Boucle `for` simple :

Fonction C

```
int somme_pairs(int n) {
    int i, res;
    res = 0;
    for (i = 0 ; i <= n ; i += 2) {
        res += i;
    }
    return res;
}
```

Fonction CAML

```
let somme_pairs n =
  let rec aux i =
    if i <= n then
      i + (aux (i + 2))
    else
      0
  in
  (aux 0)
```

Récurtivité terminale – motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et
l'appel

(fact 4)

Récurivité terminale – motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et
l'appel

(fact 4)

Cette dernière expression s'**évalue** au fil des appels récursifs en

Récurtivité terminale – motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et
l'appel

(fact 4)

Cette dernière expression s'**évalue** au fil des appels récursifs en

(fact 4)

Réversivité terminale – motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et
l'appel

(fact 4)

Cette dernière expression s'évalue au fil des appels récursifs en

(fact 4) → 4 * (fact 3)

Réversivité terminale – motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et
l'appel

(fact 4)

Cette dernière expression s'évalue au fil des appels récursifs en

(fact 4) → 4 * (fact 3) → 4 * 3 * (fact 2)

Récurtivité terminale – motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et
l'appel

(fact 4)

Cette dernière expression s'évalue au fil des appels récursifs en

$$\begin{aligned}(\text{fact } 4) &\rightarrow 4 * (\text{fact } 3) \rightarrow 4 * 3 * (\text{fact } 2) \\ &\rightarrow 4 * 3 * 2 * (\text{fact } 1)\end{aligned}$$

Réversivité terminale – motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et
l'appel

(fact 4)

Cette dernière expression s'évalue au fil des appels récursifs en

$$\begin{aligned}(\text{fact } 4) &\rightarrow 4 * (\text{fact } 3) \rightarrow 4 * 3 * (\text{fact } 2) \\ &\rightarrow 4 * 3 * 2 * (\text{fact } 1) \rightarrow 4 * 3 * 2 * 1\end{aligned}$$

Réversivité terminale – motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et
l'appel

(fact 4)

Cette dernière expression s'évalue au fil des appels récursifs en

$$\begin{aligned}(\text{fact } 4) &\rightarrow 4 * (\text{fact } 3) \rightarrow 4 * 3 * (\text{fact } 2) \\ &\rightarrow 4 * 3 * 2 * (\text{fact } 1) \rightarrow 4 * 3 * 2 * 1 \\ &\rightsquigarrow 24\end{aligned}$$

Réversivité terminale – motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1))
```

Considérons la fonction ci-contre et
l'appel

(fact 4)

Cette dernière expression s'**évalue** au fil des appels récursifs en

$$\begin{aligned}(\text{fact } 4) &\rightarrow 4 * (\text{fact } 3) \rightarrow 4 * 3 * (\text{fact } 2) \\ &\rightarrow 4 * 3 * 2 * (\text{fact } 1) \rightarrow 4 * 3 * 2 * 1 \\ &\rightsquigarrow 24\end{aligned}$$

Ce calcul, pour être mené à bien, a dû **garder en mémoire l'expression**

$$4 * 3 * 2 * 1$$

qui fait intervenir quatre (= n) opérandes et trois (= $n - 1$) opérateurs.

Récurivité terminale – motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc))
```

Considérons la fonction ci-contre et
l'appel

```
(fact 4 1)
```

Récurivité terminale – motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else
```

```
    (fact (n - 1) (n * acc))
```

Considérons la fonction ci-contre et
l'appel

```
(fact 4 1)
```

Cette dernière expression s'évalue au fil des appels récursifs en

Récurtivité terminale – motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else
```

```
    (fact (n - 1) (n * acc))
```

Considérons la fonction ci-contre et
l'appel

```
(fact 4 1)
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
(fact 4 1)
```

Récurtivité terminale – motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else
```

```
    (fact (n - 1) (n * acc))
```

Considérons la fonction ci-contre et
l'appel

```
(fact 4 1)
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
(fact 4 1) → (fact 3 (4 * 1))
```

Récurtivité terminale – motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else
```

```
    (fact (n - 1) (n * acc))
```

