Caractéristiques des principaux langages

Langage	V. t. dyn.	V. t. stat.	T. expl.	T. impl.	Imp.	Fonc.
C	Non	Oui	Oui	Non	Oui	Non
Python	Oui	Non	Non	Oui	Oui	Impur
Caml	**Non**	**Oui**	**Non**	**Oui**	**Non**	**Impur**
Haskell	Non	Oui	Non	Oui	Non	Pur

Axe 2 : concepts premiers

Programmation

Pratique

Types

Plan

La 1re chose à apprendre

« La 1re chose à apprendre lorsque l'on aborde un nouveau langage de programmation est

La 1re chose à apprendre

« La 1re chose à apprendre lorsque l'on aborde un nouveau langage de programmation est de savoir écrire des commentaires ».

— J.-P. Duval

La 1re chose à apprendre

« La 1re chose à apprendre lorsque l'on aborde un nouveau langage de programmation est de savoir écrire des commentaires ».

— J.-P. Duval

En CAML, un commentaire est constitué de tout ce qui est délimité par `(*` et `*)`. Par exemple, `(* Ceci est un commentaire. *)`.

La 1re chose à apprendre

« La 1re chose à apprendre lorsque l'on aborde un nouveau langage de programmation est de savoir écrire des commentaires ».

— J.-P. Duval

En CAML, un commentaire est constitué de tout ce qui est délimité par `(*` et `*)`. Par exemple, `(* Ceci est un commentaire. *)`.

Les symboles `(*` et `*)` **fonctionnent comme des parenthèses** : il est possible d'appareiller chaque `(*` avec un unique `*)`. Il est ainsi possible d'imbriquer les commentaires.

La 1re chose à apprendre

« La 1re chose à apprendre lorsque l'on aborde un nouveau langage de programmation est de savoir écrire des commentaires ».

— J.-P. Duval

En CAML, un commentaire est constitué de tout ce qui est délimité par `(*` et `*)`. Par exemple, `(* Ceci est un commentaire. *)`.

Les symboles `(*` et `*)` **fonctionnent comme des parenthèses** : il est possible d'appareiller chaque `(*` avec un unique `*)`. Il est ainsi possible d'imbriquer les commentaires.

```
(* Ceci est un commentaire
(* imbrique. *) *)
```

En CAML, ceci fonctionne.

La 1re chose à apprendre

« La 1re chose à apprendre lorsque l'on aborde un nouveau langage de programmation est de savoir écrire des commentaires ».

— J.-P. Duval

En CAML, un commentaire est constitué de tout ce qui est délimité par `(*` et `*)`. Par exemple, `(* Ceci est un commentaire. *)`.

Les symboles `(*` et `*)` **fonctionnent comme des parenthèses** : il est possible d'appareiller chaque `(*` avec un unique `*)`. Il est ainsi possible d'imbriquer les commentaires.

```
(* Ceci est un commentaire
(* imbrique. *) *)
```

En CAML, ceci fonctionne.

```
/* Ceci est un commentaire
/* imbrique. */ */
```

En C, ceci ne fonctionne pas.

Plan

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en Caml :

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode** ;

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode** ;
2. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur natif** ;

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode** ;
2. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur natif** ;
3. ouvrir l'**interpréteur** CAML et écrire du code, ou bien en importer.

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode** ;
2. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur natif** ;
3. ouvrir l'**interpréteur** CAML et écrire du code, ou bien en importer.

Les deux premières méthodes fournissent un exécutable (voir plus loin).

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode** ;
2. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur natif** ;
3. ouvrir l'**interpréteur** CAML et écrire du code, ou bien en importer.

Les deux premières méthodes fournissent un exécutable (voir plus loin).

La 3e est intéressante car elle permet de faire du développement incrémental, c.-à-d. l'écriture et le test pas à pas des fonctions nécessaires à la résolution d'un problème.

Dans ce cas, le programme n'est pas exécuté mais est interprété.

Interpréteur

L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.04.0

#
```

Interpréteur

L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.04.0

#
```

Celle-ci lance une boucle d'interaction qui se comporte, de manière répétée, de la façon suivante :

Interpréteur

L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.04.0

#
```

Celle-ci lance une boucle d'interaction qui se comporte, de manière répétée, de la façon suivante :

1. le programmeur **écrit** une phrase ;

Lire

Interpréteur

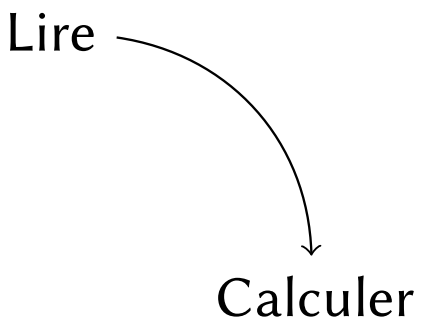
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.04.0

#
```

Celle-ci lance une boucle d'interaction qui se comporte, de manière répétée, de la façon suivante :

1. le programmeur **écrit** une phrase ;
2. le système l'**interprète** ;



Interpréteur

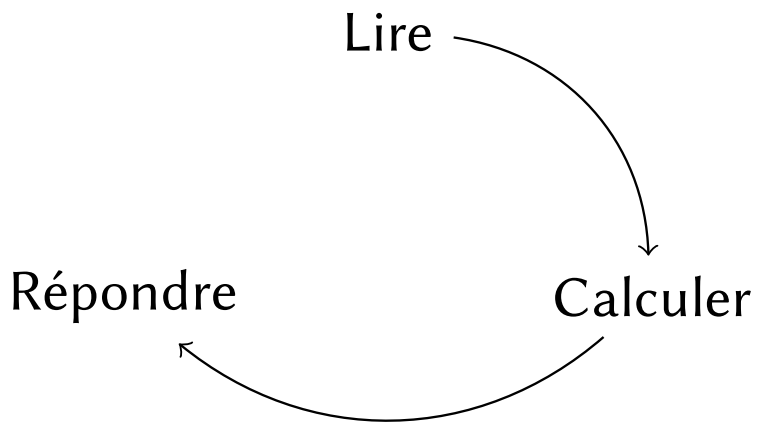
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.04.0

