

## Le type `unit`

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

`()`

## Le type `unit`

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

`()`

```
# ();;
```

```
- : unit = ()
```

## Le type `unit`

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

`()`

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

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

## Le type `unit`

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

`()`

```
# ();;
```

```
- : unit = ()
```

```
# let a = () in let b = a in  
    b;;
```

```
- : unit = ()
```

Les opérateurs relationnels sont bien définis sur `unit`.

## Le type `unit`

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

`()`

```
# ();;
```

```
- : unit = ()
```

```
# let a = () in let b = a in
```

```
    b;;
```

```
- : unit = ()
```

Les opérateurs relationnels sont bien définis sur `unit`.

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

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

# Plan

## Programmation

Interpréteur CAML

Liaisons

Types de base

**Fonctions**

# La notion de fonction

On rappelle qu'en programmation fonctionnelle,

1. l'objet de base est la **fonction**;

# La notion de fonction

On rappelle qu'en programmation fonctionnelle,

1. l'objet de base est la **fonction**;
2. un programme est une collection de définitions de fonctions;

# La notion de fonction

On rappelle qu'en programmation fonctionnelle,

1. l'objet de base est la **fonction**;
2. un programme est une collection de définitions de fonctions;
3. l'exécution d'un programme est l'application d'une fonction principale à des arguments.

# La notion de fonction

On rappelle qu'en programmation fonctionnelle,

1. l'objet de base est la **fonction**;
2. un programme est une collection de définitions de fonctions;
3. l'exécution d'un programme est l'application d'une fonction principale à des arguments.

La raison d'être d'un programme, en programmation fonctionnelle, est de **calculer une valeur**.

# La notion de fonction

On rappelle qu'en programmation fonctionnelle,

1. l'objet de base est la **fonction**;
2. un programme est une collection de définitions de fonctions;
3. l'exécution d'un programme est l'application d'une fonction principale à des arguments.

La raison d'être d'un programme, en programmation fonctionnelle, est de **calculer une valeur**.

Cette valeur calculée est précisément la **valeur de retour** de la fonction principale du programme.

# La notion de fonction

D'un point de vue formel, une **fonction**

$$f : T_1 \times T_2 \times \cdots \times T_n \rightarrow S$$

est une partie du produit cartésien  $T_1 \times T_2 \times \cdots \times T_n \times S$

## La notion de fonction

D'un point de vue formel, une fonction

$$f : T_1 \times T_2 \times \cdots \times T_n \rightarrow S$$

est une partie du produit cartésien  $T_1 \times T_2 \times \cdots \times T_n \times S$  telle que

$$(x_1, x_2, \dots, x_n, y) \in f \text{ et } (x_1, x_2, \dots, x_n, y') \in f \quad \text{implique} \quad y = y'.$$

# La notion de fonction

D'un point de vue formel, une **fonction**

$$f : T_1 \times T_2 \times \cdots \times T_n \rightarrow S$$

est une partie du produit cartésien  $T_1 \times T_2 \times \cdots \times T_n \times S$  telle que

$$(x_1, x_2, \dots, x_n, y) \in f \text{ et } (x_1, x_2, \dots, x_n, y') \in f \quad \text{implique} \quad y = y'.$$

D'usage, la propriété  $(x_1, x_2, \dots, x_n, y) \in f$  est notée

$$f(x_1, x_2, \dots, x_n) = y.$$

# La notion de fonction

D'un point de vue formel, une **fonction**

$$f : T_1 \times T_2 \times \cdots \times T_n \rightarrow S$$

est une partie du produit cartésien  $T_1 \times T_2 \times \cdots \times T_n \times S$  telle que

$$(x_1, x_2, \dots, x_n, y) \in f \text{ et } (x_1, x_2, \dots, x_n, y') \in f \text{ implique } y = y'.$$

D'usage, la propriété  $(x_1, x_2, \dots, x_n, y) \in f$  est notée

$$f(x_1, x_2, \dots, x_n) = y.$$

On appelle l'entier  $n$  l'arité de  $f$  (qui est son nombre d'entrées).