Considérons la fonction ci-contre et
l'appel

```
(fact 4 1)
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
(fact 4 1) → (fact 3 (4 * 1)) ↪ (fact 3 4)
```

Réversivité terminale – motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else
```

```
    (fact (n - 1) (n * acc))
```

Considérons la fonction ci-contre et
l'appel

```
(fact 4 1)
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
(fact 4 1)  →  (fact 3 (4 * 1))  ↪  (fact 3 4)  
             →  (fact 2 (3 * 4))
```

Récurivité terminale – motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc))
```

Considérons la fonction ci-contre et
l'appel

```
(fact 4 1)
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
(fact 4 1)  →  (fact 3 (4 * 1))  ⇝  (fact 3 4)  
            →  (fact 2 (3 * 4))  ⇝  (fact 2 12)
```

Réversivité terminale – motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc))
```

Considérons la fonction ci-contre et
l'appel

```
(fact 4 1)
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
(fact 4 1) → (fact 3 (4 * 1))  ↗ (fact 3 4)  
           → (fact 2 (3 * 4))  ↗ (fact 2 12)  
           → (fact 1 (2 * 12))
```

Récurivité terminale – motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else
```

```
    (fact (n - 1) (n * acc))
```

Considérons la fonction ci-contre et
l'appel

```
(fact 4 1)
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
(fact 4 1) → (fact 3 (4 * 1))  ↗ (fact 3 4)  
           → (fact 2 (3 * 4))  ↗ (fact 2 12)  
           → (fact 1 (2 * 12)) ↗ (fact 1 24)
```

Réversivité terminale – motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else
```

```
    (fact (n - 1) (n * acc))
```

Considérons la fonction ci-contre et
l'appel

```
(fact 4 1)
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
(fact 4 1)  →  (fact 3 (4 * 1))  ↗→  (fact 3 4)  
            →  (fact 2 (3 * 4))  ↗→  (fact 2 12)  
            →  (fact 1 (2 * 12))  ↗→  (fact 1 24)  
            →  24
```

Récurivité terminale – motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else
```

```
    (fact (n - 1) (n * acc))
```

Considérons la fonction ci-contre et
l'appel

```
(fact 4 1)
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
(fact 4 1)  →  (fact 3 (4 * 1))  ↗→  (fact 3 4)  
             →  (fact 2 (3 * 4))  ↗→  (fact 2 12)  
             →  (fact 1 (2 * 12)) ↗→  (fact 1 24)  
             →  24
```

Ce calcul, pour être mené à bien, a dû **garder en mémoire des expressions** faisant intervenir au plus deux opérandes et un opérateur, en plus de l'appel de fonction.

Réversivité terminale

La 2^e version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel réursif est renvoyé tel quel, sans l'adjoindre d'une opération.

Réversivité terminale

La 2^e version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1^{re} version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

Récurtivité terminale

La 2^e version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1^{re} version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

alors que dans la 2^e, elle ne subit aucune modification

```
(fact (n - 1) (n * acc)).
```

Réversivité terminale

La 2^e version de la fonction `fact` possède la propriété d'être **réversive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1^{re} version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

alors que dans la 2^e, elle ne subit aucune modification

```
(fact (n - 1) (n * acc)).
```

Le calcul de la 1^{re} version nécessite de garder en mémoire une expression de taille $\Theta(n)$, alors que celui de la 2^e ne travaille que sur une expression de taille $\Theta(1)$.

Réversivité terminale

La 2^e version de la fonction `fact` possède la propriété d'être **réversive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjindre d'une opération.

En effet, dans la 1^{re} version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

alors que dans la 2^e, elle ne subit aucune modification

```
(fact (n - 1) (n * acc)).
```

Le calcul de la 1^{re} version nécessite de garder en mémoire une expression de taille $\Theta(n)$, alors que celui de la 2^e ne travaille que sur une expression de taille $\Theta(1)$.

Les fonctions récursives terminales utilisent **moins de mémoire** que leurs analogues non récursives terminales. Elles sont donc à préférer.

Réversivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'accumulateur.