#
```

Celle-ci lance une boucle d'interaction qui se comporte, de manière répétée, de la façon suivante :

1. le programmeur **écrit** une phrase ;
2. le système l'**interprète** ;
3. le système affiche le **résultat** de la phrase.



Interpréteur

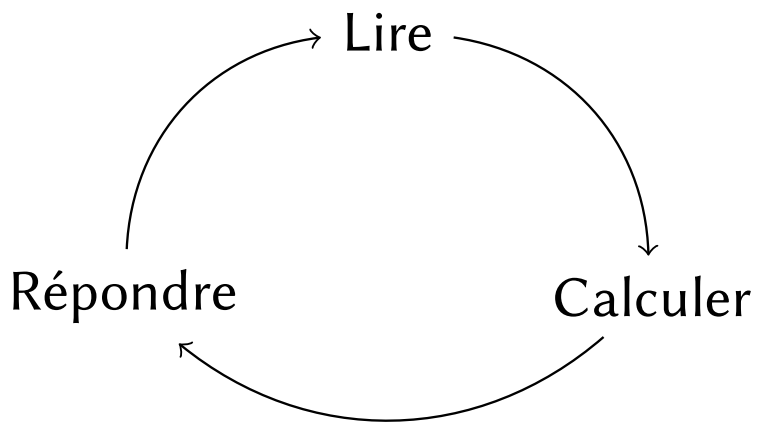
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.04.0

#
```

Celle-ci lance une boucle d'interaction qui se comporte, de manière répétée, de la façon suivante :

1. le programmeur **écrit** une phrase ;
2. le système l'**interprète** ;
3. le système affiche le **résultat** de la phrase.



Et ainsi de suite.

Phrases et réponses

Une phrase est une **expression** terminée par `;;` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Phrases et réponses

Une phrase est une **expression** terminée par `;;` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1
  + 1;;
```

Phrases et réponses

Une phrase est une **expression** terminée par `;;` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1
  + 1;;
```

L'utilisateur demande l'**évaluation** d'une phrase en appuyant sur **entrée** et le système fournit ensuite sa réponse.

```
# 1 + 1;;
- : int = 2
```

Phrases et réponses

Une phrase est une **expression** terminée par `;;` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1
  + 1;;
```

L'utilisateur demande l'**évaluation** d'une phrase en appuyant sur **entrée** et le système fournit ensuite sa réponse.

```
# 1 + 1;;
- : int = 2
```

Explications :

- le signe `-` signifie qu'une **valeur** a été calculée ;

Phrases et réponses

Une phrase est une **expression** terminée par `;;` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1
  + 1;;
```

L'utilisateur demande l'**évaluation** d'une phrase en appuyant sur **entrée** et le système fournit ensuite sa réponse.

```
# 1 + 1;;
- : int = 2
```

Explications :

- le signe `-` signifie qu'une **valeur** a été calculée ;
- `: int` signifie que cette valeur est de **type** `int` ;

Phrases et réponses

Une phrase est une **expression** terminée par `;;` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1
  + 1;;
```

L'utilisateur demande l'**évaluation** d'une phrase en appuyant sur **entrée** et le système fournit ensuite sa réponse.

```
# 1 + 1;;
- : int = 2
```

Explications :

- le signe `-` signifie qu'une **valeur** a été calculée ;
- `: int` signifie que cette valeur est de **type** `int` ;
- `= 2` signifie que cette valeur **est** `2`.

Plan

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

$$\text{« } \textit{Soit } n \textit{ l'entier } 5. \text{ »,}$$

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

$$\text{« } \textit{Soit } n \textit{ l'entier } 5. \text{ »,}$$

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Ceci s'appelle une définition, ou encore la liaison d'un nom à une valeur.

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

$$\text{« } \textit{Soit } n \textit{ l'entier } 5. \text{ »,}$$

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Ceci s'appelle une définition, ou encore la liaison d'un nom à une valeur.

On utilise pour cela la construction syntaxique

```
let ID = VAL
```

où `ID` est un nom (identificateur) et `VAL` est une expression.

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

$$\text{« } \textit{Soit } n \textit{ l'entier } 5. \text{ »,}$$

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Ceci s'appelle une définition, ou encore la liaison d'un nom à une valeur.

On utilise pour cela la construction syntaxique

```
let ID = VAL
```

où `ID` est un nom (identificateur) et `VAL` est une expression.

```
# let n = 5;;
val n : int = 5
```

La 1re phrase lie au nom `n` la valeur `5`. L'interpréteur **le signale** en commençant sa réponse par `val n`.

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

« *Soit* n *l'entier* 5. »,

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Ceci s'appelle une définition, ou encore la liaison d'un nom à une valeur.

On utilise pour cela la construction syntaxique

```
let ID = VAL
```

où `ID` est un nom (identificateur) et `VAL` est une expression.

```
# let n = 5;;
val n : int = 5
```

La 1re phrase lie au nom `n` la valeur `5`. L'interpréteur **le signale** en commençant sa réponse par `val n`.

