Bref historique

- Années 70 : développement du langage de preuve de programme ML (Meta Language) (R. Milner).

Bref historique

- Années 70 : développement du langage de preuve de programme ML (Meta Language) (R. Milner).
- 1985 : première implémentation de Caml (Categorical Abstract Machine Language) à l'INRIA (organisme de recherche français).

Bref historique

- Années 70 : développement du langage de preuve de programme ML (Meta Language) (R. Milner).
- 1985 : première implémentation de Caml (Categorical Abstract Machine Language) à l'INRIA (organisme de recherche français).
- 1996 : première version de OCaml (Objective Caml) par X. Leroy.

Bref historique

- Années 70 : développement du langage de preuve de programme ML (Meta Language) (R. Milner).
- 1985 : première implémentation de Caml (Categorical Abstract Machine Language) à l'INRIA (organisme de recherche français).
- 1996 : première version de OCaml (Objective Caml) par X. Leroy.
- 2005 : Première version de F#, variante de OCaml développé par Microsoft.

Bref historique

- Années 70 : développement du langage de preuve de programme ML (Meta Language) (R. Milner).
- 1985 : première implémentation de Caml (Categorical Abstract Machine Language) à l'INRIA (organisme de recherche français).
- 1996 : première version de OCaml (Objective Caml) par X. Leroy.
- 2005 : Première version de F#, variante de OCaml développé par Microsoft.
- 2016 : Première version de Reason, variant de Ocaml développé par Facebook.

Bref historique

- Années 70 : développement du langage de preuve de programme ML (Meta Language) (R. Milner).
- 1985 : première implémentation de Caml (Categorical Abstract Machine Language) à l'INRIA (organisme de recherche français).
- 1996 : première version de OCaml (Objective Caml) par X. Leroy.
- 2005 : Première version de F#, variante de OCaml développé par Microsoft.
- 2016 : Première version de Reason, variant de Ocaml développé par Facebook.
- 2022 : OCaml version 5.0.0

Quelques caractéristiques

- OCaml est un langage de programmation fonctionnel, les fonctions sont au coeur de ce paradigme de programmation.

Quelques caractéristiques

- OCaml est un langage de programmation fonctionnel, les fonctions sont au coeur de ce paradigme de programmation.
- Les variables sont *non modifiables* en conséquence la récursivité est fondamentale car l'écriture de boucle devient impossible. La motivation est de produire un code plus lisible, plus facile à maintenir et moins sujet aux bugs.

Quelques caractéristiques

- OCaml est un langage de programmation fonctionnel, les fonctions sont au coeur de ce paradigme de programmation.
- Les variables sont *non modifiables* en conséquence la récursivité est fondamentale car l'écriture de boucle devient impossible. La motivation est de produire un code plus lisible, plus facile à maintenir et moins sujet aux bugs.
- OCaml est typé statiquement, une variable ne peut pas changer de type au cours de l'exécution. De plus les erreurs de type seront systématiquement détectées à la compilation.

Quelques caractéristiques

- OCaml est un langage de programmation fonctionnel, les fonctions sont au coeur de ce paradigme de programmation.
- Les variables sont *non modifiables* en conséquence la récursivité est fondamentale car l'écriture de boucle devient impossible. La motivation est de produire un code plus lisible, plus facile à maintenir et moins sujet aux bugs.
- OCaml est typé statiquement, une variable ne peut pas changer de type au cours de l'exécution. De plus les erreurs de type seront systématiquement détectées à la compilation.
- Le type des variables n'a pas besoin d'être précisé, il sera automatiquement détecté par le compilateur grâce à un procédé appelé inférence de type.

Quelques caractéristiques

- OCaml est un langage de programmation fonctionnel, les fonctions sont au coeur de ce paradigme de programmation.
- Les variables sont *non modifiables* en conséquence la récursivité est fondamentale car l'écriture de boucle devient impossible. La motivation est de produire un code plus lisible, plus facile à maintenir et moins sujet aux bugs.
- OCaml est typé statiquement, une variable ne peut pas changer de type au cours de l'exécution. De plus les erreurs de type seront systématiquement détectées à la compilation.
- Le type des variables n'a pas besoin d'être précisé, il sera automatiquement détecté par le compilateur grâce à un procédé appelé inférence de type.
- Ocaml est un langage compilé, cependant un environnement interactif `utop` est disponible.

Quelques caractéristiques

- OCaml est un langage de programmation fonctionnel, les fonctions sont au coeur de ce paradigme de programmation.
- Les variables sont *non modifiables* en conséquence la récursivité est fondamentale car l'écriture de boucle devient impossible. La motivation est de produire un code plus lisible, plus facile à maintenir et moins sujet aux bugs.
- OCaml est typé statiquement, une variable ne peut pas changer de type au cours de l'exécution. De plus les erreurs de type seront systématiquement détectées à la compilation.
- Le type des variables n'a pas besoin d'être précisé, il sera automatiquement détecté par le compilateur grâce à un procédé appelé inférence de type.
- Ocaml est un langage compilé, cependant un environnement interactif `utop` est disponible.
- La gestion de la mémoire est automatique (via un ramasse-miettes *garbage collector*).