# La notion de fonction

D'un point de vue formel, une **fonction**

$$f : T_1 \times T_2 \times \cdots \times T_n \rightarrow S$$

est une partie du produit cartésien  $T_1 \times T_2 \times \cdots \times T_n \times S$  telle que

$$(x_1, x_2, \dots, x_n, y) \in f \text{ et } (x_1, x_2, \dots, x_n, y') \in f \text{ implique } y = y'.$$

D'usage, la propriété  $(x_1, x_2, \dots, x_n, y) \in f$  est notée

$$f(x_1, x_2, \dots, x_n) = y.$$

On appelle l'entier  $n$  l'arité de  $f$  (qui est son nombre d'entrées).

Distinction à faire entre

- ▶ **application** : fonction définie en tout point;

# La notion de fonction

D'un point de vue formel, une **fonction**

$$f : T_1 \times T_2 \times \cdots \times T_n \rightarrow S$$

est une partie du produit cartésien  $T_1 \times T_2 \times \cdots \times T_n \times S$  telle que

$$(x_1, x_2, \dots, x_n, y) \in f \text{ et } (x_1, x_2, \dots, x_n, y') \in f \text{ implique } y = y'.$$

D'usage, la propriété  $(x_1, x_2, \dots, x_n, y) \in f$  est notée

$$f(x_1, x_2, \dots, x_n) = y.$$

On appelle l'entier  $n$  l'arité de  $f$  (qui est son nombre d'entrées).

Distinction à faire entre

- ▶ **application** : fonction définie en tout point ;
- ▶ **fonction** : application partielle (pas de résultat pour certaines entrées).

# La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

# La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

On distingue la notion de paramètre et d'argument :

1. un **paramètre** est un **identificateur** non lié à une valeur et intervenant dans la définition d'une fonction;

# La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

On distingue la notion de paramètre et d'argument :

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

# La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

On distingue la notion de paramètre et d'argument :

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

P.ex., soit  $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  la fonction définie par  $f(x_1, x_2) := x_1 + x_2$ .

# La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

On distingue la notion de paramètre et d'argument :

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

P.ex., soit  $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  la fonction définie par  $f(x_1, x_2) := x_1 + x_2$ .

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

# La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

On distingue la notion de paramètre et d'argument :

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

P.ex., soit  $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  la fonction définie par  $f(x_1, x_2) := x_1 + x_2$ .

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

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

# La notion de fonction

En informatique, les ensembles d'entrée et de sortie des fonctions sont des **types**.

On distingue la notion de paramètre et d'argument :

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

P.ex., soit  $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  la fonction définie par  $f(x_1, x_2) := x_1 + x_2$ .

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

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

Lors d'un appel à une fonction  $f$  avec les arguments  $x_1, \dots, x_n$ , on dit que l'on **applique**  $f$  à  $x_1, \dots, x_n$ .

# Définition de fonctions

Pour définir une fonction, on utilise la construction syntaxique

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

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

# Définition de fonctions

Pour **définir une fonction**, on utilise la construction syntaxique

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

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

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

# Définition de fonctions

Pour **définir une fonction**, on utilise la construction syntaxique

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

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

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

P.ex.,

```
let oppose x = -x;;
```

produit l'évaluation

```
val oppose : int -> int = <fun>
```

# Définition de fonctions

Pour **définir une fonction**, on utilise la construction syntaxique

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

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

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

P.ex.,

```
let oppose x = -x;;
```

produit l'évaluation

```
val oppose : int -> int = <fun>
```

Explications :

- ▶ `val oppose` informe qu'on a lié au nom `oppose` une valeur;

# Définition de fonctions

Pour **définir une fonction**, on utilise la construction syntaxique

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

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

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

P.ex.,

```
let oppose x = -x;;
```

produit l'évaluation

```
val oppose : int -> int = <fun>
```

Explications :

- ▶ `val oppose` informe qu'on a lié au nom `oppose` une valeur;
- ▶ `: int -> int` informe que cette valeur est de type `int -> int`;

# Définition de fonctions

Pour **définir une fonction**, on utilise la construction syntaxique

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

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

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

P.ex.,

```
let oppose x = -x;;
```

produit l'évaluation

```
val oppose : int -> int = <fun>
```

Explications :

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

# Application de fonctions

Considérons la fonction

```
# let puissance_8 n =  
    let n2 = n * n in  
    let n4 = n2 * n2 in  
    n4 * n4;;  
val puissance_8 : int -> int = <fun>
```

# Application de fonctions

Considérons la fonction

```
# let puissance_8 n =  
    let n2 = n * n in  
    let n4 = n2 * n2 in  
    n4 * n4;;  
val puissance_8 : int -> int = <fun>
```

Pour **appliquer** `puissance_8` à l'argument 2, on écrit

```
# (puissance_8 2);;  
- : int = 256
```

On notera l'utilisation inhabituelle  
des parenthèses.