```
# n;;
- : int = 5
```

La 2e phrase donne la valeur à laquelle `n` est liée.

Liaisons locales

Il est possible de définir des noms localement à une expression.

Liaisons locales

Il est possible de définir des noms localement à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

Liaisons locales

Il est possible de définir des noms localement à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où `ID` est un nom, `VAL` est une expression et `EXP` est une expression.

Liaisons locales

Il est possible de définir des noms localement à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où `ID` est un nom, `VAL` est une expression et `EXP` est une expression.

Très important : cette expression possède comme valeur celle de `EXP` dans laquelle les occurrences libres de `ID` sont remplacées par `VAL`.

Liaisons locales

Il est possible de définir des noms localement à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où `ID` est un nom, `VAL` est une expression et `EXP` est une expression.

Très important : cette expression possède comme valeur celle de `EXP` dans laquelle les occurrences libres de `ID` sont remplacées par `VAL`.

```
# let n = 5 in n + 1;;
- : int = 6
```

La 2e occurrence de `n` a pour valeur `5` à cause de la liaison locale précédente. Ainsi, `n + 1` a pour valeur `6`.

Liaisons locales

Il est possible de définir des noms localement à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où `ID` est un nom, `VAL` est une expression et `EXP` est une expression.

Très important : cette expression possède comme valeur celle de `EXP` dans laquelle les occurrences libres de `ID` sont remplacées par `VAL`.

```
# let n = 5 in n + 1;;
- : int = 6
```

La 2e occurrence de `n` a pour valeur `5` à cause de la liaison locale précédente. Ainsi, `n + 1` a pour valeur `6`.

```
# let n = 3;;
val n : int = 3
# let n = 4 in 2 * n;;
- : int = 8
# n;;
- : int = 3
```

La liaison de `n` à `4` dans la 2e phrase est locale : le nom global `n` défini en 1re phrase reste inchangé.

Liaisons locales

Il est possible de définir des noms localement à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où `ID` est un nom, `VAL` est une expression et `EXP` est une expression.

Très important : cette expression possède comme valeur celle de `EXP` dans laquelle les occurrences libres de `ID` sont remplacées par `VAL`.

```
# let n = 5 in n + 1;;
- : int = 6
```

La 2e occurrence de `n` a pour valeur `5` à cause de la liaison locale précédente. Ainsi, `n + 1` a pour valeur `6`.

```
# let n = 3;;
val n : int = 3
# let n = 4 in 2 * n;;
- : int = 8
# n;;
- : int = 3
```

La liaison de `n` à `4` dans la 2e phrase est locale : le nom global `n` défini en 1re phrase reste inchangé.

Ceci renseigne sur la valeur du nom `n` de la 3e phrase.

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

```
# let s = 2 in
      let s = s * s in
          let s = s * s in
              s + 4;;
```

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

```
# let s = 2 in
      let s = s * s in
          let s = s * s in
              s + 4;;
- : int = 20
```

Chaque occurrence du nom `s` fait référence à la valeur du nom `s` de la liaison précédente. On retrouve ainsi le résultat affiché.

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

```
# let s = 2 in
      let s = s * s in
          let s = s * s in
              s + 4;;
- : int = 20

# let s = let x = 3 in
      x * x;;
```

Chaque occurrence du nom `s` fait référence à la valeur du nom `s` de la liaison précédente. On retrouve ainsi le résultat affiché.

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

```
# let s = 2 in
      let s = s * s in
          let s = s * s in
              s + 4;;
- : int = 20
```

Chaque occurrence du nom `s` fait référence à la valeur du nom `s` de la liaison précédente. On retrouve ainsi le résultat affiché.

```
# let s = let x = 3 in
      x * x;;
val s : int = 9
```

Le nom `s` est lié à la valeur `9`.

En effet, la sous-expression `let x = 3 in x * x` a pour valeur `9`.

Liaisons locales — exemples

```
# let x =
      let y = 2 in
          let z = 3 in
              y + let z = 8 in
                  z * z;;
```

Liaisons locales — exemples

```
# let x =
      let y = 2 in
          let z = 3 in
              y + let z = 8 in
                  z * z;;
Warning 26: unused variable z.
val x : int = 66
```

La nom `z` défini en l. 3 n'est pas utilisé pour attribuer une valeur au nom `x`.

L'interpréteur le signale par un message d'avertissement.

Liaisons locales — exemples

```
# let x =
      let y = 2 in
          let z = 3 in
              y + let z = 8 in
                  z * z;;
Warning 26: unused variable z.
val x : int = 66

# let x = let y = 2 in
      x + y;;
```

La nom `z` défini en l. 3 n'est pas utilisé pour attribuer une valeur au nom `x`.

L'interpréteur le signale par un message d'avertissement.

Liaisons locales — exemples

```
# let x =
      let y = 2 in
          let z = 3 in
              y + let z = 8 in
                  z * z;;
Warning 26: unused variable z.
val x : int = 66
```

La nom `z` défini en l. 3 n'est pas utilisé pour attribuer une valeur au nom `x`.

L'interpréteur le signale par un message d'avertissement.

```
# let x = let y = 2 in
      x + y;;
Error: Unbound value x
```

L'évaluation de cette phrase produit une erreur lors de son interprétation. En effet, le nom `x` de la l. 2 n'est pas défini.

Liaisons (locales) simultanées

Il est possible de définir des noms ou de réaliser des liaisons locales simultanément dans une même phrase.

Liaisons (locales) simultanées

Il est possible de définir des noms ou de réaliser des liaisons locales simultanément dans une même phrase.

On utilise pour cela les constructions syntaxiques

```
let ID1 = VAL1 and ID2 = VAL2
```

ou

```
let ID1 = VAL1 and ID2 = VAL2 in EXP
```

où `ID1` et `ID2` sont des identificateurs, et `VAL1`, `VAL2` et `EXP` sont des expressions.

Liaisons (locales) simultanées

Il est possible de définir des noms ou de réaliser des liaisons locales simultanément dans une même phrase.

On utilise pour cela les constructions syntaxiques