Fonctions sur les entiers et les flottants

1. Fonction qui pour un entier n, renvoie $n(n-1)$.

```
1  let f n = n * (n-1)
```

Fonctions sur les entiers et les flottants

1. Fonction qui pour un entier n, renvoie $n(n-1)$.

```
1  let f n = n * (n-1)
```

- ⚠ Dans le paradigme fonctionnelle, on écrit des expressions et pas des instructions (paradigme impératif). Une expression est évaluée et renvoie une valeur.

Fonctions sur les entiers et les flottants

1. Fonction qui pour un entier n, renvoie $n(n-1)$.

```
1  let f n = n * (n-1)
```

- ⚠ Dans le paradigme fonctionnelle, on écrit des expressions et pas des instructions (paradigme impératif). Une expression est évaluée et renvoie une valeur.
- On remarquera la proximité avec $n \mapsto n(n-1)$ (et l'absence de `return`).

Fonctions sur les entiers et les flottants

1. Fonction qui pour un entier n, renvoie $n(n-1)$.

```
let f n = n * (n-1)
```

   - ⚠ Dans le paradigme fonctionnelle, on écrit des expressions et pas des instructions (paradigme impératif). Une expression est évaluée et renvoie une valeur.
   - On remarquera la proximité avec $n \mapsto n(n-1)$ (et l'absence de `return`).
   - Les opérateurs `*`, `-` portent sur les entiers et permettent d'inférer le type de l'argument et du résultat.

2. Fonction qui pour un flottant x, renvoie $x^2 - 3x + 7$.

```
let g x = x**2. -. 3.0*.x +. 7.0
```

Fonctions sur les entiers et les flottants

1. Fonction qui pour un entier n, renvoie $n(n-1)$.

```
let f n = n * (n-1)
```

   - ⚠ Dans le paradigme fonctionnelle, on écrit des expressions et pas des instructions (paradigme impératif). Une expression est évaluée et renvoie une valeur.
   - On remarquera la proximité avec $n \mapsto n(n-1)$ (et l'absence de `return`).
   - Les opérateurs `*`, `-` portent sur les entiers et permettent d'inférer le type de l'argument et du résultat.

2. Fonction qui pour un flottant x, renvoie $x^2 - 3x + 7$.

```
let g x = x**2. -. 3.0*.x +. 7.0
```

   - Les opérateurs `+.`, `*.` et `-.` concernent les flottants.

Fonctions sur les entiers et les flottants

1. Fonction qui pour un entier n, renvoie $n(n-1)$.

```
let f n = n * (n-1)
```

   - ⚠ Dans le paradigme fonctionnelle, on écrit des expressions et pas des instructions (paradigme impératif). Une expression est évaluée et renvoie une valeur.
   - On remarquera la proximité avec $n \mapsto n(n-1)$ (et l'absence de `return`).
   - Les opérateurs `*`, `-` portent sur les entiers et permettent d'inférer le type de l'argument et du résultat.

2. Fonction qui pour un flottant x, renvoie $x^2 - 3x + 7$.

```
let g x = x**2. -. 3.0*.x +. 7.0
```

   - Les opérateurs `+.`, `*.` et `-.` concernent les flottants.
   - L'exponentiation est `**` (sur les flottants).

Polymorphisme, booléens, conditionnelle

1. Fonction qui renvoie `true` lorsque deux des trois arguments sont égaux.

```
1  let deux_egaux a b c = (a=b) || (b=c) || (a=c)
```

Polymorphisme, booléens, conditionnelle

1. Fonction qui renvoie `true` lorsque deux des trois arguments sont égaux.

```
1  let deux_egaux a b c = (a=b) || (b=c) || (a=c)
```

   - Les opérateurs logiques sont `&&`, `||` et `not`.

Polymorphisme, booléens, conditionnelle

1. Fonction qui renvoie `true` lorsque deux des trois arguments sont égaux.

```
1  let deux_egaux a b c = (a=b) || (b=c) || (a=c)
```

   - Les opérateurs logiques sont `&&`, `||` et `not`.
   - Ici l'inférence de type ne permet pas de déterminer le type des arguments, on dit que la fonction est polymorphe.

Polymorphisme, booléens, conditionnelle

1. Fonction qui renvoie `true` lorsque deux des trois arguments sont égaux.

```
let deux_egaux a b c = (a=b) || (b=c) || (a=c)
```

   - Les opérateurs logiques sont `&&`, `||` et `not`.
   - Ici l'inférence de type ne permet pas de déterminer le type des arguments, on dit que la fonction est polymorphe.

