Exemple

Considérons le programme OCaml

```
let pair (x : int) (y : bool) : (int * bool) =
    (x, y)

let () =
    let p = pair 2 true in
    print_int (fst p)
```

Les **annotations de types explicites** indiquent que `pair` est une fonction acceptant un entier et un booléen et renvoyant un couple dont la 1re composante est un entier et la 2e est un booléen.

Ces annotations de types forment aussi des **contraintes** : par exemple, l'appel `pair 2 5` provoquerait une erreur lors de la vérification des types.

Exemple

Considérons le programme OCaml, similaire au précédent,

```
let pair x y =
    (x, y)

let () =
    let p1 = pair 2 true in
    print_int (fst p1);
    let p2 = pair 2 5 in
    print_int (snd p2)
```

Ici, les **annotations de types sont implicites** au niveau de la définition de la fonction `pair`.

Le système d'inférence des types lui attribue automatiquement un type. Il devine le **type le plus général possible** pour cette fonction.

Ce type est `'a -> 'b -> 'a * 'b`, mentionnant que `pair` est une fonction acceptant une valeur de n'importe quel type (`'a`), une valeur de n'importe quel type (`'b`, potentiellement différent de `'a`) et renvoyant un couple dont la 1re composante est de type `'a` et la 2e est de type `'b`.

Nous observons ici que la fonction `pair` est appelée deux fois et avec des types d'arguments différents.

Tout ceci est en lien avec le **polymorphisme** (notion qui sera vue en détail plus tard).

Exemple

Considérons le programme OCaml

```
let next x =
    x + 1

let () =
    print_int (next 5)
```

Ici, `next` 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 système d'inférence des types suggère que `next` 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.

Attribution explicite.

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

Attribution implicite.

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

Note : dans certains langages, il est possible d'attribuer explicitement certains types et de laisser le système d'inférence de types en deviner certains. L'OCaml s'inscrit dans cette approche hybride.

5.4. Portée d'un identificateur

La *portée* d'un identificateur (de variable, de fonction, ou encore dans certains cas, de types) dans un programme désigne l'étendue dans laquelle cet identificateur existe.

Il existe deux principales sortes de portées :

1. *portée statique*, où la portée d'un identificateur dépend de sa **position** dans le programme ;
2. *portée dynamique*, où la portée d'un identificateur dépend de la façon dont le programme s'**exécute**. Elle peut donc varier d'une exécution à une autre.

Note : les identificateurs à portée dynamique sont peu courants dans les langages modernes.

Exemple

Considérons le programme OCaml

```
let () =
    let x = 10 in
    let y = x + 1 in
    print_int (x + y)
```

Ici, la portée de `x` va de la l. 3 à la l. 4. La portée de `y` va de la l. 4 à la l. 4.

Ces portées sont **statiques** : elles ne dépendent que des positions des `let` définissant `x` et `y`.

Exemple

Considérons le programme OCaml

```
let () =
    let x =
        if read_int () mod 2 = 0 then 10 else 20
    in
    print_int x
```

Même idée ici : la portée de `x` va de la l. 5 à la l. 5 et est statique. Elle ne dépend pas de l'entier saisi sur l'entrée standard.

Exemple

Considérons le programme OCaml

```
let () =
    let x = 10 in
    let y1 = x + 1 in

    let x = 20 in
    let y2 = x + 1 in

    print_int y1;
    print_newline ();
    print_int y2
```

Ici, il y a deux définitions de l'identificateur `x`. La portée du 1er s'étend de la l. 3 à la l. 4. La portée 2e s'étend de la l. 6 à la l. 10.

Le 1er identificateur `x` est **masqué** car un autre identificateur du même nom est déclarée dans la portée qu'il devait avoir initialement.

Ce phénomène s'appelle l'*ombrage de nom*.

Exemple

Considérons les programmes similaires OCaml et Bash

```
let x = 5


let f1 () =
    print_int x

let f2 () =
    let x = 10 in
    f1 ()


let () =
    f1 ();
    f2 ()
```

```
#!/bin/bash
x=5

function f1 {
    echo $x
}

function f2 {
    local x=10
    f1
}

f1
f2
```

Sans surprise, l'exécution du programme de gauche affiche `5` puis `5`. En effet, la définition du `x` dans `f2` n'a pas d'influence sur le `x` de `f1` qui est dans la portée du `x` de la l. 1.