```
let ID1 = VAL1 and ID2 = VAL2
```

ou

```
let ID1 = VAL1 and ID2 = VAL2 in EXP
```

où `ID1` et `ID2` sont des identificateurs, et `VAL1`, `VAL2` et `EXP` sont des expressions.

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

Cette phrase lie simultanément au nom `x` la valeur `1` et au nom `y` la valeur `2`.

Liaisons (locales) simultanées

Il est possible de définir des noms ou de réaliser des liaisons locales simultanément dans une même phrase.

On utilise pour cela les constructions syntaxiques

```
let ID1 = VAL1 and ID2 = VAL2
```

ou

```
let ID1 = VAL1 and ID2 = VAL2 in EXP
```

où `ID1` et `ID2` sont des identificateurs, et `VAL1`, `VAL2` et `EXP` sont des expressions.

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

Cette phrase lie simultanément au nom `x` la valeur `1` et au nom `y` la valeur `2`.

```
# let x = 1 in
      let y = 3
      and z = 4 in
          x + y + z;;
- : int = 8
```

Il est possible d'imbriquer les définitions locales et simultanées.

On notera l'indentation différente impliquée par les `in` et les `and`.

Liaisons simultanées — exemples

```
# let x = 1 in
      let x = 2
      and y = x + 3 in
          x + y;;
```

Liaisons simultanées — exemples

```
# let x = 1 in
      let x = 2
      and y = x + 3 in
          x + y;;
- : int = 6
```

Dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 1.

Liaisons simultanées — exemples

```
# let x = 1 in
      let x = 2
      and y = x + 3 in
          x + y;;
- : int = 6

# let x = 1 in
      let x = 2 in
          let y = x + 3 in
              x + y;;
```

Dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 1.

Liaisons simultanées — exemples

```
# let x = 1 in
      let x = 2
      and y = x + 3 in
          x + y;;
- : int = 6
```

Dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 1.

```
# let x = 1 in
      let x = 2 in
          let y = x + 3 in
              x + y;;
Warning 26: unused variable x.
- : int = 7
```

Cette phrase est obtenue en remplaçant le `and` par un `in let`.

Liaisons simultanées — exemples

```
# let x = 1 in
      let x = 2
      and y = x + 3 in
          x + y;;
- : int = 6
```

Dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 1.

```
# let x = 1 in
      let x = 2 in
          let y = x + 3 in
              x + y;;
Warning 26: unused variable x.
- : int = 7
```

Cette phrase est obtenue en remplaçant le `and` par un `in let`.

Le résultat est différent du précédent : dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 2.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
          x * x * x;;
```

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
      x * x * x;;
val x : int = 1
val y : int = 2
val z : int = 4096
```

Il y a trois liaisons simultanées.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
          x * x * x;;
val x : int = 1
val y : int = 2
val z : int = 4096
```

Il y a trois liaisons simultanées.

L'occurrence du nom `x` en l. 4 est celle définie en l. 3. Cette définition n'influe pas sur la définition de `x` en l. 1.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
          x * x * x;;
val x : int = 1
val y : int = 2
val z : int = 4096

# let x = 1
  and y = x + 1;;
```

Il y a trois liaisons simultanées.

L'occurrence du nom `x` en l. 4 est celle définie en l. 3. Cette définition n'influe pas sur la définition de `x` en l. 1.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
      x * x * x;;
val x : int = 1
val y : int = 2
val z : int = 4096
```

Il y a trois liaisons simultanées.

L'occurrence du nom `x` en l. 4 est celle définie en l. 3. Cette définition n'influe pas sur la définition de `x` en l. 1.

```
# let x = 1
  and y = x + 1;;
Error: Unbound value x
```

L'évaluation de cette phrase produit une erreur lors de son interprétation. En effet, le nom `x` de la l. 2 n'est pas défini.

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

En effet,

- une liaison `let ID = VAL` permet de former une phrase qui n'a pas de valeur et dont le but est de créer un alias `ID` pour la valeur de `VAL` dans la suite du programme.

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

En effet,

- une liaison `let ID = VAL` permet de former une phrase qui n'a pas de valeur et dont le but est de créer un alias `ID` pour la valeur de `VAL` dans la suite du programme. P.ex.,

```
# let x = 3;;
val x : int = 3
...
# x + 1;;
- : int = 4
```

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

En effet,

- une liaison `let ID = VAL` permet de former une phrase qui n'a pas de valeur et dont le but est de créer un alias `ID` pour la valeur de `VAL` dans la suite du programme. P.ex.,

```
# let x = 3;;
val x : int = 3
...
# x + 1;;
- : int = 4
```

- Une liaison locale `let ID = VAL in EXP` permet de former une phrase qui a une valeur et qui est celle de `EXP` dans laquelle les occurrences de `ID` sont remplacées par la valeur de `VAL`.

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

En effet,

- une liaison `let ID = VAL` permet de former une phrase qui n'a pas de valeur et dont le but est de créer un alias `ID` pour la valeur de `VAL` dans la suite du programme. P.ex.,

```
# let x = 3;;
val x : int = 3
...
# x + 1;;
- : int = 4
```

- Une liaison locale `let ID = VAL in EXP` permet de former une phrase qui a une valeur et qui est celle de `EXP` dans laquelle les occurrences de `ID` sont remplacées par la valeur de `VAL`. P.ex.,

```
# let x = 3 in x + 1;;
- : int = 4
```

Plan

Les six types de base

Nom du type	Utilisation
`int`	Représentation des **entiers signés**
`float`	Représentation des **nombres à virgule signés**
`char`	Représentation des **caractères**
`string`	Représentation des **chaînes de caractères**
`bool`	Représentation des **booléens**
`unit`	Type contenant une **unique valeur**

Le type `bool`

Le type `bool` contient exactement deux valeurs : `true` et `false`.