2. Terme suivant de la suite de syracuse :

```
let syracuse n =
  if n mod 2 = 0 then n/2 else 3*n+1
```

Polymorphisme, booléens, conditionnelle

1. Fonction qui renvoie `true` lorsque deux des trois arguments sont égaux.

```
let deux_egaux a b c = (a=b) || (b=c) || (a=c)
```

   - Les opérateurs logiques sont `&&`, `||` et `not`.
   - Ici l'inférence de type ne permet pas de déterminer le type des arguments, on dit que la fonction est polymorphe.

2. Terme suivant de la suite de syracuse :

```
let syracuse n =
  if n mod 2 = 0 then n/2 else 3*n+1
```

   - On notera la construction `if ...then ...else`.

Polymorphisme, booléens, conditionnelle

1. Fonction qui renvoie `true` lorsque deux des trois arguments sont égaux.

```
let deux_egaux a b c = (a=b) || (b=c) || (a=c)
```

   - Les opérateurs logiques sont `&&`, `||` et `not`.
   - Ici l'inférence de type ne permet pas de déterminer le type des arguments, on dit que la fonction est polymorphe.

2. Terme suivant de la suite de syracuse :

```
let syracuse n =
  if n mod 2 = 0 then n/2 else 3*n+1
```

   - On notera la construction `if ...then ...else`.
   - Attention, le test d'égalité est le `=` simple.

Polymorphisme, booléens, conditionnelle

1. Fonction qui renvoie `true` lorsque deux des trois arguments sont égaux.

```
let deux_egaux a b c = (a=b) || (b=c) || (a=c)
```

   - Les opérateurs logiques sont `&&`, `||` et `not`.
   - Ici l'inférence de type ne permet pas de déterminer le type des arguments, on dit que la fonction est polymorphe.

2. Terme suivant de la suite de syracuse :

```
let syracuse n =
  if n mod 2 = 0 then n/2 else 3*n+1
```

   - On notera la construction `if ...then ...else`.
   - Attention, le test d'égalité est le `=` simple.
   - Le modulo s'obtient avec `mod`.

Récursivité

1. Fonction qui calcule la somme des n premiers carrés.

```
1  let rec somme_carre n =
2    if n=0 then 0 else n*n + somme_carre (n-1)
```

Récursivité

1. Fonction qui calcule la somme des n premiers carrés.

```
1  let rec somme_carre n =
2    if n=0 then 0 else n*n + somme_carre (n-1)
```

   - On notera la mot clé `rec` pour préciser que la fonction est récursive.

Récursivité

1. Fonction qui calcule la somme des n premiers carrés.

```
1  let rec somme_carre n =
2    if n=0 then 0 else n*n + somme_carre (n-1)
```

- On notera la mot clé `rec` pour préciser que la fonction est récursive.
- Les parenthèses autour de `n-1` permettent d'éviter la confusion avec `(somme carre n)-1` (et donc d'avoir une récursion infinie)

Récursivité

1. Fonction qui calcule la somme des n premiers carrés.

```
let rec somme_carre n =
  if n=0 then 0 else n*n + somme_carre (n-1)
```

   - On notera la mot clé `rec` pour préciser que la fonction est récursive.
   - Les parenthèses autour de `n-1` permettent d'éviter la confusion avec `(somme carre n)-1` (et donc d'avoir une récursion infinie)

2. Fonction qui compte à rebours depuis n et affiche "Partez" !

```
let rec compte_rebours n =
  if n=0 then print_endline "Partez" else (
    print_int n;
    print_endline "";
    compte_rebours (n-1) )
```

Récursivité

1. Fonction qui calcule la somme des n premiers carrés.

```
let rec somme_carre n =
  if n=0 then 0 else n*n + somme_carre (n-1)
```

   - On notera la mot clé `rec` pour préciser que la fonction est récursive.
   - Les parenthèses autour de `n-1` permettent d'éviter la confusion avec `(somme carre n)-1` (et donc d'avoir une récursion infinie)

2. Fonction qui compte à rebours depuis n et affiche "Partez" !

```
let rec compte_rebours n =
  if n=0 then print_endline "Partez" else (
    print_int n;
    print_endline "";
    compte_rebours (n-1) )
```

   - Regroupement de la clause du `else` dans un bloc.

Récursivité

1. Fonction qui calcule la somme des n premiers carrés.

```
let rec somme_carre n =
  if n=0 then 0 else n*n + somme_carre (n-1)
```

   - On notera la mot clé `rec` pour préciser que la fonction est récursive.
   - Les parenthèses autour de `n-1` permettent d'éviter la confusion avec `(somme carre n)-1` (et donc d'avoir une récursion infinie)

2. Fonction qui compte à rebours depuis n et affiche "Partez" !

```
let rec compte_rebours n =
  if n=0 then print_endline "Partez" else (
    print_int n;
    print_endline "";
    compte_rebours (n-1) )
```

   - Regroupement de la clause du `else` dans un bloc.
   - Les résultats des affichages sont de type `unit` (et on tous la même valeur : `()`).

