Typage explicite *vs* implicite

La plupart des langages de programmation manipulent des variables et des valeurs qui possèdent un type (ou même plusieurs).

Typage explicite *vs* implicite

La plupart des langages de programmation manipulent des variables et des valeurs qui possèdent un type (ou même plusieurs).

Il y a deux manières principales de **connaître les types** des variables utilisées dans un programme :

1. typage explicite, où les **types** des variables sont **mentionnés** dans le programme ;

Typage explicite *vs* implicite

La plupart des langages de programmation manipulent des variables et des valeurs qui possèdent un type (ou même plusieurs).

Il y a deux manières principales de **connaître les types** des variables utilisées dans un programme :

1. typage explicite, où les **types** des variables sont **mentionnés** dans le programme ;
2. typage implicite, où les **types** des variables ne sont **pas mentionnés** dans le programme. Ils sont devinés lors de la compilation (si typage statique) ou lors de l'exécution (si typage dynamique) en fonction du contexte.

Typage explicite *vs* implicite

La plupart des langages de programmation manipulent des variables et des valeurs qui possèdent un type (ou même plusieurs).

Il y a deux manières principales de **connaître les types** des variables utilisées dans un programme :

1. typage explicite, où les **types** des variables sont **mentionnés** dans le programme ;
2. typage implicite, où les **types** des variables ne sont **pas mentionnés** dans le programme. Ils sont devinés lors de la compilation (si typage statique) ou lors de l'exécution (si typage dynamique) en fonction du contexte.

   Ce mécanisme s'appelle l'inférence des types.

Typage explicite

Considérons le programme C

```
#include <stdio.h>

typedef struct {
    int x, y;
} Point2D;

typedef struct {
    int x, y, z;
} Point3D;

void afficher(Point2D *p) {
    printf("(%d, %d)", p->x, p->y);
}

int main() {
    Point3D p;
    p.x = 1, p.y = 2, p.z = 3;
    afficher(&p);
    return 0;
}
```

Typage explicite

Considérons le programme C

```
#include <stdio.h>

typedef struct {
    int x, y;
} Point2D;

typedef struct {
    int x, y, z;
} Point3D;

void afficher(Point2D *p) {
    printf("(%d, %d)", p->x, p->y);
}

int main() {
    Point3D p;
    p.x = 1, p.y = 2, p.z = 3;
    afficher(&p);
    return 0;
}
```

Dans la fonction `main`, on a demandé explicitement que la variable `p` soit de type `Point3D`. La fonction `afficher` demande l'existence des champs `x` et `y` de son argument. Ces champs existent dans `p` et cependant, ce code ne compile pas.

Typage explicite

Considérons le programme C

```
#include <stdio.h>

typedef struct {
    int x, y;
} Point2D;

typedef struct {
    int x, y, z;
} Point3D;

void afficher(Point2D *p) {
    printf("(%d, %d)", p->x, p->y);
}

int main() {
    Point3D p;
    p.x = 1, p.y = 2, p.z = 3;
    afficher(&p);
    return 0;
}
```

Dans la fonction `main`, on a demandé explicitement que la variable `p` soit de type `Point3D`. La fonction `afficher` demande l'existence des champs `x` et `y` de son argument. Ces champs existent dans `p` et cependant, ce code ne compile pas.

En effet, même si une variable de type `Point3D` semble pouvoir être utilisée comme une variable de type `Point2D`, ceci est impossible en C qui est un langage à typage explicite : le type de `p` a été fixé lors de la déclaration de `afficher`.

Typage implicite

Considérons le programme Caml

```
let suivant x = x + 1 in
(print_int (suivant 5))
```

Typage implicite

Considérons le programme Caml

```
let suivant x = x + 1 in
(print_int (suivant 5))
```

Ici, `suivant` est une fonction. Le type de son paramètre `x` n'est pas spécifié. Cependant, le fait que l'on réalise une opération arithmétique (`x + 1`) avec ce paramètre indique qu'il s'agit d'un entier.