Réversivité terminale — accumulateurs

Pour écrire des fonctions réversives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur (en général cette valeur correspond au **cas terminal de la récursion**).

Réversivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur (en général cette valeur correspond au **cas terminal de la récursion**).

En général, il est d'usage d'**enrober** une fonction avec accumulateur pour la rendre plus facilement utilisable :

Réversivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur (en général cette valeur correspond au **cas terminal de la récursion**).

En général, il est d'usage d'**enrober** une fonction avec accumulateur pour la rendre plus facilement utilisable :

la fonction

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc))
```

Réversivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur (en général cette valeur correspond au **cas terminal de la récursion**).

En général, il est d'usage d'**enrober** une fonction avec accumulateur pour la rendre plus facilement utilisable :

la fonction

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc))
```

devient

```
let fact n =  
  let rec aux n acc =  
    if n <= 1 then  
      acc  
    else  
      (aux (n - 1) (n * acc))  
  in  
  (aux n 1)
```

Récurtivité terminale — Fibonacci

```
let rec fibo n =  
  if n <= 1 then  
    n  
  else  
    (fibo (n - 1))  
    + (fibo (n - 2))
```

C'est la version non récursive terminale de la fonction calculant le n^{e} nombre de Fibonacci.

Réversivité terminale — Fibonacci

```
let rec fibo n =  
  if n <= 1 then  
    n  
  else  
    (fibo (n - 1))  
    + (fibo (n - 2))
```

C'est la version non récursive terminale de la fonction calculant le n^e nombre de Fibonacci.

En effet, l'appel récursif (double) est adjoint d'une opération (somme).

Réversivité terminale — Fibonacci

```
let rec fibo n =  
  if n <= 1 then  
    n  
  else  
    (fibo (n - 1))  
    + (fibo (n - 2))
```

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      (aux (n - 1)  
        (acc1 + acc2) acc1)  
  in  
  (aux n 1 0)
```

C'est la version non réursive terminale de la fonction calculant le n^e nombre de Fibonacci.

En effet, l'appel récursif (double) est adjoint d'une opération (somme).

C'est la version réursive terminale de la fonction précédente.

Réversivité terminale — Fibonacci

```
let rec fibo n =  
  if n <= 1 then  
    n  
  else  
    (fibo (n - 1))  
    + (fibo (n - 2))
```

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      (aux (n - 1)  
        (acc1 + acc2) acc1)  
  in  
  (aux n 1 0)
```

C'est la version non récursive terminale de la fonction calculant le n^{e} nombre de Fibonacci.

En effet, l'appel récursif (double) est adjoint d'une opération (somme).

C'est la version récursive terminale de la fonction précédente.

Elle utilise deux accumulateurs (à cause du double appel récursif précédent). `acc1` contient la valeur du $n - 1^{\text{e}}$ nombre de Fibonacci et `acc2` contient la valeur du $n - 2^{\text{e}}$ nombre de Fibonacci.

Récurtivité terminale — forme générale

Une fonction récursive terminale a pour **forme générale**

Fonction *rec* $f(x_1, \dots, x_n, acc_1, \dots, acc_m)$:

Si C :

$f(\text{maje}(x_1, \dots, x_n), \text{maj}_s(acc_1, \dots, acc_m))$

Sinon :

R

Fin

Fin

où

- ▶ x_1, \dots, x_n sont les paramètres (entrées);
- ▶ acc_1, \dots, acc_m sont les accumulateurs (sorties);
- ▶ C est une expression booléenne dépendant des entrées et / ou des sorties;
- ▶ R est une expression résultat obtenue à partir des accumulateurs;
- ▶ *maje* indique la mise à jour des x_1, \dots, x_n lors de l'appel récursif;
- ▶ *maj_s* indique la mise à jour des acc_1, \dots, acc_m lors de l'appel récursif.

Réversivité terminale — dérécursivatiion

La **dérécursivatiion** est un procédé qui permet de transformer toute fonction récursive terminale en une fonction itérative.

Les notations sont ici les mêmes que celles utilisées précédemment.