Le type `bool`

Le type `bool` contient exactement deux valeurs : `true` et `false`.

Opérateur	Arité	Rôle
`not`	1	Non logique
`&&`	2	Et logique
`||`	2	Ou logique

Ces opérateurs produisent des valeurs de type `bool`.

Le type `bool`

Le type `bool` contient exactement deux valeurs : `true` et `false`.

Opérateur	Arité	Rôle
`not`	1	Non logique
`&&`	2	Et logique
`||`	2	Ou logique

Ces opérateurs produisent des valeurs de type `bool`.

Par exemple :

```
# (false || (not false)) && (not (true || false));;
- : bool = false

# not true && false;;
- : bool = false
```

Le type `bool`

Le type `bool` contient exactement deux valeurs : `true` et `false`.

Opérateur	Arité	Rôle
`not`	1	Non logique
`&&`	2	Et logique
`||`	2	Ou logique

Ces opérateurs produisent des valeurs de type `bool`.

Par exemple :

```
# (false || (not false)) && (not (true || false));;
- : bool = false

# not true && false;;
- : bool = false
```

Règle : ne jamais hésiter à introduire des parenthèses (sans exagérer) pour gagner en lisibilité.

Le type `int`

Une valeur de type `int` peut s'écrire en

- décimal, sans préfixe (p.ex. `0`, `1024`, `-82`);

Le type `int`

Une valeur de type `int` peut s'écrire en

- décimal, sans préfixe (p.ex. `0`, `1024`, `-82`);
- hexadécimal, avec le préfixe `0x` (p.ex. `0x0`, `0x400`, `-0xAE00F23`);

Le type `int`

Une valeur de type `int` peut s'écrire en

- décimal, sans préfixe (p.ex. `0`, `1024`, `-82`);
- hexadécimal, avec le préfixe `0x` (p.ex. `0x0`, `0x400`, `-0xAE00F23`);
- binaire, avec le préfixe `0b` (p.ex. `0b1011011`, `-0b101`).

Le type `int`

Une valeur de type `int` peut s'écrire en

- décimal, sans préfixe (p.ex. `0`, `1024`, `-82`);
- hexadécimal, avec le préfixe `0x` (p.ex. `0x0`, `0x400`, `-0xAE00F23`);
- binaire, avec le préfixe `0b` (p.ex. `0b1011011`, `-0b101`).

La plage des `int` s'étend de `min_int` à `max_int`.

Sur un système `64` bits, ceci va de

$$-2^{62} = -4611686018427387904$$

à

$$2^{62} - 1 = 4611686018427387903.$$

Le type `int`

Une valeur de type `int` peut s'écrire en

- décimal, sans préfixe (p.ex. `0`, `1024`, `-82`);
- hexadécimal, avec le préfixe `0x` (p.ex. `0x0`, `0x400`, `-0xAE00F23`);
- binaire, avec le préfixe `0b` (p.ex. `0b1011011`, `-0b101`).

La plage des `int` s'étend de `min_int` à `max_int`.

Sur un système `64` bits, ceci va de

$$-2^{62} = -4611686018427387904$$

à

$$2^{62} - 1 = 4611686018427387903.$$

La plage ne s'étend pas de -2^{n-1} à $2^{n-1} - 1$: utilisation d'un bit pour la gestion automatique de la mémoire.

Opérations arithmétiques sur les `int`

Opérateur	Arité	Rôle
`-`, `+`	1	Moins, Plus (signe)
`-`, `+`	2	Soustraction, Addition
`/`, `*`	2	Division, Multiplication
`succ`	1	Successeur
`pred`	1	Prédécesseur
`abs`	1	Valeur absolue
`mod`	2	Modulo

Ces opérateurs produisent des valeurs de type `int`.

Opérations arithmétiques sur les `int`

Opérateur	Arité	Rôle
`-`, `+`	1	Moins, Plus (signe)
`-`, `+`	2	Soustraction, Addition
`/`, `*`	2	Division, Multiplication
`succ`	1	Successeur
`pred`	1	Prédécesseur
`abs`	1	Valeur absolue
`mod`	2	Modulo

Ces opérateurs produisent des valeurs de type `int`.

`succ`, `pred` et `abs` sont des fonctions.

Opérations arithmétiques sur les `int`

Opérateur	Arité	Rôle
`-`, `+`	1	Moins, Plus (signe)
`-`, `+`	2	Soustraction, Addition
`/`, `*`	2	Division, Multiplication
`succ`	1	Successeur
`pred`	1	Prédécesseur
`abs`	1	Valeur absolue
`mod`	2	Modulo

Ces opérateurs produisent des valeurs de type `int`.

`succ`, `pred` et `abs` sont des fonctions.

`mod` n'est pas une fonction, c'est un opérateur.

Opérations relationnelles sur les `int`

Opérateur	Arité	Rôle
`=`, `<>`	2	Égalité, Différence
`<`, `>`	2	Comparaison stricte
`<=`, `>=`	2	Comparaison large

Ces opérateurs produisent des valeurs de type `bool`.

Opérations relationnelles sur les `int`

Opérateur	Arité	Rôle
=, <>	2	Égalité, Différence
<, >	2	Comparaison stricte
<=, >=	2	Comparaison large

Ces opérateurs produisent des valeurs de type `bool`.

Par exemple :

```
# (2 = 1) || (32 <= 64);;
- : bool = true
```

Opérations bit à bit sur les `int`

Opérateur	Arité	Rôle
`lnot`	1	Non bit à bit
`land`	2	Et bit à bit
`lor`	2	Ou bit à bit
`lxor`	2	Ou exclusif bit à bit
`lsl`, `lsr`	2	Décalage à gauche, droite bit à bit
`asr`	2	Décalage à droite avec respect du signe bit à bit

Ces opérateurs produisent des valeurs de type `int`.