Typage implicite

Considérons le programme Caml

```
let suivant x = x + 1 in
(print_int (suivant 5))
```

Ici, `suivant` est une fonction. Le type de son paramètre `x` n'est pas spécifié. Cependant, le fait que l'on réalise une opération arithmétique (`x + 1`) avec ce paramètre indique qu'il s'agit d'un entier.

De cette manière, le mécanisme d'inférence des types suggère au compilateur que `suivant` est une fonction qui accepte un entier et qui renvoie un entier.

Typage implicite

Considérons le programme Caml

```
let suivant x = x + 1 in
(print_int (suivant 5))
```

Ici, `suivant` est une fonction. Le type de son paramètre `x` n'est pas spécifié. Cependant, le fait que l'on réalise une opération arithmétique (`x + 1`) avec ce paramètre indique qu'il s'agit d'un entier.

De cette manière, le mécanisme d'inférence des types suggère au compilateur que `suivant` est une fonction qui accepte un entier et qui renvoie un entier.

Cette valeur renvoyée peut donc être traitée par la fonction `print_int` qui accepte en entrée des valeurs entières.

Avantages et inconvénients

Typage explicite.

- Avantages : meilleure lisibilité des programmes. En général, la compilation et l'exécution sont plus efficaces.

Avantages et inconvénients

Typage explicite.

- Avantages : meilleure lisibilité des programmes. En général, la compilation et l'exécution sont plus efficaces.
- Inconvénients : programmes verbeux.

Avantages et inconvénients

Typage explicite.

- Avantages : meilleure lisibilité des programmes. En général, la compilation et l'exécution sont plus efficaces.
- Inconvénients : programmes verbeux.

Typage implicite.

- Avantages : programmes plus concis, le programmeur ne se préoccupe pas d'indiquer les types : le compilateur se charge de trouver ceux qui sont les plus adaptés.

Avantages et inconvénients

Typage explicite.

- Avantages : meilleure lisibilité des programmes. En général, la compilation et l'exécution sont plus efficaces.
- Inconvénients : programmes verbeux.

Typage implicite.

- Avantages : programmes plus concis, le programmeur ne se préoccupe pas d'indiquer les types : le compilateur se charge de trouver ceux qui sont les plus adaptés.
- Inconvénients : programmes moins lisibles. Si le typage est statique, la compilation peut être plus longue (il faut deviner les types). Si le typage est dynamique, l'exécution peut être moins efficace.

Portée statique *vs* dynamique

La portée d'un identificateur dans un programme désigne l'endroit dans lequel il est possible de l'utiliser.

Portée statique *vs* dynamique

La portée d'un identificateur dans un programme désigne l'endroit dans lequel il est possible de l'utiliser.

Les identificateurs dans un langage de programmation peuvent être de deux principales catégories :

1. de portée statique, où chaque occurrence d'un identificateur doit être mis en relation avec sa définition préalable ;

Portée statique *vs* dynamique

La portée d'un identificateur dans un programme désigne l'endroit dans lequel il est possible de l'utiliser.

Les identificateurs dans un langage de programmation peuvent être de deux principales catégories :

1. de portée statique, où chaque occurrence d'un identificateur doit être mis en relation avec sa définition préalable ;
2. de portée dynamique, où ce que représente un identificateur peut dépendre de l'exécution du programme (dans certains *mauvais* cas, il peut même ne rien représenter du tout).

Portée statique

Considérons le programme Caml

```
let fact n =
    if n <= 1 then
        1
    else
        n * (fact (n - 1))
in
(fact 3)
```

Portée statique

Considérons le programme Caml

```
let fact n =
    if n <= 1 then
        1
    else
        n * (fact (n - 1))
in
(fact 3)
```

Lors de sa compilation, une erreur se produit : l'identificateur `fact`, utilisé en l. 5 est encore non défini.