Réversivité terminale — dérécursivatiion

La **dérécursivatiion** est un procédé qui permet de transformer toute fonction récursive terminale en une fonction itérative.

Voici une fonction récursive terminale dans sa forme générale et sa version dérécursivée :

```
Fonction rec  $f(x_1, \dots, x_n,$   
           $acc_1, \dots, acc_m) :$   
  Si  $C :$   
     $f(\text{maje}(x_1, \dots, x_n),$   
        $\text{maj}_s(acc_1, \dots, acc_m))$   
  Sinon :  
     $R$   
  Fin  
Fin
```

```
Fonction it  $g(x_1, \dots, x_n,$   
           $acc_1, \dots, acc_m) :$   
  Tant que  $C :$   
     $(x_1, \dots, x_n)$   
       $:= \text{maje}(x_1, \dots, x_n)$   
     $(acc_1, \dots, acc_m)$   
       $:= \text{maj}_s(acc_1, \dots, acc_m)$   
  Fin  
   $R$   
Fin
```

Les notations sont ici les mêmes que celles utilisées précédemment.

Récurtivité terminale — dérécursivation (exemple)

Fonction en forme habituelle :

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      (aux (n - 1)  
        (acc1 + acc2) acc1)  
  in  
  (aux n 1 0)
```

Récurtivité terminale — dérécursivation (exemple)

Fonction en forme habituelle :

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      (aux (n - 1)  
        (acc1 + acc2) acc1)  
  in  
  (aux n 1 0)
```

Fonction en forme générale :

```
let rec fibo n acc1 acc2 =  
  if n >= 2 then  
    (fibo (n - 1)  
     (acc1 + acc2) acc1)  
  else  
    if n = 0 then  
      acc2  
    else  
      acc1
```

Récurtivité terminale — dérécursivation (exemple)

Fonction en forme habituelle :

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      (aux (n - 1)  
        (acc1 + acc2) acc1)  
  in  
  (aux n 1 0)
```

Fonction en forme générale :

```
let rec fibo n acc1 acc2 =  
  if n >= 2 then  
    (fibo (n - 1)  
     (acc1 + acc2) acc1)  
  else  
    if n = 0 then  
      acc2  
    else  
      acc1
```

Version dérécursivée en C :

```
int fibo(int n, int acc1, int acc2) {  
  while (n >= 2) {  
    n = n - 1;  
    acc1 = acc1 + acc2;  
    acc2 = acc1 - acc2;  
  }  
  if (n == 0) return acc2;  
  else return acc1;  
}
```

Plan

Notions

Récurtivité

Filtrage

Fonctions d'ordre supérieur

Polymorphisme

Stratégies d'évaluation

Exemple introductif

Considérons le type

```
type point =  
  |Droite of int  
  |Plan of int * int  
  |Espace of int * int * int
```

Exemple introductif

Considérons le type

```
type point =  
  |Droite of int  
  |Plan of int * int  
  |Espace of int * int * int
```

On souhaite écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

Exemple introductif

Considérons le type

```
type point =  
  |Droite of int  
  |Plan of int * int  
  |Espace of int * int * int
```

On souhaite écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

La meilleure manière de faire consiste à utiliser un **filtrage de motifs** :

```
let oppose p =  
  match p with  
  |(Droite x) -> (Droite (-x))  
  |(Plan (x, y)) -> (Plan (-x, -y))  
  |(Espace (x, y, z)) -> (Espace (-x, -y, -z))
```

Exemple introductif

Considérons le type

```
type point =  
  |Droite of int  
  |Plan of int * int  
  |Espace of int * int * int
```

On souhaite écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

La meilleure manière de faire consiste à utiliser un **filtrage de motifs** :

```
let oppose p =  
  match p with  
  |(Droite x) -> (Droite (-x))  
  |(Plan (x, y)) -> (Plan (-x, -y))  
  |(Espace (x, y, z)) -> (Espace (-x, -y, -z))
```

```
# (oppose (Plan (3,1)));;  
- : point = Plan (-3, -1)
```

Exemple introductif

Considérons le type

```
type point =  
  |Droite of int  
  |Plan of int * int  
  |Espace of int * int * int
```

On souhaite écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

La meilleure manière de faire consiste à utiliser un **filtrage de motifs** :