Récursivité

1. Fonction qui calcule la somme des n premiers carrés.

```
let rec somme_carre n =
  if n=0 then 0 else n*n + somme_carre (n-1)
```

- On notera la mot clé `rec` pour préciser que la fonction est récursive.
- Les parenthèses autour de `n-1` permettent d'éviter la confusion avec `(somme carre n)-1` (et donc d'avoir une récursion infinie)

2. Fonction qui compte à rebours depuis n et affiche "Partez" !

```
let rec compte_rebours n =
  if n=0 then print_endline "Partez" else (
    print_int n;
    print_endline "";
    compte_rebours (n-1) )
```

- Regroupement de la clause du `else` dans un bloc.
- Les résultats des affichages sont de type `unit` (et on tous la même valeur : `()`).
- Les `;` séparent les différents affichages, et ignorent la valeur renvoyée.

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `/`, `mod`, `abs`	Entiers signés sur 64 bits valeurs dans $[\![-2^{62}; 2^{62}-1]\!]$

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `/`, `mod`, `abs`	Entiers signés sur 64 bits valeurs dans $[\![-2^{62}; 2^{62}-1]\!]$
`float`	`+.`, `-.`, `*.`, `/.`, `**`	Correspond au type double de la norme IEEE-754. Fonctions mathématiques usuelles ($\sin, \cos, \exp, \ldots$)

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `/`, `mod`, `abs`	Entiers signés sur 64 bits valeurs dans $[\![-2^{62}; 2^{62}-1]\!]$
`float`	`+.`, `-.`, `*.`, `/.`, `**`	Correspond au type double de la norme IEEE-754. Fonctions mathématiques usuelles ($\sin, \cos, \exp, \ldots$)
`bool`	`&&`, `||`, `not`	Evaluations paresseuses.

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `/`, `mod`, `abs`	Entiers signés sur 64 bits valeurs dans $[\![-2^{62}; 2^{62}-1]\!]$
`float`	`+.`, `-.`, `*.`, `/.`, `**`	Correspond au type double de la norme IEEE-754. Fonctions mathématiques usuelles ($\sin, \cos, \exp, \ldots$)
`bool`	`&&`, `||`, `not`	Evaluations paresseuses.
`char`		Se note entre apostrophe (`'`). Les `char` sont comparables (ordre du code ASCII).

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `/`, `mod`, `abs`	Entiers signés sur 64 bits valeurs dans $[\![-2^{62}; 2^{62}-1]\!]$
`float`	`+.`, `-.`, `*.`, `/.`, `**`	Correspond au type double de la norme IEEE-754. Fonctions mathématiques usuelles ($\sin, \cos, \exp, \ldots$)
`bool`	`&&`, `||`, `not`	Evaluations paresseuses.
`char`		Se note entre apostrophe (`'`). Les `char` sont comparables (ordre du code ASCII).
`string`	`^`, `.[]`, `String.length`	Immutabilité. Concaténation de deux chaines : `"Bon"^"jour"`. Accès au ième avec `.[i]`

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `/`, `mod`, `abs`	Entiers signés sur 64 bits valeurs dans $[\![-2^{62}; 2^{62}-1]\!]$
`float`	`+.`, `-.`, `*.`, `/.`, `**`	Correspond au type double de la norme IEEE-754. Fonctions mathématiques usuelles $(\sin, \cos, \exp, \dots)$
`bool`	`&&`, `||`, `not`	Evaluations paresseuses.
`char`		Se note entre apostrophe (`'`). Les `char` sont comparables (ordre du code ASCII).
`string`	`^`, `.[]`, `String.length`	Immutabilité. Concaténation de deux chaines : `"Bon"^"jour"`. Accès au ième avec `.[i]`

- Le type `unit` possède une seule valeur notée `()` à rapprocher du type `void` du C. Un affichage renvoie `()`.

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `/`, `mod`, `abs`	Entiers signés sur 64 bits valeurs dans $[\![-2^{62}; 2^{62}-1]\!]$
`float`	`+.`, `-.`, `*.`, `/.`, `**`	Correspond au type double de la norme IEEE-754. Fonctions mathématiques usuelles $(\sin, \cos, \exp, \dots)$
`bool`	`&&`, `||`, `not`	Evaluations paresseuses.
`char`		Se note entre apostrophe (`'`). Les `char` sont comparables (ordre du code ASCII).
`string`	`^`, `.[]`, `String.length`	Immutabilité. Concaténation de deux chaines : `"Bon"^"jour"`. Accès au ième avec `.[i]`

- Le type `unit` possède une seule valeur notée `()` à rapprocher du type `void` du C. Un affichage renvoie `()`.
- Les opérateurs de comparaison (`=`, `<>`, `>`, `>=`, `<`, `<=`) sont polymorphes mais s'appliquent à deux objets *de même type*.