On obtient le message suivant du compilateur :

```
File "Prog.ml", line 5, characters 12-16:
Error: Unbound value fact
```

Portée dynamique

Considérons le programme Python

```
n = int(input())      # Lecture d'un entier.
if n == 0 :
    res = "a"         # ‘res‘ est une chaîne de caractères.
elif n == 1 :
    res = [1, 2, 3]   # ‘res‘ est une liste.
print res              # Tentative d'affichage de ‘res‘.
```

Portée dynamique

Considérons le programme Python

```
n = int(input())      # Lecture d'un entier.
if n == 0 :
    res = "a"         # ‘res‘ est une chaîne de caractères.
elif n == 1 :
    res = [1, 2, 3]   # ‘res‘ est une liste.
print res             # Tentative d'affichage de ‘res‘.
```

Lors de son exécution, l'identificateur `res` en l. 6 peut faire référence à des définitions différentes :

- si l'utilisateur saisit `0`, `res` est défini comme étant la chaîne `"a"` ;

Portée dynamique

Considérons le programme Python

```
n = int(input())      # Lecture d'un entier.
if n == 0 :
    res = "a"         # ‘res‘ est une chaîne de caractères.
elif n == 1 :
    res = [1, 2, 3]   # ‘res‘ est une liste.
print res             # Tentative d'affichage de ‘res‘.
```

Lors de son exécution, l'identificateur `res` en l. 6 peut faire référence à des définitions différentes :

- si l'utilisateur saisit `0`, `res` est défini comme étant la chaîne `"a"` ;
- si l'utilisateur saisit `1`, `res` est défini comme étant la liste contenant successivement les éléments `1`, `2` et `3` ;

Portée dynamique

Considérons le programme Python

```
n = int(input())      # Lecture d'un entier.
if n == 0 :
    res = "a"         # ‘res‘ est une chaîne de caractères.
elif n == 1 :
    res = [1, 2, 3]   # ‘res‘ est une liste.
print res             # Tentative d'affichage de ‘res‘.
```

Lors de son exécution, l'identificateur `res` en l. 6 peut faire référence à des définitions différentes :

- si l'utilisateur saisit `0`, `res` est défini comme étant la chaîne `"a"` ;
- si l'utilisateur saisit `1`, `res` est défini comme étant la liste contenant successivement les éléments `1`, `2` et `3` ;
- dans tous les autres cas, `res` est un identificateur non défini.

  Cette information est donnée, **lors de l'exécution**, par

```
Traceback (most recent call last):
  File "Prog.py", line 6, in <module>
    print res

NameError: name 'res' is not defined
```

Langages fonctionnels purs *vs* impurs

Il y a deux sortes de langages fonctionnels :

1. les langages fonctionnels purs, où tout effet de bord est interdit. Des problèmes se posent notamment lors de la gestion des entrées sorties (mais qui sont résolus élégamment au moyen des *monades*).

Langages fonctionnels purs *vs* impurs

Il y a deux sortes de langages fonctionnels :

1. les langages fonctionnels purs, où tout effet de bord est interdit. Des problèmes se posent notamment lors de la gestion des entrées sorties (mais qui sont résolus élégamment au moyen des *monades*).

2. les langages fonctionnels impurs, où certaines particularités des langages impératifs sont utilisables, comme la gestion classique des entrées/sorties, les affectations ou encore les instructions de boucle.