```
let oppose p =  
  match p with  
  |(Droite x) -> (Droite (-x))  
  |(Plan (x, y)) -> (Plan (-x, -y))  
  |(Espace (x, y, z)) -> (Espace (-x, -y, -z))
```

```
# (oppose (Plan (3,1)));;  
- : point = Plan (-3, -1)
```

```
# (oppose (Espace(1, 0, -1)));;  
- : point = Espace (-1, 0, 1)
```

Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où EXP, EXP_1, \dots, EXP_n sont des expressions d'un même type et $MOTIF_1, \dots, MOTIF_n$ des motifs, met en place un **filtrage de motifs** sur EXP .

Syntaxe et évaluation

La construction syntaxique

```
match EXP with  
  |MOTIF1 -> EXP1  
  ...  
  |MOTIFn -> EXPn
```

où EXP , EXP_1 , ..., EXP_n sont des expressions d'un même type et $MOTIF_1$, ..., $MOTIF_n$ des motifs, met en place un **filtrage de motifs** sur EXP .

Chaque ligne $MOTIF_i \rightarrow EXP_i$ est une **clause**.

Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où EXP , EXP_1 , ..., EXP_n sont des expressions d'un même type et $MOTIF_1$, ..., $MOTIF_n$ des motifs, met en place un **filtrage de motifs** sur EXP .

Chaque ligne $MOTIF_i \rightarrow EXP_i$ est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où EXP , EXP_1 , ..., EXP_n sont des expressions d'un même type et $MOTIF_1$, ..., $MOTIF_n$ des motifs, met en place un **filtrage de motifs** sur EXP .

Chaque ligne $MOTIF_i \rightarrow EXP_i$ est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

1. EXP est évaluée;

Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où EXP , EXP_1 , ..., EXP_n sont des expressions d'un même type et $MOTIF_1$, ..., $MOTIF_n$ des motifs, met en place un **filtrage de motifs** sur EXP .

Chaque ligne $MOTIF_i \rightarrow EXP_i$ est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

1. EXP est évaluée;
2. on essaye de **filtrer** (faire correspondre) la valeur de EXP avec l'un des motifs, de haut en bas;

Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où EXP, EXP_1, \dots, EXP_n sont des expressions d'un même type et $MOTIF_1, \dots, MOTIF_n$ des motifs, met en place un **filtrage de motifs** sur EXP .

Chaque ligne $MOTIF_i \rightarrow EXP_i$ est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

1. EXP est évaluée;
2. on essaye de **filtrer** (faire correspondre) la valeur de EXP avec l'un des motifs, de haut en bas;
3. si un motif $MOTIF_i$ filtre la valeur de EXP , la valeur de toute l'expression est celle de EXP_i ;

Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où EXP , EXP_1 , ..., EXP_n sont des expressions d'un même type et $MOTIF_1$, ..., $MOTIF_n$ des motifs, met en place un **filtrage de motifs** sur EXP .

Chaque ligne $MOTIF_i \rightarrow EXP_i$ est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

1. EXP est évaluée;
2. on essaye de **filtrer** (faire correspondre) la valeur de EXP avec l'un des motifs, de haut en bas;
3. si un motif $MOTIF_i$ filtre la valeur de EXP , la valeur de toute l'expression est celle de EXP_i ;
4. si aucun motif ne filtre la valeur de EXP , une erreur est signalée (à l'exécution).

Principe du filtrage

Le filtrage de motifs peut se penser en 1^{re} approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

Principe du filtrage

Le filtrage de motifs peut se penser en 1^{re} approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

On l'utilise principalement :

1. lorsque le traitement à effectuer dépend d'avantage de la **structure d'une valeur** que de la valeur elle-même. P.ex.,

Principe du filtrage

Le filtrage de motifs peut se penser en 1^{re} approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

On l'utilise principalement :

1. lorsque le traitement à effectuer dépend d'avantage de la **structure d'une valeur** que de la valeur elle-même. P.ex.,