# Application de fonctions

Considérons la fonction

```
# let puissance_8 n =  
    let n2 = n * n in  
    let n4 = n2 * n2 in  
    n4 * n4;;  
val puissance_8 : int -> int = <fun>
```

Pour **appliquer** `puissance_8` à l'argument 2, on écrit

```
# (puissance_8 2);;  
- : int = 256
```

On notera l'utilisation inhabituelle  
des parenthèses.

Il est possible d'appeler une fonction dans une autre :

```
# let puissance_16 n =  
    let n8 = (puissance_8 n) in  
    n8 * n8;;  
val puissance_16 : int -> int = <fun>
```

## Fonctions locales

Une **fonction locale** est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

## Fonctions locales

Une **fonction locale** est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

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

# Fonctions locales

Une **fonction locale** est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

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

# (f "");;
- : string = "abab"

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

## Fonctions locales

Une **fonction locale** est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

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

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

## Fonctions locales

Une **fonction locale** est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

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

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

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

## Fonctions locales

Une **fonction locale** est une fonction définie à l'intérieure d'une autre. Sa **portée lexicale** s'étend à la fonction dans laquelle elle est définie.

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

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

```
# let f x =                                     # (f 1);;
  let g y =                                     - : bool = false
    y - 2 = x
  and h x =                                     # (f 2);;
    2 * x                                       - : bool = true
  in
  (g (h x));;
val f : int -> bool = <fun>
```

# Expressions conditionnelles

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

```
if COND then EXP1 else EXP2
```

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

# Expressions conditionnelles

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

```
if COND then EXP1 else EXP2
```

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

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

# Expressions conditionnelles

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

```
if COND then EXP1 else EXP2
```

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

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

Toute expression conditionnelle **possède une valeur** :

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

# Imbrications d'expressions conditionnelles

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

# Imbrications d'expressions conditionnelles

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

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

# Imbrications d'expressions conditionnelles

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

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

La (bonne) indentation aide à comprendre cette phrase.

# Imbrications d'expressions conditionnelles

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

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

La (bonne) indentation aide à comprendre cette phrase.

Néanmoins, l'interpréteur CAML la comprend sans ambiguïté.

# Imbrications d'expressions conditionnelles

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

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

La (bonne) indentation aide à comprendre cette phrase.

Néanmoins, l'interpréteur CAML la comprend sans ambiguïté.

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

# Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

## Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

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

# Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

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

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

# Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

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

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24  
→ if true then 21 else 24
```

## Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

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

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24  
→ if true then 21 else 24 → 21,
```

## Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

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

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

```
→ if true then 21 else 24 → 21,
```

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
```

## Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

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

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

```
→ if true then 21 else 24 → 21,
```

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
```

```
→ if if true then 1 = 0 else true then 28 else 21
```

## Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

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

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

```
→ if true then 21 else 24 → 21,
```

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
```

```
→ if if true then 1 = 0 else true then 28 else 21
```

```
→ if 1 = 0 then 28 else 21
```

## Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

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

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

```
→ if true then 21 else 24 → 21,
```

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
```

```
→ if if true then 1 = 0 else true then 28 else 21
```

```
→ if 1 = 0 then 28 else 21
```

```
→ if false then 28 else 21
```

## Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

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

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
```

```
→ if true then 21 else 24 → 21,
```

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
```

```
→ if if true then 1 = 0 else true then 28 else 21
```

```
→ if 1 = 0 then 28 else 21
```

```
→ if false then 28 else 21 → 21.
```

## Expressions conditionnelles, évaluations et valeurs

On rappelle qu'en programmation fonctionnelle, il n'y a pas d'effet de bord.

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

Quelques exemples d'**évaluations** d'expressions conditionnelles :

```
if 3 >= 1 then 21 else 24
→ if true then 21 else 24 → 21,
```

```
if if "ab" = "ab" then 1 = 0 else true then 28 else 21
→ if if true then 1 = 0 else true then 28 else 21
→ if 1 = 0 then 28 else 21
→ if false then 28 else 21 → 21.
```

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

# Demi-expressions conditionnelles

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

```
if COND then EXP
```

où **COND** est une expression dont la valeur est de type `bool` et **EXP** est une expression.