Exemple

1. Ecrire une fonction `abs_entier` qui renvoie la valeur absolue de l'entier donné en argument .

Exemple

1. Ecrire une fonction `abs_entier` qui renvoie la valeur absolue de l'entier donné en argument .

```
1  let abs_entier n =
2    if n <0 then -n else n
```

Exemple

1. Ecrire une fonction `abs_entier` qui renvoie la valeur absolue de l'entier donné en argument .

```
1  let abs_entier n =
2    if n <0 then -n else n
```

2. Ecrire une fonction `abs_flottant` qui renvoie la valeur absolue du flottant donné en argument .

Exemple

1. Ecrire une fonction `abs_entier` qui renvoie la valeur absolue de l'entier donné en argument .

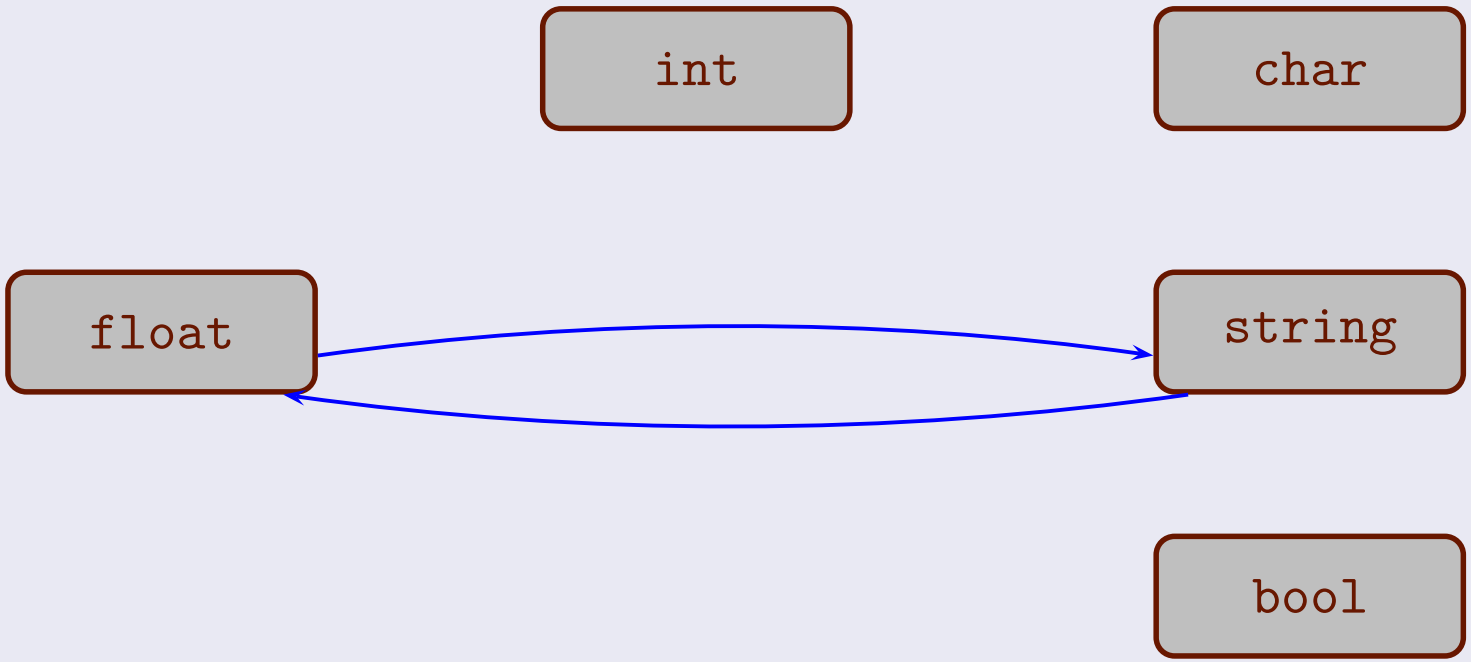
```
let abs_entier n =
  if n <0 then -n else n
```