Caractéristiques des principaux langages

Langage	T. dyn.	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	Oui	Impur
Haskell	Non	Oui	Non	Oui	Non	Pur

Caractéristiques des principaux langages

Langage	T. dyn.	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

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 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 4.01.0

#
```

Interpréteur

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

```
        Objective Caml version 4.01.0

#
```

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 4.01.0

#
```

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

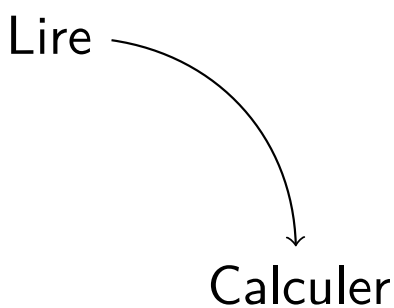
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.01.0

#
```

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

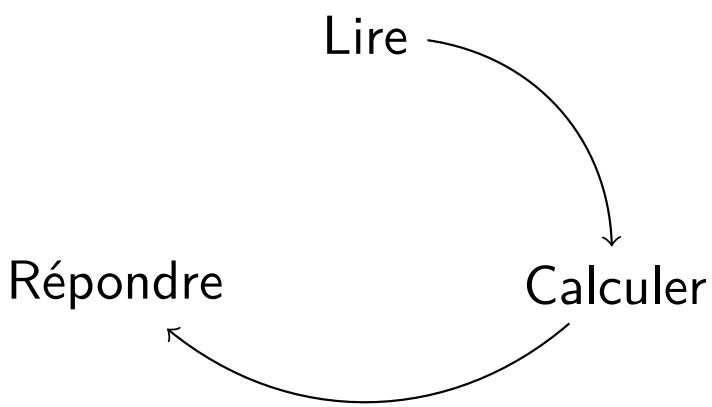
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.01.0

#
```

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

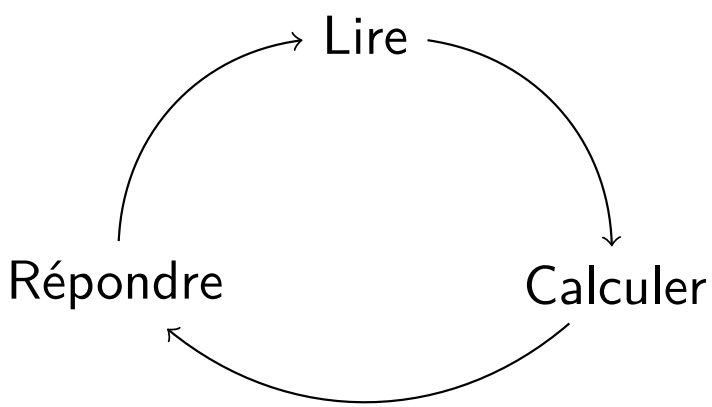
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.01.0

#
```

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

« *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 = EXP
```

où `ID` est un nom (identificateur) et `EXP` 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 = EXP`

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

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

# 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 = EXP`

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

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

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

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

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 valeur 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 valeur et `EXP` est une expression.

Cette expression possède une valeur : celle de `EXP`.

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 valeur et `EXP` est une expression.

Cette expression possède une valeur : celle de `EXP`.

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

La 2e occurrence de `n` a pour valeur `5` à cause de la liaison 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 valeur et `EXP` est une expression.

Cette expression possède une valeur : celle de `EXP`.

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

La 2e occurrence de `n` a pour valeur `5` à cause de la liaison 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 valeur et `EXP` est une expression.

Cette expression possède une valeur : celle de `EXP`.

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

La 2e occurrence de `n` a pour valeur `5` à cause de la liaison 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 explique 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
```

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;;
```

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, l'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 simultanées

Il est possible de définir des noms simultanément dans une même phrase.

Liaisons simultanées

Il est possible de définir des noms simultanément dans une même phrase.

On utilise pour cela la construction syntaxique

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

où `ID1` et `ID2` sont des identificateurs et `VAL1` et `VAL2` sont des valeurs.

Liaisons simultanées

Il est possible de définir des noms simultanément dans une même phrase.

On utilise pour cela la construction syntaxique

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

où `ID1` et `ID2` sont des identificateurs et `VAL1` et `VAL2` sont des valeurs.

```
# 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 simultanées

Il est possible de définir des noms simultanément dans une même phrase.

On utilise pour cela la construction syntaxique

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

où `ID1` et `ID2` sont des identificateurs et `VAL1` et `VAL2` sont des valeurs.