# Demi-expressions conditionnelles

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

```
if COND then EXP
```

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

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

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

# Demi-expressions conditionnelles

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

```
if COND then EXP
```

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

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

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

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

# Demi-expressions conditionnelles

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

```
if COND then EXP
```

où **COND** est une expression dont la valeur est de type `bool` et **EXP** est une expression.

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

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

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

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

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

est équivalent à

# Le type fonction

Considérons une fonction de la forme

```
let F X1 ... Xn = EXP
```

# Le type fonction

Considérons une fonction de la forme

```
let F X1 ... Xn = EXP
```

Lors de son évaluation, l'interpréteur va afficher

```
val F : E1 -> ... -> En -> S = <fun>
```

où  $E_1, \dots, E_n$  et  $S$  sont des types.

# Le type fonction

Considérons une fonction de la forme

```
let F X1 ... Xn = EXP
```

Lors de son évaluation, l'interpréteur va afficher

```
val F : E1 -> ... -> En -> S = <fun>
```

où  $E_1, \dots, E_n$  et  $S$  sont des types.

Plus précisément,

- ▶ pour tout  $1 \leq i \leq n$ ,  $E_i$  est le type attendu du  $i^{\text{e}}$  argument de  $F$ ;

# Le type fonction

Considérons une fonction de la forme

```
let F X1 ... Xn = EXP
```

Lors de son évaluation, l'interpréteur va afficher

```
val F : E1 -> ... -> En -> S = <fun>
```

où  $E_1, \dots, E_n$  et  $S$  sont des types.

Plus précisément,

- ▶ pour tout  $1 \leq i \leq n$ ,  $E_i$  est le type attendu du  $i^{\text{e}}$  argument de  $F$ ;
- ▶  $S$  est le type de retour de  $F$ .

# Le type fonction

Considérons une fonction de la forme

```
let F X1 ... Xn = EXP
```

Lors de son évaluation, l'interpréteur va afficher

```
val F : E1 -> ... -> En -> S = <fun>
```

où  $E_1, \dots, E_n$  et  $S$  sont des types.

Plus précisément,

- ▶ pour tout  $1 \leq i \leq n$ ,  $E_i$  est le type attendu du  $i^{\text{e}}$  argument de  $F$ ;
- ▶  $S$  est le type de retour de  $F$ .

$E_1 -> \dots -> E_n -> S$  est un type particulier, dit **type fonction**.

# Le type fonction et le constructeur flèche

Considérons la fonction

```
# let f x y =  
    (int_of_char y) + x;;  
val f : int -> char -> int = <fun>
```

La ligne `int -> char -> int` signifie que `f` est paramétrée par un `int` et un `char` et renvoie un `int`.

# Le type fonction et le constructeur flèche

Considérons la fonction

```
# let f x y =  
    (int_of_char y) + x;;  
val f : int -> char -> int = <fun>
```

La ligne `int -> char -> int` signifie que `f` est paramétrée par un `int` et un `char` et renvoie un `int`.

Le **constructeur de types** `->` (« flèche ») n'est pas associatif : toute expression le contenant nécessite des parenthèses.

# Le type fonction et le constructeur flèche

Considérons la fonction

```
# let f x y =  
    (int_of_char y) + x;;  
val f : int -> char -> int = <fun>
```

La ligne `int -> char -> int` signifie que `f` est paramétrée par un `int` et un `char` et renvoie un `int`.

Le **constructeur de types** `->` (« flèche ») n'est pas associatif : toute expression le contenant nécessite des parenthèses.

Cependant, lorsque l'on en met pas, la convention stipule que le parenthésage est fait **de droite à gauche**.

# Le type fonction et le constructeur flèche

Considérons la fonction

```
# let f x y =  
    (int_of_char y) + x;;  
val f : int -> char -> int = <fun>
```

La ligne `int -> char -> int` signifie que `f` est paramétrée par un `int` et un `char` et renvoie un `int`.

Le **constructeur de types** `->` (« flèche ») n'est pas associatif : toute expression le contenant nécessite des parenthèses.

Cependant, lorsque l'on en met pas, la convention stipule que le parenthésage est fait **de droite à gauche**.

Ainsi, le type `int -> char -> int` est équivalent au type dénoté par l'expression totalement parenthésée `(int -> (char -> int))`.

