

Axe 2 : concepts premiers

Programmation

Pratique

Types

Plan

Programmation

- Interpréteur CAML

- Liaisons

- Types de base

- Fonctions

La 1^{re} chose à apprendre

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

La 1^{re} chose à apprendre

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

— J.-P. Duval

La 1^{re} chose à apprendre

« La 1^{re} 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 1^{re} chose à apprendre

« La 1^{re} 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 1^{re} chose à apprendre

« La 1^{re} 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 1^{re} chose à apprendre

« La 1^{re} 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

Programmation

Interpréteur CAML

Liaisons

Types de base

Fonctions

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 3^e 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

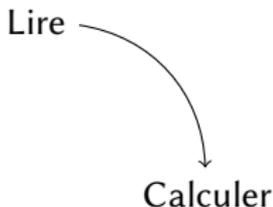
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
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

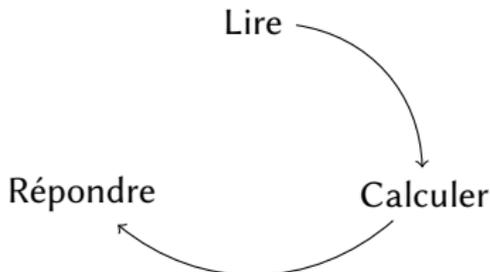
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
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

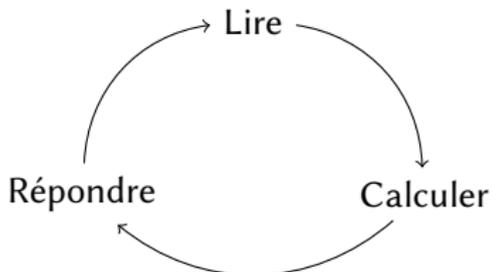
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
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

Programmation

Interpréteur CAML

Liaisons

Types de base

Fonctions

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.

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**.

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.

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 1^{re} 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
```

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

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

La 2^e 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 2^e 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 2^e 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 2^e phrase est locale : le nom global **n** défini en 1^{re} 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 2^e 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 2^e phrase est locale : le nom global **n** défini en 1^{re} phrase reste inchangé.

Ceci renseigne sur la valeur du nom **n** de la 3^e 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
```

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

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.

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
```

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

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.

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

Programmation

Interpréteur CAML

Liaisons

Types de base

Fonctions

Les six types de base

Nom du type	Utilisation
<code>int</code>	Représentation des entiers signés
<code>float</code>	Représentation des nombres à virgule signés
<code>char</code>	Représentation des caractères
<code>string</code>	Représentation des chaînes de caractères
<code>bool</code>	Représentation des booléens
<code>unit</code>	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
<code>not</code>	1	Non logique
<code>&&</code>	2	Et logique
<code> </code>	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
<code>not</code>	1	Non logique
<code>&&</code>	2	Et logique
<code> </code>	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
<code>not</code>	1	Non logique
<code>&&</code>	2	Et logique
<code> </code>	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
<code>-</code> , <code>+</code>	1	Moins, Plus (signe)
<code>-</code> , <code>+</code>	2	Soustraction, Addition
<code>/</code> , <code>*</code>	2	Division, Multiplication
<code>succ</code>	1	Successeur
<code>pred</code>	1	Prédécesseur
<code>abs</code>	1	Valeur absolue
<code>mod</code>	2	Modulo

Ces opérateurs produisent des valeurs de type `int`.

Opérations arithmétiques sur les `int`

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

Ces opérateurs produisent des valeurs de type `int`.

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

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

Opérations relationnelles sur les `int`

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

Ces opérateurs produisent des valeurs de type `bool`.

Opérations relationnelles sur les `int`

Opérateur	Arité	Rôle
<code>=, <></code>	2	Égalité, Différence
<code><, ></code>	2	Comparaison stricte
<code><=, >=</code>	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
<code>lnot</code>	1	Non bit à bit
<code>land</code>	2	Et bit à bit
<code>lor</code>	2	Ou bit à bit
<code>lxor</code>	2	Ou exclusif bit à bit
<code>lsl, lsr</code>	2	Décalage à gauche, droite bit à bit
<code>asr</code>	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
<code>lnot</code>	1	Non bit à bit
<code>land</code>	2	Et bit à bit
<code>lor</code>	2	Ou bit à bit
<code>lxor</code>	2	Ou exclusif bit à bit
<code>lsl, lsr</code>	2	Décalage à gauche, droite bit à bit
<code>asr</code>	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

$$-\text{max_float} = -1.79769313486231571 \times 10^{308}$$

à

$$\text{max_float} = 1.79769313486231571 \times 10^{308}.$$

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
<code>abs_float</code>	1	Valeur absolue
<code>floor, ceil</code>	1	Partie entière inf., sup.
<code>sqrt</code>	1	Racine carrée
<code>log, exp</code>	1	Logarithme népérien, exponentielle
<code>cos, sin, tan</code>	1	Fonctions trigonométriques
<code>**</code>	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
<code>abs_float</code>	1	Valeur absolue
<code>floor, ceil</code>	1	Partie entière inf., sup.
<code>sqrt</code>	1	Racine carrée
<code>log, exp</code>	1	Logarithme népérien, exponentielle
<code>cos, sin, tan</code>	1	Fonctions trigonométriques
<code>**</code>	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_of_int 2);;  
- : char = '('                                       - : 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";;                # let u = "ab" and v = "ab" in
- : bool = false                  u = v;;
                                  - : bool = true

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

Les opérateurs de comparaison travaillent selon l'ordre lexicographique.