2. Ecrire une fonction `abs_flottant` qui renvoie la valeur absolue du flottant donné en argument .

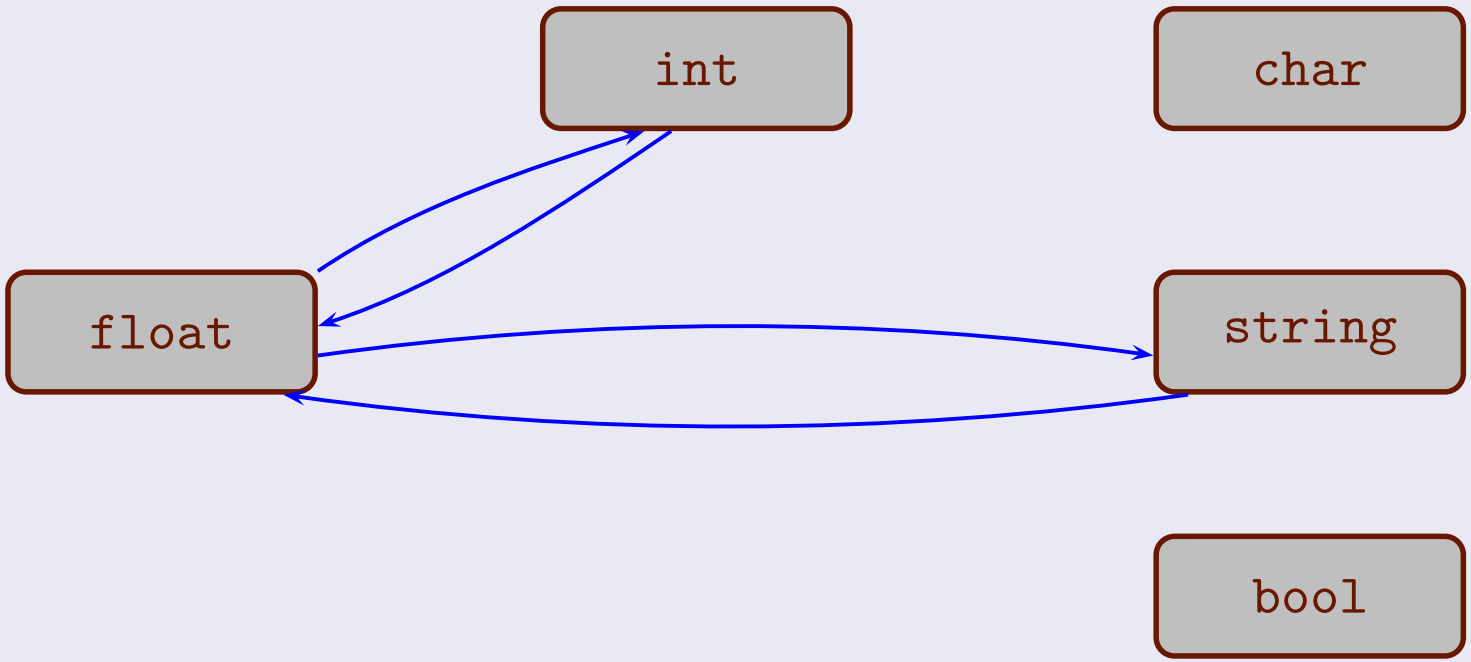
```
let abs_flottant n =
  if n < 0. then -.n else n
```