# Applications partielles de fonctions

Considérons une fonction  $f$  à deux paramètres de types  $E1$  et  $E2$  et à type de retour  $S$ .

# Applications partielles de fonctions

Considérons une fonction  $f$  à deux paramètres de types  $E1$  et  $E2$  et à type de retour  $S$ .

Deux faits évidents :

1. la fonction  $f$  est de type  $E1 \rightarrow E2 \rightarrow S$ ;

# Applications partielles de fonctions

Considérons une fonction  $f$  à deux paramètres de types  $E1$  et  $E2$  et à type de retour  $S$ .

Deux faits évidents :

1. la fonction  $f$  est de type  $E1 \rightarrow E2 \rightarrow S$ ;
2. si  $e1$  et  $e2$  sont des valeurs de types respectifs  $E1$  et  $E2$ , l'application de  $f$  à  $e1$  et  $e2$  par  $(f\ e1\ e2)$  renvoie une valeur de type  $S$ .

# Applications partielles de fonctions

Considérons une fonction  $f$  à deux paramètres de types  $E1$  et  $E2$  et à type de retour  $S$ .

Deux faits évidents :

1. la fonction  $f$  est de type  $E1 \rightarrow E2 \rightarrow S$ ;
2. si  $e1$  et  $e2$  sont des valeurs de types respectifs  $E1$  et  $E2$ , l'application de  $f$  à  $e1$  et  $e2$  par  $(f\ e1\ e2)$  renvoie une valeur de type  $S$ .

Il est cependant possible d'**appliquer**  $f$  à **seulement**  $e1$  par  $(f\ e1)$ . On obtient ainsi une valeur de type  $E2 \rightarrow S$  qui est donc une **fonction**.

# Applications partielles de fonctions

Considérons une fonction  $f$  à deux paramètres de types  $E1$  et  $E2$  et à type de retour  $S$ .

Deux faits évidents :

1. la fonction  $f$  est de type  $E1 \rightarrow E2 \rightarrow S$ ;
2. si  $e1$  et  $e2$  sont des valeurs de types respectifs  $E1$  et  $E2$ , l'application de  $f$  à  $e1$  et  $e2$  par  $(f\ e1\ e2)$  renvoie une valeur de type  $S$ .

Il est cependant possible d'**appliquer**  $f$  à **seulement**  $e1$  par  $(f\ e1)$ . On obtient ainsi une valeur de type  $E2 \rightarrow S$  qui est donc une **fonction**.

La fonction  $(f\ e1)$  est ainsi une fonction qui se comporte comme  $f$  lorsque son 1<sup>er</sup> paramètre est fixé à la valeur  $e1$ .

# Applications partielles de fonctions

Considérons une fonction  $f$  à deux paramètres de types  $E1$  et  $E2$  et à type de retour  $S$ .

Deux faits évidents :

1. la fonction  $f$  est de type  $E1 \rightarrow E2 \rightarrow S$ ;
2. si  $e1$  et  $e2$  sont des valeurs de types respectifs  $E1$  et  $E2$ , l'application de  $f$  à  $e1$  et  $e2$  par  $(f\ e1\ e2)$  renvoie une valeur de type  $S$ .

Il est cependant possible d'**appliquer**  $f$  à **seulement**  $e1$  par  $(f\ e1)$ . On obtient ainsi une valeur de type  $E2 \rightarrow S$  qui est donc une **fonction**.

La fonction  $(f\ e1)$  est ainsi une fonction qui se comporte comme  $f$  lorsque son 1<sup>er</sup> paramètre est fixé à la valeur  $e1$ .

On dit que  $(f\ e1)$  est une **application partielle** de  $f$  à des arguments.

# Applications partielles de fonctions

Il y a donc équivalence entre l'appel

`(f e1 e2)`

et l'appel

`((f e1) e2).`

# Applications partielles de fonctions

Il y a donc équivalence entre l'appel

$$(f \ e1 \ e2)$$

et l'appel

$$((f \ e1) \ e2).$$

Plus généralement, si  $f$  est une fonction de type

$$E1 \rightarrow \dots \rightarrow E_n \rightarrow S$$

et  $e_1, \dots, e_k$  sont des valeurs de types respectifs  $E_1, \dots, E_k$  avec  $k \leq n$ ,