```
let dimension p =                                     |(Espace (x, y, z)) -> 3
  match p with
    |(Droite x) -> 1                                   renvoie le nombre de
    |(Plan (x, y)) -> 2                               coordonnées du point p.
```

Principe du filtrage

Le filtrage de motifs peut se penser en 1^{re} approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

On l'utilise principalement :

1. lorsque le traitement à effectuer dépend d'avantage de la **structure d'une valeur** que de la valeur elle-même. P.ex.,

```
let dimension p = | (Espace (x, y, z)) -> 3
                  match p with
                    |(Droite x) -> 1          renvoie le nombre de
                    |(Plan (x, y)) -> 2      coordonnées du point p.
```

2. lorsque l'on souhaite d'accéder à une **partie d'une valeur**, le filtrage permettant de **déconstruire**. P.ex.,

Principe du filtrage

Le filtrage de motifs peut se penser en 1^{re} approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

On l'utilise principalement :

1. lorsque le traitement à effectuer dépend d'avantage de la **structure d'une valeur** que de la valeur elle-même. P.ex.,

```
let dimension p =                                     |(Espace (x, y, z)) -> 3
  match p with
    |(Droite x) -> 1                                 renvoie le nombre de
    |(Plan (x, y)) -> 2                             coordonnées du point p.
```

2. lorsque l'on souhaite d'accéder à une **partie d'une valeur**, le filtrage permettant de **déconstruire**. P.ex.,

```
let projection_x p =                                  |(Espace (x, y, z)) -> x
  match p with
    |(Droite x) -> x                                 renvoie la première coordonnée
    |(Plan (x, y)) -> x                             du point p.
```

Exhaustivité du filtrage

L'interpréteur vérifie si le filtrage est **exhaustif**, c.-à-d. si toute expression du type attendu **peut être filtrée par au moins un motif**.

Exhaustivité du filtrage

L'interpréteur vérifie si le filtrage est **exhaustif**, c.-à-d. si toute expression du type attendu **peut être filtrée par au moins un motif**.

Si ça n'est pas le cas, un avertissement est signalé. P.ex.,

```
# let dimension p =  
  match p with  
    |(Droite x) -> 1  
    |(Espace (x, y, z)) -> 3;;  
Warning 8: this pattern-matching is not exhaustive.  
Here is an example of a value that is not matched:  
Plan (_, _)  
val dimension : point -> int = <fun>
```

Exhaustivité du filtrage

L'interpréteur vérifie si le filtrage est **exhaustif**, c.-à-d. si toute expression du type attendu **peut être filtrée par au moins un motif**.

Si ça n'est pas le cas, un avertissement est signalé. P.ex.,

```
# let dimension p =  
  match p with  
    |(Droite x) -> 1  
    |(Espace (x, y, z)) -> 3;;  
Warning 8: this pattern-matching is not exhaustive.  
Here is an example of a value that is not matched:  
Plan (_, _)  
val dimension : point -> int = <fun>
```

Le **joker** `_` est un motif universel : il filtre toute valeur. Son utilisation rend donc tous les filtrages exhaustifs. P.ex.,

Exhaustivité du filtrage

L'interpréteur vérifie si le filtrage est **exhaustif**, c.-à-d. si toute expression du type attendu **peut être filtrée par au moins un motif**.

Si ça n'est pas le cas, un avertissement est signalé. P.ex.,

```
# let dimension p =  
  match p with  
    |(Droite x) -> 1  
    |(Espace (x, y, z)) -> 3;;  
Warning 8: this pattern-matching is not exhaustive.  
Here is an example of a value that is not matched:  
Plan (_, _)  
val dimension : point -> int = <fun>
```

Le **joker** `_` est un motif universel : il filtre toute valeur. Son utilisation rend donc tous les filtrages exhaustifs. P.ex.,