En revanche, l'exécution du programme de droite affiche `5` puis `10`. En effet, le `x` défini en l. 9 à une portée incluant le code de `f1`. La portée de ce `x` est dynamique.

5.5. Pureté fonctionnelle

Il y a plusieurs niveaux de pureté dans les langages fonctionnels.

1. Dans un langage *fonctionnel pur*, la notion d'effet secondaire est inexistante.

   Des difficultés se posent alors notamment lors de la gestion des entrées/sorties. Elles sont résolus élégamment au moyen des **monades**.

2. Dans un langage *fonctionnel impur*, certaines particularités des langages impératifs sont intégrées.

   Cela englobe la gestion classique des entrées/sorties ou le fait d'autoriser certaines données à être mutables.

L'OCaml s'inscrit dans cette dernière catégorie. Nous ferons en effet appel de temps en temps à des fonctions d'impression.

5.6. Caractéristiques de quelques langages usuels

Langage	Vérif. dyn.	Vérif. stat.	Attr. expl.	Attr. impl.	Impératif	Fonctionnel
C	Non	Oui	Oui	Non	Oui	Non
Python	Oui	Non	Non	Oui	Oui	Impur
Java	Non	Oui	Oui	Non	Oui	Non
OCaml	Non	Oui	Oui	Oui	Oui	Impur
Haskell	Non	Oui	Oui	Oui	Non	Pur

6. Bases de la programmation OCaml

6.1. Généralités

Le système d'exploitation conseillé est **Linux** (Manjaro, Debian, *etc.*).

Nous aurons besoin (au minimum) des logiciels suivants :

- un **émulateur de terminal** (XTERM, RXVT, ALACRITTY, KITTY, *etc.*) ;
- un **éditeur de texte** (EMACS, VIM, *etc.*) ;
- une **distribution OCaml** (OPAM fortement conseillé).

Le site officiel du langage

https://ocaml.org

regorge d'informations (pour l'installation, mais aussi pour de la documentation).

Le site

https://ocaml.org/manual/5.1/api/index.html

contient la documentation de la **bibliothèque standard**.

Voir aussi

https://github.com/ocaml-community/awesome-ocaml

pour beaucoup de **ressources à propos du langage**.

L'OCaml est utilisé dans certains projets d'envergure comme

- MirageOS (https://mirage.io), une bibliothèque pour systèmes d'exploitation ;
- Flow (https://flow.org), un vérificateur de types statique pour JavaScript ;
- Hack (https://hacklang.org), un compilateur ;
- Liquidsoap (https://www.liquidsoap.info), un langage pour le streaming multimédia ;
- google-drive-ocamlfuse (https://astrada.github.io/google-drive-ocamlfuse), un système de fichiers FUSE pour Google Drive.

Il est utilisé dans plusieurs entreprises importantes comme

- Meta (https://about.meta.com), une des principales GAFAM ;
- Jane Street (https://www.janestreet.com), une entreprise de services financiers et d'opérations boursières quantitatives ;
- Tezos (https://tezos.com), une blockchain.

Voir https://ocaml.org/industrial-users pour d'autres informations.

En OCaml, un *commentaire* est constitué de tout ce qui est délimité par `(*` et `*)`.

Exemple

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

Les symboles `(*` et `*)` **fonctionnent comme des parenthèses** : il doit être possible d'appareiller chaque `(*` avec un unique `*)`. Les **commentaires imbriqués** sont autorisés.

Exemples

En OCaml, ceci fonctionne :

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

En C, ceci ne fonctionne pas :

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

Il n'existe pas de moyen de commenter une seule ligne en OCaml.

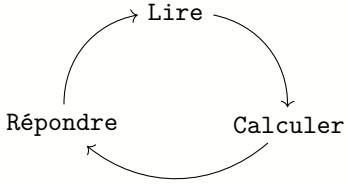
6.2. Interpréteur

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

```
Objective Caml version 5.2.0
...
#
```

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.



Ce cycle est suivi jusqu'à ce que l'utilisateur l'interrompe en saisissant CTRL+D, `exit 0;;`, ou encore `#quit;;`.