# Applications partielles de fonctions

Il y a donc équivalence entre l'appel

$$(f \ e1 \ e2)$$

et l'appel

$$((f \ e1) \ e2).$$

Plus généralement, si  $f$  est une fonction de type

$$E1 \rightarrow \dots \rightarrow E_n \rightarrow S$$

et  $e_1, \dots, e_k$  sont des valeurs de types respectifs  $E_1, \dots, E_k$  avec  $k \leq n$ ,  
l'application partielle

$$(f \ e1 \ \dots \ e_k)$$

est une fonction de type

$$E_{k+1} \rightarrow \dots \rightarrow E_n \rightarrow S$$

# Applications partielles de fonctions — exemple

```
# let distr a b c =  
    a * b + a * c;;  
val distr : int -> int  
    -> int -> int = <fun>
```

Cette fonction représente  $f : \mathbb{Z}^3 \rightarrow \mathbb{Z}$   
vérifiant  $(a, b, c) \mapsto ab + ac$ .

# Applications partielles de fonctions — exemple

```
# let distr a b c =  
    a * b + a * c;;  
val distr : int -> int  
    -> int -> int = <fun>
```

Cette fonction représente  $f : \mathbb{Z}^3 \rightarrow \mathbb{Z}$   
vérifiant  $(a, b, c) \mapsto ab + ac$ .

```
# let distr_3 = (distr 3);;  
val distr_3 : int -> int  
    -> int = <fun>
```

Cette fonction représente  $f_3 : \mathbb{Z}^2 \rightarrow \mathbb{Z}$   
vérifiant  $(b, c) \mapsto 3b + 3c$ .

# Applications partielles de fonctions — exemple

```
# let distr a b c =  
    a * b + a * c;;  
val distr : int -> int  
    -> int -> int = <fun>
```

Cette fonction représente  $f : \mathbb{Z}^3 \rightarrow \mathbb{Z}$   
vérifiant  $(a, b, c) \mapsto ab + ac$ .

```
# let distr_3 = (distr 3);;  
val distr_3 : int -> int  
    -> int = <fun>
```

Cette fonction représente  $f_3 : \mathbb{Z}^2 \rightarrow \mathbb{Z}$   
vérifiant  $(b, c) \mapsto 3b + 3c$ .

```
# let distr_3_5 = (distr_3 5);;  
val distr_3_5 : int -> int = <fun>
```

Cette fonction représente  $f_{3,5} : \mathbb{Z} \rightarrow \mathbb{Z}$   
vérifiant  $c \mapsto 15 + 3c$ .

# Applications partielles de fonctions — exemple

```
# let distr a b c =  
    a * b + a * c;;  
val distr : int -> int  
    -> int -> int = <fun>
```

Cette fonction représente  $f : \mathbb{Z}^3 \rightarrow \mathbb{Z}$   
vérifiant  $(a, b, c) \mapsto ab + ac$ .

```
# let distr_3 = (distr 3);;  
val distr_3 : int -> int  
    -> int = <fun>
```

Cette fonction représente  $f_3 : \mathbb{Z}^2 \rightarrow \mathbb{Z}$   
vérifiant  $(b, c) \mapsto 3b + 3c$ .

```
# let distr_3_5 = (distr_3 5);;  
val distr_3_5 : int -> int = <fun>
```

Cette fonction représente  $f_{3,5} : \mathbb{Z} \rightarrow \mathbb{Z}$   
vérifiant  $c \mapsto 15 + 3c$ .

La fonction `distr_3_5` peut aussi être définie directement par l'une ou l'autre des deux manières suivantes :

```
let distr_3_5 c = (distr 3 5 c)
```

```
let distr_3_5 = (distr 3 5)
```

# Fonctions anonymes

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

# Fonctions anonymes

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

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

# Fonctions anonymes

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

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

La construction syntaxique

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

où  $P_1, \dots, P_n$  sont des paramètres et  $EXP$  est une expression permet de définir une fonction anonyme.

# Fonctions anonymes

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

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

La construction syntaxique

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

où  $P_1, \dots, P_n$  sont des paramètres et  $EXP$  est une expression permet de définir une fonction anonyme.

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

Ceci définit une fonction anonyme appliquée d'emblée aux arguments 4 et 3.

# Fonctions anonymes

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

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

La construction syntaxique

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

où  $P_1, \dots, P_n$  sont des paramètres et  $EXP$  est une expression permet de définir une fonction anonyme.