Conversion de types

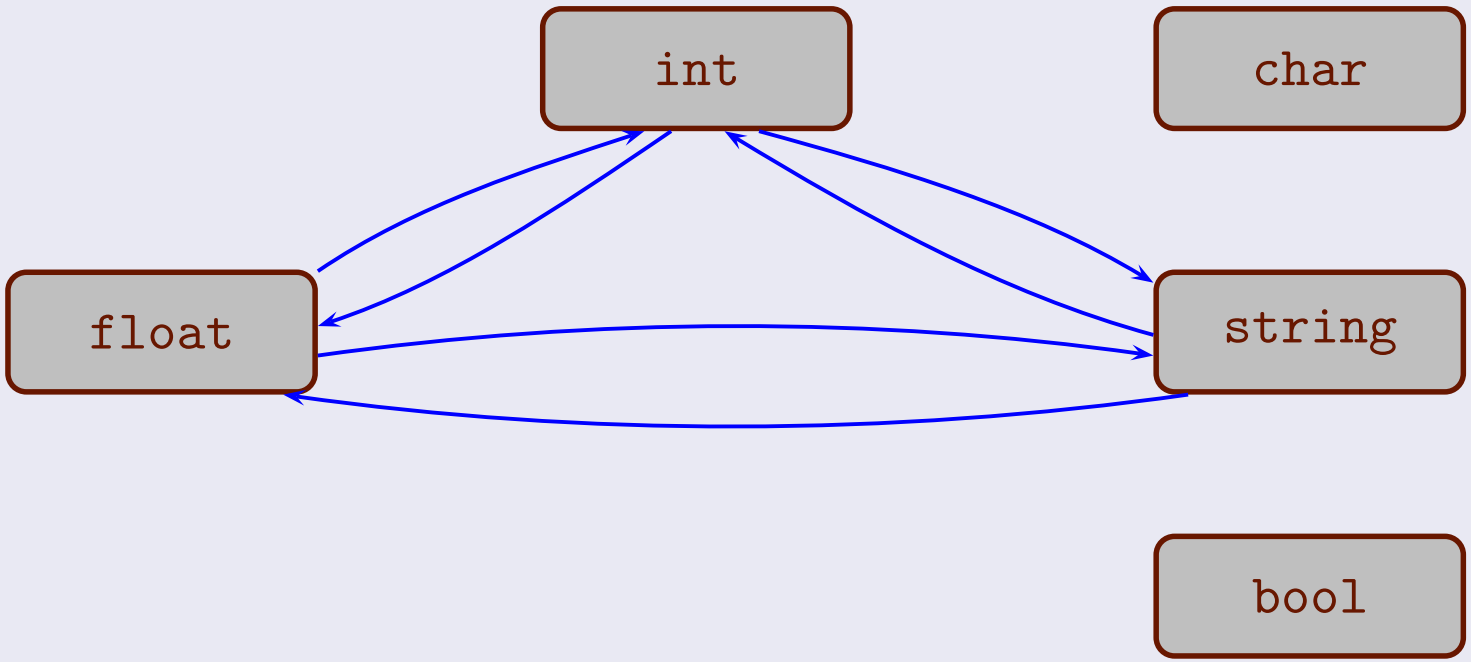
Conversion de types



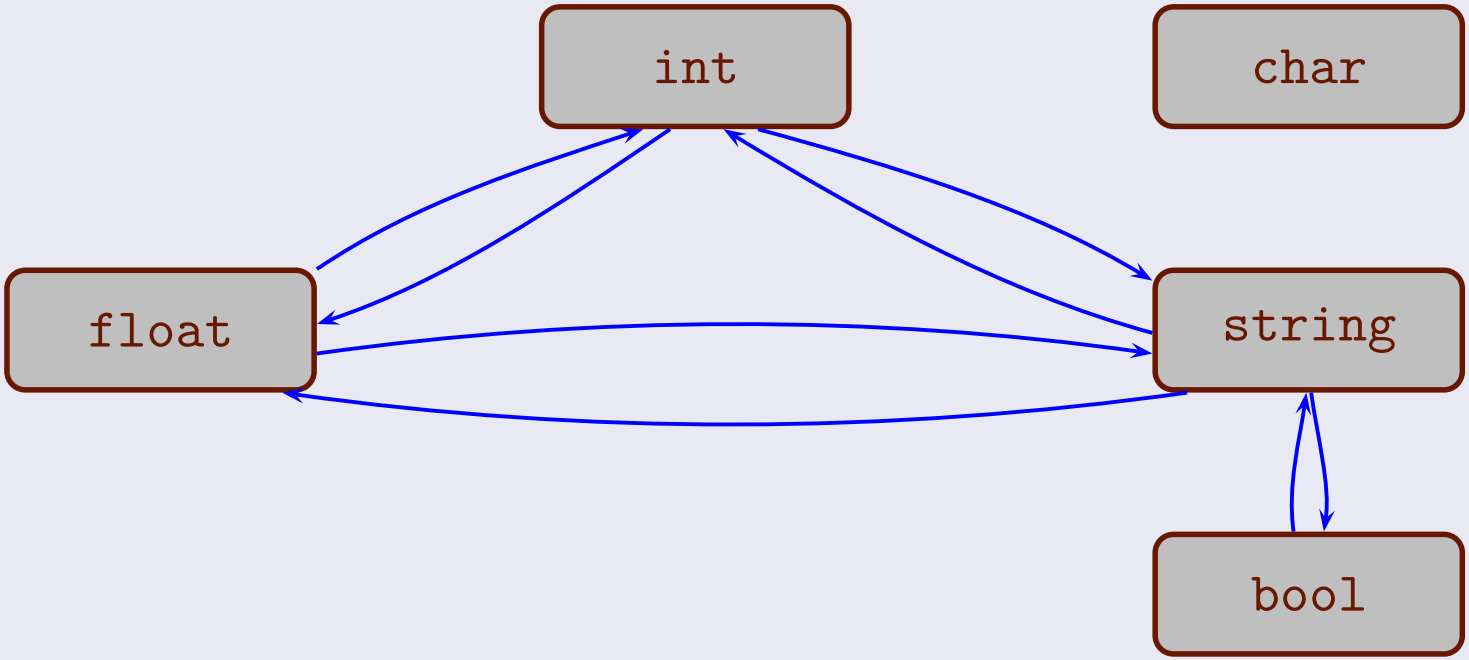
Conversion de types