```
# 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
```

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;;
```

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.

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"
```

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`.

```
# 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`.

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 `string`. On peut ainsi totalement comparer des chaînes de caractères.

Les opérateurs de comparaison considèrent 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 `string`. On peut ainsi totalement comparer des chaînes de caractères.

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

Les opérateurs de comparaison considèrent 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 `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 considèrent 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 `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 considèrent 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 `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 considèrent 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

$$f : E_1 \times E_2 \times \cdots \times E_n \to S$$

est une partie du produit cartésien $E_1 \times E_2 \times \cdots \times E_n \times S$

La notion de fonction

D'un point de vue formel, une fonction

$$f : E_1 \times E_2 \times \cdots \times E_n \to S$$

est une partie du produit cartésien $E_1 \times E_2 \times \cdots \times E_n \times S$ telle que

$$(e_1, e_2, \ldots, e_n, s) \in f \text{ et } (e_1, e_2, \ldots, e_n, s') \in f \quad \text{implique} \quad s = s'.$$

La notion de fonction

D'un point de vue formel, une fonction

$$f : E_1 \times E_2 \times \cdots \times E_n \to S$$

est une partie du produit cartésien $E_1 \times E_2 \times \cdots \times E_n \times S$ telle que

$$(e_1, e_2, \ldots, e_n, s) \in f \text{ et } (e_1, e_2, \ldots, e_n, s') \in f \quad \text{implique} \quad s = s'.$$

D'usage, la propriété $(e_1, e_2, \ldots, e_n, s) \in f$ est ainsi notée

$$f(e_1, e_2, \ldots, e_n) = s.$$

La notion de fonction

D'un point de vue formel, une fonction

$$f : E_1 \times E_2 \times \cdots \times E_n \to S$$

est une partie du produit cartésien $E_1 \times E_2 \times \cdots \times E_n \times S$ telle que

$$(e_1, e_2, \ldots, e_n, s) \in f \text{ et } (e_1, e_2, \ldots, e_n, s') \in f \quad \text{implique} \quad s = s'.$$

D'usage, la propriété $(e_1, e_2, \ldots, e_n, s) \in f$ est ainsi notée

$$f(e_1, e_2, \ldots, e_n) = s.$$

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

La notion de fonction

D'un point de vue formel, une fonction

$$f : E_1 \times E_2 \times \cdots \times E_n \to S$$

est une partie du produit cartésien $E_1 \times E_2 \times \cdots \times E_n \times S$ telle que

$$(e_1, e_2, \ldots, e_n, s) \in f \text{ et } (e_1, e_2, \ldots, e_n, s') \in f \quad \text{implique} \quad s = s'.$$

D'usage, la propriété $(e_1, e_2, \ldots, e_n, s) \in f$ est ainsi notée

$$f(e_1, e_2, \ldots, e_n) = s.$$

On appelle `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

$$f : E_1 \times E_2 \times \cdots \times E_n \to S$$

est une partie du produit cartésien $E_1 \times E_2 \times \cdots \times E_n \times S$ telle que

$$(e_1, e_2, \ldots, e_n, s) \in f \text{ et } (e_1, e_2, \ldots, e_n, s') \in f \quad \text{implique} \quad s = s'.$$

D'usage, la propriété $(e_1, e_2, \ldots, e_n, s) \in f$ est ainsi notée

$$f(e_1, e_2, \ldots, e_n) = s.$$

On appelle `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 $f(x, y) := x + y$.

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 $f(x, y) := x + y$.

Les noms x et y 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 $f(x, y) := x + y$.

Les noms x et y 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 $f(x, y) := x + y$.

Les noms x et y 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 $e_1, \ldots, e_n$, on dit que l'on applique f à $e_1, \ldots, e_n$.