Opérations bit à bit sur les `int`

Opérateur	Arité	Rôle
`lnot`	1	Non bit à bit
`land`	2	Et bit à bit
`lor`	2	Ou bit à bit
`lxor`	2	Ou exclusif bit à bit
`lsl`, `lsr`	2	Décalage à gauche, droite bit à bit
`asr`	2	Décalage à droite avec respect du signe bit à bit

Ces opérateurs produisent des valeurs de type `int`.

Par exemple :

```
# 1 lsl 10;;
- : int = 1024

# (lnot 0) lsr 1;;
- : int = 4611686018427387903
```

Le type `float`

Le type `float` permet de représenter des nombres à virgule.

Le type `float`

Le type `float` permet de représenter des nombres à virgule.

Une valeur de type `float` s'écrit obligatoirement avec une virgule « `.` ».
P.ex., `4.52` est l'écriture du nombre 4.52.

Le type `float`

Le type `float` permet de représenter des nombres à virgule.

Une valeur de type `float` s'écrit obligatoirement avec une virgule « `.` ».
P.ex., `4.52` est l'écriture du nombre 4.52.

Le zéro à virgule s'écrit `0.`.

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

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

Le type `float`

Le type `float` permet de représenter des nombres à virgule.

Une valeur de type `float` s'écrit obligatoirement avec une virgule « `.` ».
P.ex., `4.52` est l'écriture du nombre 4.52.

Le zéro à virgule s'écrit `0.`.

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

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

La plage des `float` s'étend de -`max_float` à `max_float`.

Sur un système `64` bits, ceci va de

$$-\texttt{max\_float} = -1.7976931348623157 \times 10^{308}$$

à

$$\texttt{max\_float} = 1.7976931348623157 \times 10^{308}.$$

Opérations sur les `float`

Règle : les opérations arithmétiques sur les `int` ont leur analogue sur les `float` en ajoutant un « `.` » à l'opérateur : `-.`, `+.`, `/.`, `*.`.

```
# 1. +. 1.;;
- : float = 2.
```

Opérations sur les `float`

Règle : les opérations arithmétiques sur les `int` ont leur analogue sur les `float` en ajoutant un « `.` » à l'opérateur : `-.`, `+.`, `/.`, `*.`.

```
# 1. +. 1.;;
- : float = 2.
```

Attention : on ne peut pas mélanger les `int` et les `float` de manière non explicite :

```
# 2 +. 3.5;;
Error: This expression has type int but an expression was
         expected of type float
```

Opérations sur les `float`

Règle : les opérations arithmétiques sur les `int` ont leur analogue sur les `float` en ajoutant un « `.` » à l'opérateur : `-.`, `+.`, `/.`, `*.`.

```
# 1. +. 1.;;
- : float = 2.
```

Attention : on ne peut pas mélanger les `int` et les `float` de manière non explicite :

```
# 2 +. 3.5;;
Error: This expression has type int but an expression was
         expected of type float
```

On peut convertir un `int` en `float` par la fonction `float_of_int` :

```
# (float_of_int 32);;
- : float = 32.
```

Opérations sur les `float`

Règle : les opérations arithmétiques sur les `int` ont leur analogue sur les `float` en ajoutant un « `.` » à l'opérateur : `-.`, `+.`, `/.`, `*.`.

```
# 1. +. 1.;;
- : float = 2.
```

Attention : on ne peut pas mélanger les `int` et les `float` de manière non explicite :

```
# 2 +. 3.5;;
Error: This expression has type int but an expression was
         expected of type float
```

On peut convertir un `int` en `float` par la fonction `float_of_int` :

```
# (float_of_int 32);;
- : float = 32.
```

et un `float` en `int` par la fonction `int_of_float` (troncature) :

```
# (int_of_float 21.9);;
- : int = 21
```

Opérations sur les `float`

Les opérateurs relationnels sur les `float` sont les mêmes que ceux sur les `int`.

```
# 32. <= 89.99;;
- : bool = true
```

Opérations sur les `float`

Les opérateurs relationnels sur les `float` sont les mêmes que ceux sur les `int`.

```
# 32. <= 89.99;;
- : bool = true
```

Ici aussi, on ne peut pas mélanger les `int` et les `float` de manière non explicite :

```
# 67.67 = 8;;
Error: This expression has type int but an expression was
         expected of type float
```

Opérations sur les `float`

Les opérateurs relationnels sur les `float` sont les mêmes que ceux sur les `int`.

```
# 32. <= 89.99;;
- : bool = true
```

Ici aussi, on ne peut pas mélanger les `int` et les `float` de manière non explicite :

```
# 67.67 = 8;;
Error: This expression has type int but an expression was
         expected of type float
```

Il faut en revanche écrire

```
# 67.67 = (float_of_int 8);;
- : bool = false
```

Opérations sur les `float`

Les opérateurs relationnels sur les `float` sont les mêmes que ceux sur les `int`.

```
# 32. <= 89.99;;
- : bool = true
```

Ici aussi, on ne peut pas mélanger les `int` et les `float` de manière non explicite :

```
# 67.67 = 8;;
Error: This expression has type int but an expression was
         expected of type float
```

Il faut en revanche écrire

```
# 67.67 = (float_of_int 8);;
- : bool = false
```

pour demander explicitement la conversion d'un `int` en un `float`.

Opérations sur les `float`

Il existe des opérateurs spécifiques aux `float`. Entre autres :

Opérateur	Arité	Rôle
`abs_float`	1	Valeur absolue
`floor`, `ceil`	1	Partie entière inf., sup.
`sqrt`	1	Racine carrée
`log`, `exp`	1	Logarithme népérien, exponentielle
`cos`, `sin`, `tan`	1	Fonctions trigonométriques
`**`	2	Exponentiation

Ces opérateurs produisent des valeurs de type `float`.