Il existe un interpréteur amélioré, `utop`.

Cet interpréteur propose diverses fonctionnalités pratiques comme

- une gestion de l'**historique** (flèche haut et flèche bas) ;
- un accès à l'**autosuggestion des noms** (alias, fonctions, types, *etc.*). Il apparaissent en dessous de la ligne d'entree ;
- une gestion de l'**autocomplétion des noms**. Il est possible de sélectionner un nom suggéré par ALT+GAUCHE et ALT+DROITE et de l'utiliser par ALT+TAB.

Il est possible de **charger un fichier** `File.ml` dans `utop` (ou dans `ocaml`) par

```
#use "File.ml";;
```

(Le symbole `#` fait partie de la commande.) Ceci permet d'avoir accès, dans l'interpréteur, aux entités définies dans `File.ml`.

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

Exemple

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

Une phrase peut tenir sur plusieurs lignes. Les retours à la ligne n'ont pas d'influence sur son sens.

Exemple

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

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

Exemple

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

Explications :

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

6.3. Liaisons

De la même manière que nous pouvons écrire

« *Soit* n *l'entier* 5. »,

pour définir ce que représente le symbole « n » pour avoir le droit de l'utiliser ensuite, il est possible en programmation de donner un nom à une valeur.

Ceci s'appelle une *définition*, ou encore une *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. Il s'agit ici d'une *liaison globale*.

Exemple

```
# 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 lié.

Il est possible de définir des noms **localement** à une expression. La *portée* d'un nom défini localement est étendue à l'expression où figure sa définition. Nous utilisons pour cela la construction syntaxique

```
let ID = EXP1 in EXP2
```

où `ID` est un nom et `EXP1` et `EXP2` sont des expressions. C'est une *liaison locale*.

Très important : cette expression **possède comme valeur** celle de `EXP2` dans laquelle les occurrences libres de `ID` sont remplacées par la valeur de `EXP1`.

Exemples

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

La 2e occurrence de `n` a pour valeur `5` à cause de la liaison locale précédente. Ainsi, `n + 1` a pour valeur `6`, ainsi donc que toute l'expression.

```
# 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.

Ce nom local n'existe plus en dehors, ce qui fait que dans la 3e phrase, le `n` global est considéré.

Pour comprendre ce à quoi fait référence un nom, il est important de visualiser la portée de chaque identificateur défini par une liaison.

Il est alors possible **relier** chaque occurrence d'un nom à sa définition.

Il est très important de tenir compte du phénomène d'**ombrage de nom** dans le cas où un nom `ID` est lié au sein de la portée d'un nom identique `ID`.

Exemples

```
# 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`.

Il est possible de mettre cette sous-expression entre parenthèses pour améliorer la lisibilité.

Voici quelques autres exemples mélangeant liaisons globales et liaisons locales avec des phénomènes d'ombrage de nom.

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.

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 = EXP1 and ID2 = EXP2
```

ou

```
let ID1 = EXP1 and ID2 = EXP2 in EXP3
```

où `ID1` et `ID2` sont des identificateurs, et `EXP1`, `EXP2` et `EXP3` sont des expressions.