```
# let entier_vers_chaine n =  
  match n with  
    |0 -> "zero"  
    |1 -> "un"  
    |2 -> "deux"  
    |_ -> "autre";;  
val entier_vers_chaine : int -> string = <fun>
```

Gardes

Il est possible de **raffiner le filtrage** en incluant des tests à des clauses

MOTIF -> EXP.

Gardes

Il est possible de **raffiner le filtrage** en incluant des tests à des clauses

MOTIF -> EXP.

On utilise pour cela la syntaxe

MOTIF when TEST -> EXP

où TEST est une expression de type `bool` appelée **garde**.

Gardes

Il est possible de **raffiner le filtrage** en incluant des tests à des clauses

MOTIF -> EXP.

On utilise pour cela la syntaxe

MOTIF when TEST -> EXP

où TEST est une expression de type `bool` appelée **garde**.

Pour que ce motif filtre une expression, il faut en plus que la valeur de TEST soit `true`.

Gardes

Il est possible de **raffiner le filtrage** en incluant des tests à des clauses

MOTIF -> EXP.

On utilise pour cela la syntaxe

MOTIF when TEST -> EXP

où TEST est une expression de type `bool` appelée **garde**.

Pour que ce motif filtre une expression, il faut en plus que la valeur de TEST soit `true`.

P.ex.,

```
let est_dans_quart_de_plan p =  
  match p with  
  | (Droite _) -> false  
  | (Plan (x, y)) when x >= 0 && y >= 0 -> true  
  | (Plan (_, _)) -> false  
  | (Espace (_, _, _)) -> false
```

teste si l'argument est un point du plan à coordonnées positives.

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (0, true, (), 'a', "abc", etc.);

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (0, true, (), 'a', "abc", etc.);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (0, true, (), 'a', "abc", etc.);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (0, true, (), 'a', "abc", etc.);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement x , y et/ou z à des valeurs.

```
let somme p =  
  match p with  
  |(Droite x) -> x  
  |(Plan (x, y)) -> x + y  
  |(Espace (x, y, z)) ->  
    x + y + z
```

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (0, true, (), 'a', "abc", etc.);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement x , y et/ou z à des valeurs.

```
let somme p =  
  match p with  
  |(Droite x) -> x  
  |(Plan (x, y)) -> x + y  
  |(Espace (x, y, z)) ->  
    x + y + z
```

Attention : ici
l'occurrence de n dans le
2^e motif ne fait pas
référence à la liaison
précédente.

```
let f x =  
  let n = 2 in  
  match x with  
  |0 -> 0  
  |n -> -1  
  |_ -> 1
```

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (0, true, (), 'a', "abc", etc.);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement x, y et/ou z à des valeurs.

```
let somme p =  
  match p with  
  |(Droite x) -> x  
  |(Plan (x, y)) -> x + y  
  |(Espace (x, y, z)) ->  
    x + y + z
```

Attention : ici l'occurrence de n dans le 2^e motif ne fait pas référence à la liaison précédente.

```
let f x = - : int = 0  
  let n = 2 in  
  match x with  
  |0 -> 0  
  |n -> -1  
  |_ -> 1  
  
# (f 0);;
```

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (0, true, (), 'a', "abc", etc.);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement x, y et/ou z à des valeurs.

```
let somme p =  
  match p with  
  |(Droite x) -> x  
  |(Plan (x, y)) -> x + y  
  |(Espace (x, y, z)) ->  
    x + y + z
```

Attention : ici l'occurrence de n dans le 2^e motif ne fait pas référence à la liaison précédente.

```
let f x =  
  let n = 2 in  
  match x with  
  |0 -> 0  
  |n -> -1  
  |_ -> 1  
  
# (f 0);;  
- : int = 0  
# (f 3);;  
- : int = -1
```

Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (0, true, (), 'a', "abc", etc.);
2. les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement x , y et/ou z à des valeurs.

```
let somme p =  
  match p with  
  |(Droite x) -> x  
  |(Plan (x, y)) -> x + y  
  |(Espace (x, y, z)) ->  
    x + y + z
```

Attention : ici
l'occurrence de n dans le
2^e motif ne fait pas
référence à la liaison
précédente.

```
let f x =  
  let n = 2 in  
  match x with  
  |0 -> 0  
  |n -> -1  
  |_ -> 1
```

```
# (f 0);;
```

```
- : int = 0  
# (f 3);;  
- : int = -1
```

Ainsi, le motif n filtre
toutes les valeurs.