Opérations sur les `float`

Il existe des opérateurs spécifiques aux `float`. Entre autres :

Opérateur	Arité	Rôle
`abs_float`	1	Valeur absolue
`floor`, `ceil`	1	Partie entière inf., sup.
`sqrt`	1	Racine carrée
`log`, `exp`	1	Logarithme népérien, exponentielle
`cos`, `sin`, `tan`	1	Fonctions trigonométriques
`**`	2	Exponentiation

Ces opérateurs produisent des valeurs de type `float`.

Hormis `**` qui est bien un opérateur du langage, les autres sont en réalité des fonctions prédéfinies.

Le type `char`

Le type `char` permet de représenter les caractères ASCII.

Une valeur de type `char` peut s'écrire

- par un caractère entre apostrophes (p.ex., `'a'`, `'8'`, `'?'`);
- par son code ASCII, sur trois chiffres précédés de `\`, le tout entre apostrophes (p. ex `'\101'`, `'\000'`, `'\035'`).

Le type `char`

Le type `char` permet de représenter les caractères ASCII.

Une valeur de type `char` peut s'écrire

- par un caractère entre apostrophes (p.ex., `'a'`, `'8'`, `'?'`);
- par son code ASCII, sur trois chiffres précédés de `\`, le tout entre apostrophes (p. ex `'\101'`, `'\000'`, `'\035'`).

La fonction `int_of_char` calcule le code ASCII d'un caractère.

```
# (int_of_char 'G');;
- : int = 71
```

Le type `char`

Le type `char` permet de représenter les caractères ASCII.

Une valeur de type `char` peut s'écrire

- par un caractère entre apostrophes (p.ex., `'a'`, `'8'`, `'?'`);
- par son code ASCII, sur trois chiffres précédés de `\`, le tout entre apostrophes (p. ex `'\101'`, `'\000'`, `'\035'`).

La fonction `int_of_char` calcule le code ASCII d'un caractère.

```
# (int_of_char 'G');;
- : int = 71
```

La fonction `char_of_int` calcule le caractère de code ASCII spécifié.

```
# (char_of_int 40);;
- : char = '('
```

```
# (char_of_int 2);;
- : char = '\002'
```

Le type `string`

Le type `string` permet de représenter des chaînes de caractères.

Le type `string`

Le type `string` permet de représenter des chaînes de caractères.

Une chaîne de caractères s'écrit par une suite de caractères entre guillemets. P.ex., `"abc123"`, `"\100\101f"`.

Le type `string`

Le type `string` permet de représenter des chaînes de caractères.

Une chaîne de caractères s'écrit par une suite de caractères entre guillemets. P.ex., `"abc123"`, `"\100\101f"`.

L'opérateur `^` permet de concaténer deux chaînes de caractères. P.ex.,

```
# "abc" ^ "def";;
- : string = "abcdef"
```

Le type `string`

Le type `string` permet de représenter des chaînes de caractères.

Une chaîne de caractères s'écrit par une suite de caractères entre guillemets. P.ex., `"abc123"`, `"\100\101f"`.

L'opérateur `^` permet de concaténer deux chaînes de caractères. P.ex.,

```
# "abc" ^ "def";;
- : string = "abcdef"


# let u = "ab"
  and v = "ba" in
      u ^ v ^ u ^ v;;
- : string = "abbaabba"
```

Le type `string`

Le type `string` permet de représenter des chaînes de caractères.

Une chaîne de caractères s'écrit par une suite de caractères entre guillemets. P.ex., `"abc123"`, `"\100\101f"`.

L'opérateur `^` permet de concaténer deux chaînes de caractères. P.ex.,

```
# "abc" ^ "def";;
- : string = "abcdef"
```

```
# let u = "ab"
  and v = "ba" in
      u ^ v ^ u ^ v;;
- : string = "abbaabba"
```

Plus précisément, si `u` et `v` sont des noms liés à des chaînes de caractères, `u ^ v` est une expression dont la valeur est la concaténation de `u` et de `v`.

Le type `string`

Le type `string` permet de représenter des chaînes de caractères.

Une chaîne de caractères s'écrit par une suite de caractères entre guillemets. P.ex., `"abc123"`, `"\100\101f"`.

L'opérateur `^` permet de concaténer deux chaînes de caractères. P.ex.,

```
# "abc" ^ "def";;
- : string = "abcdef"
```

```
# let u = "ab"
  and v = "ba" in
      u ^ v ^ u ^ v;;
- : string = "abbaabba"
```

Plus précisément, si `u` et `v` sont des noms liés à des chaînes de caractères, `u ^ v` est une expression dont la valeur est la concaténation de `u` et de `v`.

Il n'y a pas d'effet de bord : les chaînes `u` et `v` ne sont pas modifiées lors de leur concaténation.

Le type `string`

Il existe des fonctions de conversion autour du type `string`. Parmi celles-ci, il y a

- `string_of_bool` et `bool_of_string`;
- `string_of_int` et `int_of_string`;
- `string_of_float` et `float_of_string`.

Le type `string`

Il existe des fonctions de conversion autour du type `string`. Parmi celles-ci, il y a

- `string_of_bool` et `bool_of_string`;
- `string_of_int` et `int_of_string`;
- `string_of_float` et `float_of_string`.

Les opérateurs relationnels sont bien définis sur les valeurs de type `string`. On peut ainsi totalement comparer des chaînes de caractères.

Les opérateurs de comparaison travaillent selon l'ordre lexicographique.