Exemples

```
# 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.

Notons l'usage particulier de l'indentation impliqué par les `in` et les `and`.

Cette construction est particulièrement utile pour définir des noms **mutuellement récursifs** (cas exploré plus loin).

Voici quelques exemples qui illustrent quelques particularités des liaisons simultanées.

Exemples

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

Dans la définition locale 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 dans la phrase précédente le `and` par un `in let`.

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

Voici encore deux autres exemples supplémentaires.

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 fait référence à la définition de `x` 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.

Les liaisons `let ID = EXP` et les liaisons locales `let ID = EXP1 in EXP2` sont des entités totalement différentes.

En effet,

- une liaison globale `let ID = EXP` est une phrase qui **n'a pas de valeur**.

  Son but est de **déclarer** un alias `ID` pour la valeur de `EXP` qui sera **visible globalement** dans la suite du programme.

  Ceci **n'est pas une expression** mais une **déclaration**.

Exemple

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

- Une liaison locale `let ID = EXP1 in EXP2` est une phrase qui **a une valeur** et qui est celle de `EXP2` dans laquelle les occurrences de `ID` sont remplacées par la valeur de `EXP1`.

  Ceci **n'est pas une déclaration** mais une **expression**.

Exemple

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

6.4. 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` contient exactement deux valeurs : `true` et `false`.

Voici les *opérations booléennes* :

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

Ces opérateurs produisent des valeurs de type `bool`.

Exemples

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

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

Règle : ne pas hésiter à introduire des parenthèses pour gagner en lisibilité.

Règle (encore meilleure) : apprendre les priorités usuelles entre les opérateurs pour limiter les parenthèses vraiment inutiles.

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

- **décimal**, sans préfixe ;
- **hexadécimal**, avec le préfixe `0x` ;
- **binaire**, avec le préfixe `0b`.

Exemples

- `0`, `1024`, `-82`
- `0x0` (entier 0), `0x400` (entier 1024), `-0xAE00F23` (entier -182456099)
- `0b1011011` (entier 91), `-0b101` (entier -5)

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

Sur un système `64` bits, ceci va de

$$-2^{62} = -4611686018427387904 \qquad \text{à} \qquad 2^{62} - 1 = 4611686018427387903.$$

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

Voici les *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`.

Quelques remarques :

- `succ`, `pred` et `abs` sont des fonctions ;
- `mod` n'est pas une fonction, c'est un opérateur.

Voici les *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`.

Exemples

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

```
# true = (false || false);;
- : bool = false
```

Important : ne jamais utiliser l'opérateur `==` qui teste l'**égalité physique** au lieu de l'**égalité sémantique** (chose que fait `=`). L'opérateur `==` est intéressant dans un contexte où les **données mutables** existent, ainsi que la notion de référence, ce qui n'est pas notre cas.

Remarque : nous verrons que ces opérations fonctionnent aussi sur des valeurs ayant des types différents de `int`.

Voici les *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`.

Exemples

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

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

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

Une valeur de type `float` s'écrit obligatoirement avec une virgule « `.` ».

Exemple

`4.52` est l'écriture du nombre 4.52.

Le zéro à virgule s'écrit `0.0` ou encore `0.`.

Exemples

```
# 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

$$-\texttt{max\_float} = -1.79769313486231571 \times 10^{308} \quad \text{à} \quad \texttt{max\_float} = 1.79769313486231571 \times 10^{308}.$$

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

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

Il est possible de **convertir** un `int` en `float` par la fonction `float_of_int` et un `float` en `int` par la fonction `int_of_float` (réalisant une troncature).

Exemple

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

Exemple

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

Exemples

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

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

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

Exemples

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

```
# 2. *. 3. = 2. *. 2. +. 2. *. 2.;;
- : bool = false
```

Observation : il peut paraître à première vue surprenant que les opérateurs relationnels soient communs aux `int` et aux `float` sans que les opérateurs arithmétiques le soient. Ce point sera élucidé dans la suite.

Ici aussi, il est impossible de mélanger les `int` et les `float` de manière non explicite.

Exemple

Il ne faut pas écrire

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

mais plutôt

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

Il faut en effet demander explicitement la conversion d'un `int` en un `float` pour les utiliser au sein d'un opérateur (`=` ici).

Il existe des opérateurs spécifiques aux expressions de type `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` permet de représenter les caractères ASCII.

Une valeur de type `char` peut s'écrire

- par un caractère entre apostrophes ;
- par son code ASCII, sur trois chiffres en base dix précédés de `\`, le tout entre apostrophes.

Exemples

- `'a'`, `'8'`, `'?'`
- `'\101'`, `'\000'`, `'\035'`

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

Exemple

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

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

Exemples

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

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

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

Une **chaîne de caractères** s'écrit par une suite de caractères entre guillemets.

Exemples

- `"abc123"`,
- `"\100\101f"`

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

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`.

Rappel important : il n'y a pas d'effet secondaire : les chaînes `u` et `v` ne sont **pas modifiées** lors de leur concaténation.

Exemples

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

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

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 comparer des chaînes de caractères.

Les opérateurs de comparaison travaillent selon l'**ordre lexicographique**.

Exemples

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

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

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

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