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

Ceci définit une fonction anonyme appliquée d'emblée aux arguments 4 et 3.

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

`(produit 10)` renvoie une fonction de type `int -> int` qui multiplie par 10 son argument.

# Deviner le type d'une fonction

Le **système de typage** de CAML agit statiquement et implicitement. Il se charge de **deviner le type** d'une fonction en analysant son code.

# Deviner le type d'une fonction

Le **système de typage** de CAML agit statiquement et implicitement. Il se charge de **deviner le type** d'une fonction en analysant son code.

Devinons les types des fonctions suivantes :

```
let mystere x y z =  
  if x && (y 1) then  
    z
```

# Deviner le type d'une fonction

Le **système de typage** de CAML agit statiquement et implicitement. Il se charge de **deviner le type** d'une fonction en analysant son code.

Devinons les types des fonctions suivantes :

```
let mystere x y z =  
  if x && (y 1) then  
    z
```

Cette fonction est de type

```
bool -> (int -> bool) -> unit -> unit.
```

# Deviner le type d'une fonction

Le **système de typage** de CAML agit statiquement et implicitement. Il se charge de **deviner le type** d'une fonction en analysant son code.

Devinons les types des fonctions suivantes :

```
let mystere x y z =  
  if x && (y 1) then  
    z
```

Cette fonction est de type

```
bool -> (int -> bool) -> unit -> unit.
```

```
let etrange x y =  
  "a" ^ ((y 1) ((x 'a') + 1)) ^ "b"
```

# Deviner le type d'une fonction

Le **système de typage** de CAML agit statiquement et implicitement. Il se charge de **deviner le type** d'une fonction en analysant son code.

Devinons les types des fonctions suivantes :

```
let mystere x y z =  
  if x && (y 1) then  
    z
```

Cette fonction est de type

```
bool -> (int -> bool) -> unit -> unit.
```

```
let etrange x y =  
  "a" ^ ((y 1) ((x 'a') + 1)) ^ "b"
```

Cette fonction est de type

```
(char -> int) -> (int -> int -> string) -> string.
```

# Deviner le type d'une fonction

Devinons les types des fonctions suivantes :

# Deviner le type d'une fonction

Devinons les types des fonctions suivantes :

```
let mystere x y z t =  
  1. +. (x ((y (not z) (t - 1)) ^ "ab") y)
```

# Deviner le type d'une fonction

Devinons les types des fonctions suivantes :

```
let mystere x y z t =  
  1. +. (x ((y (not z) (t - 1)) ^ "ab") y)
```

Cette fonction est de type

```
(string -> (bool -> int -> string) -> float) ->  
(bool -> int -> string) -> bool -> int -> float.
```

# Deviner le type d'une fonction

Devinons les types des fonctions suivantes :

```
let mystere x y z t =  
  1. +. (x ((y (not z) (t - 1)) ^ "ab") y)
```

Cette fonction est de type

```
(string -> (bool -> int -> string) -> float) ->  
(bool -> int -> string) -> bool -> int -> float.
```

```
let etrange y =  
  ((fun x -> x + 1) 2) + (y (string_of_int 3))
```

# Deviner le type d'une fonction

Devinons les types des fonctions suivantes :

```
let mystere x y z t =  
  1. +. (x ((y (not z) (t - 1)) ^ "ab") y)
```

Cette fonction est de type

```
(string -> (bool -> int -> string) -> float) ->  
(bool -> int -> string) -> bool -> int -> float.
```

```
let etrange y =  
  ((fun x -> x + 1) 2) + (y (string_of_int 3))
```

Cette fonction est de type

```
(string -> int) -> int.
```

# Plan

## Pratique

Entrées et sorties

Compilation

# Plan

Pratique

Entrées et sorties

Compilation

# Entrées et sorties

La boucle d'interaction suffit pour afficher les résultats d'un programme.

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

# Entrées et sorties

La boucle d'interaction suffit pour afficher les résultats d'un programme.

Il existe néanmoins 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 `()`.

# Entrées et sorties

La boucle d'interaction suffit pour afficher les résultats d'un programme.

Il existe néanmoins 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 que l'on **sort du paradigme de programmation fonctionnelle pure** car elles produisent un effet (affichage ou bien attente d'une action de l'utilisateur).

# Fonctions d'écriture

En principe, les fonctions d'écriture ne renvoient rien. Leur **effet** est leur seul intérêt.