Le type `string`

Il existe des fonctions de conversion autour du type `string`. Parmi celles-ci, il y a

- `string_of_bool` et `bool_of_string`;
- `string_of_int` et `int_of_string`;
- `string_of_float` et `float_of_string`.

Les opérateurs relationnels sont bien définis sur les valeurs de type `string`. On peut ainsi totalement comparer des chaînes de caractères.

```
# "abc" = "abde";;
- : bool = false
```

Les opérateurs de comparaison travaillent selon l'ordre lexicographique.

Le type `string`

Il existe des fonctions de conversion autour du type `string`. Parmi celles-ci, il y a

- `string_of_bool` et `bool_of_string`;
- `string_of_int` et `int_of_string`;
- `string_of_float` et `float_of_string`.

Les opérateurs relationnels sont bien définis sur les valeurs de type `string`. On peut ainsi totalement comparer des chaînes de caractères.

```
# "abc" = "abde";;
- : bool = false
```

```
# let u = "ab" and v = "ab" in
      u = v;;
- : bool = true
```

Les opérateurs de comparaison travaillent selon l'ordre lexicographique.

Le type `string`

Il existe des fonctions de conversion autour du type `string`. Parmi celles-ci, il y a

- `string_of_bool` et `bool_of_string`;
- `string_of_int` et `int_of_string`;
- `string_of_float` et `float_of_string`.

Les opérateurs relationnels sont bien définis sur les valeurs de type `string`. On peut ainsi totalement comparer des chaînes de caractères.

```
# "abc" = "abde";;
- : bool = false
```

```
# let u = "ab" and v = "ab" in
      u = v;;
- : bool = true
```

```
# "abc" <= "aaaa";;
- : bool = false
```

Les opérateurs de comparaison travaillent selon l'ordre lexicographique.

Le type `string`

Il existe des fonctions de conversion autour du type `string`. Parmi celles-ci, il y a

- `string_of_bool` et `bool_of_string`;
- `string_of_int` et `int_of_string`;
- `string_of_float` et `float_of_string`.

Les opérateurs relationnels sont bien définis sur les valeurs de type `string`. On peut ainsi totalement comparer des chaînes de caractères.

```
# "abc" = "abde";;
- : bool = false
```

```
# let u = "ab" and v = "ab" in
      u = v;;
- : bool = true
```

```
# "abc" <= "aaaa";;
- : bool = false
```

```
# "abc" < "ad";;
- : bool = true
```

Les opérateurs de comparaison travaillent selon l'ordre lexicographique.

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

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

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

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

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

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

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

La notion de fonction

D'un point de vue formel, une fonction

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

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

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

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

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

La notion de fonction

D'un point de vue formel, une fonction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Les noms x_1 et x_2 sont les paramètres de f.

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

Les noms x_1 et x_2 sont les paramètres de f.

Lors de l'appel $f(2, 6)$, 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 $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ la fonction définie par $f(x_1, x_2) := x_1 + x_2$.

Les noms x_1 et x_2 sont les paramètres de f.

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

Lors d'un appel à une fonction f avec les arguments x_1, …, x_n, on dit que l'on applique f à x_1, …, 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 =
      let aux u =
          u ^ "a" ^ u
      in
      (aux u) ^ (aux ("b" ^ u));;
val f : string -> string = <fun>
```

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

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

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

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"
```

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

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

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

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

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

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

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

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
```

$\rightarrow$ `if true then 21 else 24` $\rightarrow$ `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
        ()
```

est équivalent à

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

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 `E1`, …, `En` 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 `E1`, …, `En` et `S` sont des types.

Plus précisément,

- pour tout $1 \leqslant i \leqslant n$, `Ei` est le type attendu du i^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 `E1`, …, `En` et `S` sont des types.

Plus précisément,

- pour tout $1 \leqslant i \leqslant n$, `Ei` est le type attendu du i^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 `E1`, …, `En` et `S` sont des types.

Plus précisément,

- pour tout $1 \leqslant i \leqslant n$, `Ei` est le type attendu du i^e argument de `F` ;
- `S` est le type de retour de `F`.

`E1 -> ... -> En -> 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 -> E2 -> 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 -> E2 -> 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 -> E2 -> 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 -> 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 -> E2 -> 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 -> S` qui est donc une **fonction**.

La fonction `(f e1)` est ainsi une fonction qui se comporte comme `f` lorsque son 1er 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 -> E2 -> 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 -> S` qui est donc une **fonction**.

La fonction `(f e1)` est ainsi une fonction qui se comporte comme `f` lorsque son 1er 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 -> ... -> En -> S
```

et `e1`, …, `ek` sont des valeurs de types respectifs `E1`, …, `Ek` avec $k \leqslant n$,

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 -> ... -> En -> S
```

et `e1`, …, `ek` sont des valeurs de types respectifs `E1`, …, `Ek` avec $k \leqslant n$, l'application partielle

```
(f e1 ... ek)
```

est une fonction de type

```
Ek+1 -> ... -> En -> 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 $f : \mathbb{Z}^3 \to \mathbb{Z}$ 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 $f : \mathbb{Z}^3 \to \mathbb{Z}$ vérifiant $(a, b, c) \mapsto ab + ac$.

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

Cette fonction représente $f_3 : \mathbb{Z}^2 \to \mathbb{Z}$ 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 $f : \mathbb{Z}^3 \to \mathbb{Z}$ vérifiant $(a, b, c) \mapsto ab + ac$.

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

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

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

Cette fonction représente $f_{3,5} : \mathbb{Z} \to \mathbb{Z}$ 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 $f : \mathbb{Z}^3 \to \mathbb{Z}$ vérifiant $(a, b, c) \mapsto ab + ac$.

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

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

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

Cette fonction représente $f_{3,5} : \mathbb{Z} \to \mathbb{Z}$ 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)
```