# Fonctions d'écriture

En principe, les fonctions d'écriture ne renvoient rien. Leur **effet** est leur seul intérêt.

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

## Fonctions d'écriture

En principe, les fonctions d'écriture ne renvoient rien. Leur **effet** est leur seul 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
```

# Fonctions de lecture

En principe, les fonctions de lecture ne prennent rien en entrée. Ce qu'elles renvoient, c.-à-d. la **valeur saisie** par l'utilisateur, est leur seul intérêt.

## Fonctions de lecture

En principe, les fonctions de lecture ne prennent rien en entrée. Ce qu'elles renvoient, c.-à-d. la **valeur saisie** par l'utilisateur, est leur seul intérêt.

Cependant, une fonction qui ne possède pas de paramètre est une constante.

# Fonctions de lecture

En principe, les fonctions de lecture ne prennent rien en entrée. Ce qu'elles renvoient, c.-à-d. la **valeur saisie** par l'utilisateur, est leur seul intérêt.

Cependant, une fonction qui ne possède pas de paramètre est une constante.

Ceci n'est clairement pas la nature d'une fonction de lecture puisque la valeur qu'elle renvoie varie selon l'état de ce qu'elle lit.

## Fonctions de lecture

En principe, les fonctions de lecture ne prennent rien en entrée. Ce qu'elles renvoient, c.-à-d. la **valeur saisie** par l'utilisateur, est leur seul intérêt.

Cependant, une fonction qui ne possède pas de paramètre est une constante.

Ceci n'est clairement pas la nature d'une fonction de lecture puisque la valeur qu'elle renvoie varie selon l'état de ce qu'elle lit.

L'astuce consiste à avoir des fonctions paramétrées par une valeur de type `unit`. Les fonctions de lecture s'appellent donc avec l'argument `()`.

## Fonctions de lecture

En principe, les fonctions de lecture ne prennent rien en entrée. Ce qu'elles renvoient, c.-à-d. la **valeur saisie** par l'utilisateur, est leur seul intérêt.

Cependant, une fonction qui ne possède pas de paramètre est une constante.

Ceci n'est clairement pas la nature d'une fonction de lecture puisque la valeur qu'elle renvoie varie selon l'état de ce qu'elle lit.

L'astuce consiste à avoir des fonctions paramétrées par une valeur de type `unit`. Les fonctions de lecture s'appellent donc avec l'argument `()`.

Les fonctions de lecture principales sont

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

# Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

# Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre.

# Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

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 de EXP1; EXP2 est celle de EXP2.

# Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

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 :

$$\text{EXP1}; \text{EXP2}$$

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

Le parenthésage d'une expression

$$E1; E2; \dots ; E_n$$

est fait implicitement de gauche à droite en

$$((\dots(E1; E2); \dots); E_n)$$

## Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

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 :

$$\text{EXP1}; \text{EXP2}$$

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

Le parenthésage d'une expression

$$E1; E2; \dots ; E_n$$

est fait implicitement de gauche à droite en

$$(((\dots(E1; E2); \dots); E_n)$$

La valeur de `E1; E2; ... ; En` est ainsi celle de `En`.

## Séquences

L'utilisation des fonctions d'entrée / sortie est adaptée à la programmation impérative.

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 :

$$\text{EXP1}; \text{EXP2}$$

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

Le parenthésage d'une expression

$$E1; E2; \dots ; E_n$$

est fait implicitement de gauche à droite en

$$(((\dots(E1; E2); \dots); E_n)$$

La valeur de `E1; E2; ... ; En` est ainsi celle de `En`.

De ce fait, `E1, ..., En-1` doivent être de type `unit`.

## Séquences — exemples

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

## Séquences — exemples

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

# 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>
```

# Séquences et blocs

Considérons la fonction

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

Error: Syntax error

# Séquences et blocs

Considérons la fonction

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

Error: Syntax error

L'opérateur de séquence est moins prioritaire que la conditionnelle. Ainsi, cette fonction est comprise en

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

Ceci explique l'**erreur de syntaxe**.

# Séquences et blocs

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

```
begin EXP end
```

où EXP est une expression.

# Séquences et blocs

Pour résoudre ce problème, on utilise 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 de `EXP`.

## Séquences et blocs

Pour résoudre ce problème, on utilise 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 de `EXP`.

La fonction précédente devient ainsi correcte en écrivant

```
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;  
end;
```