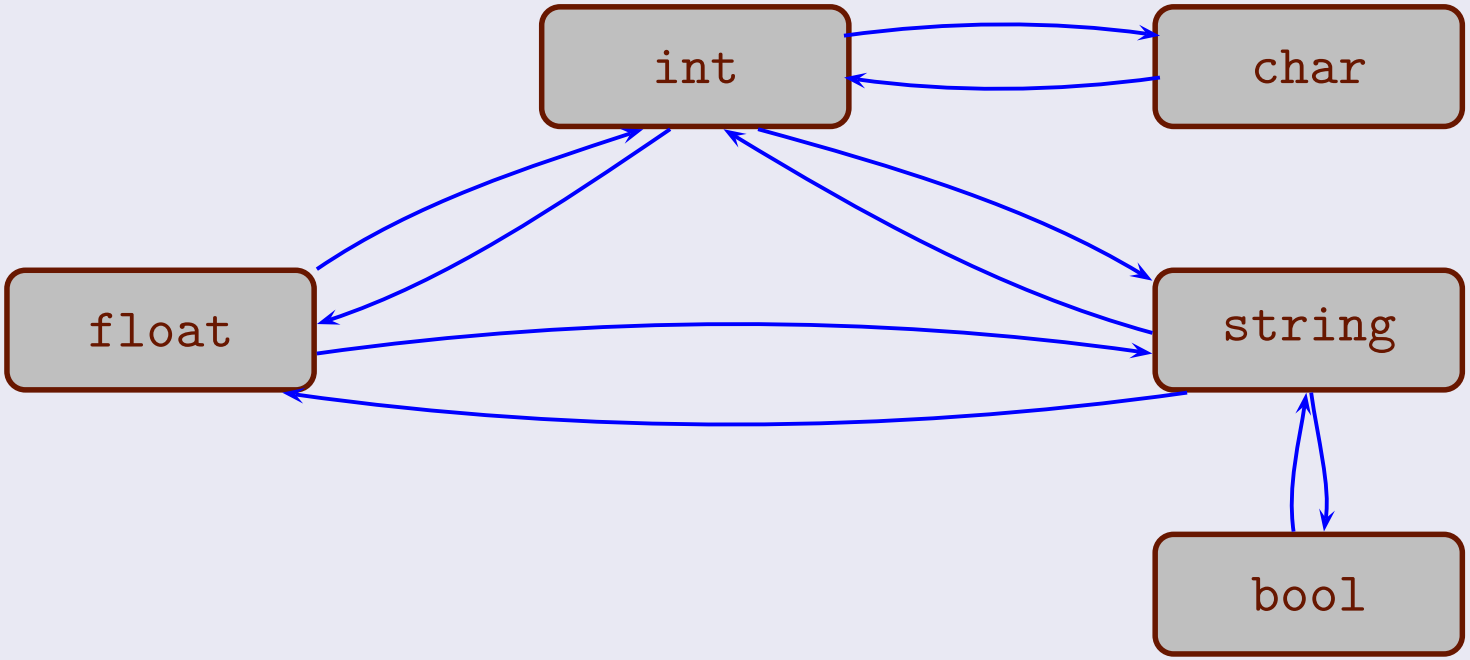
Conversion de types



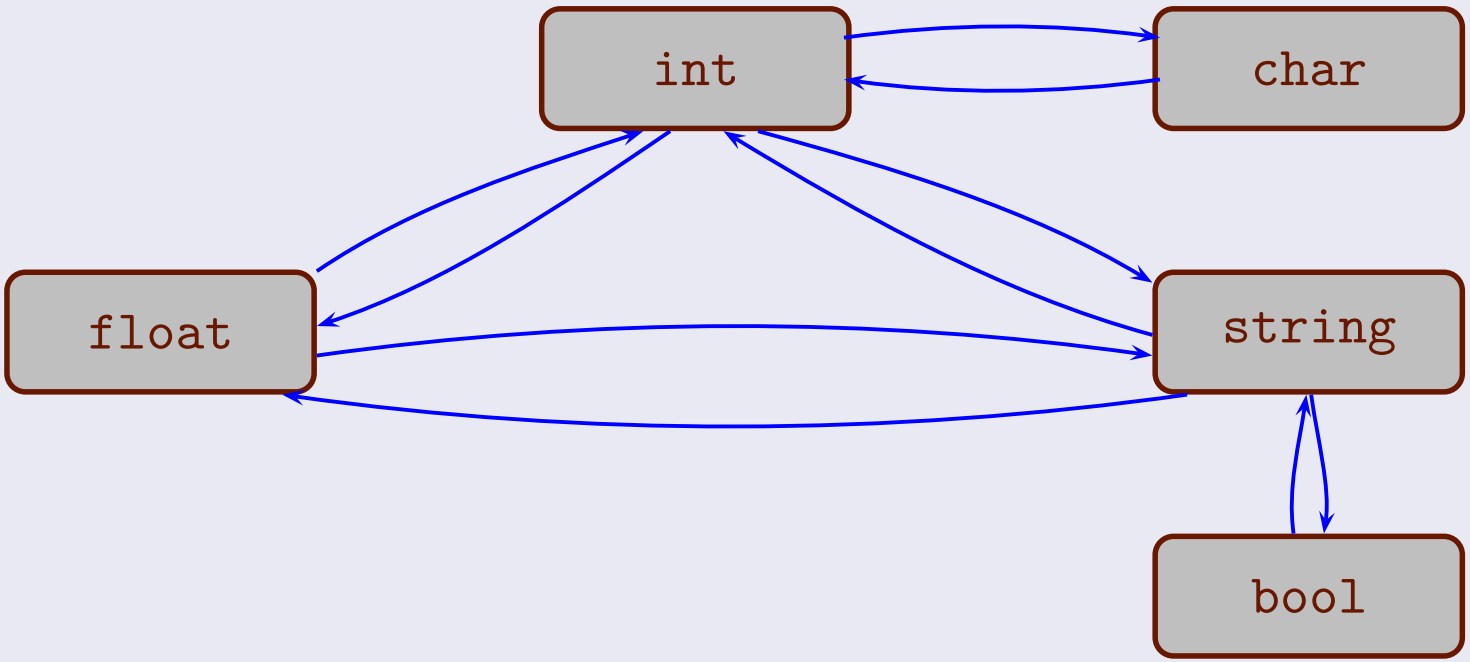
Conversion de types



Conversion de types

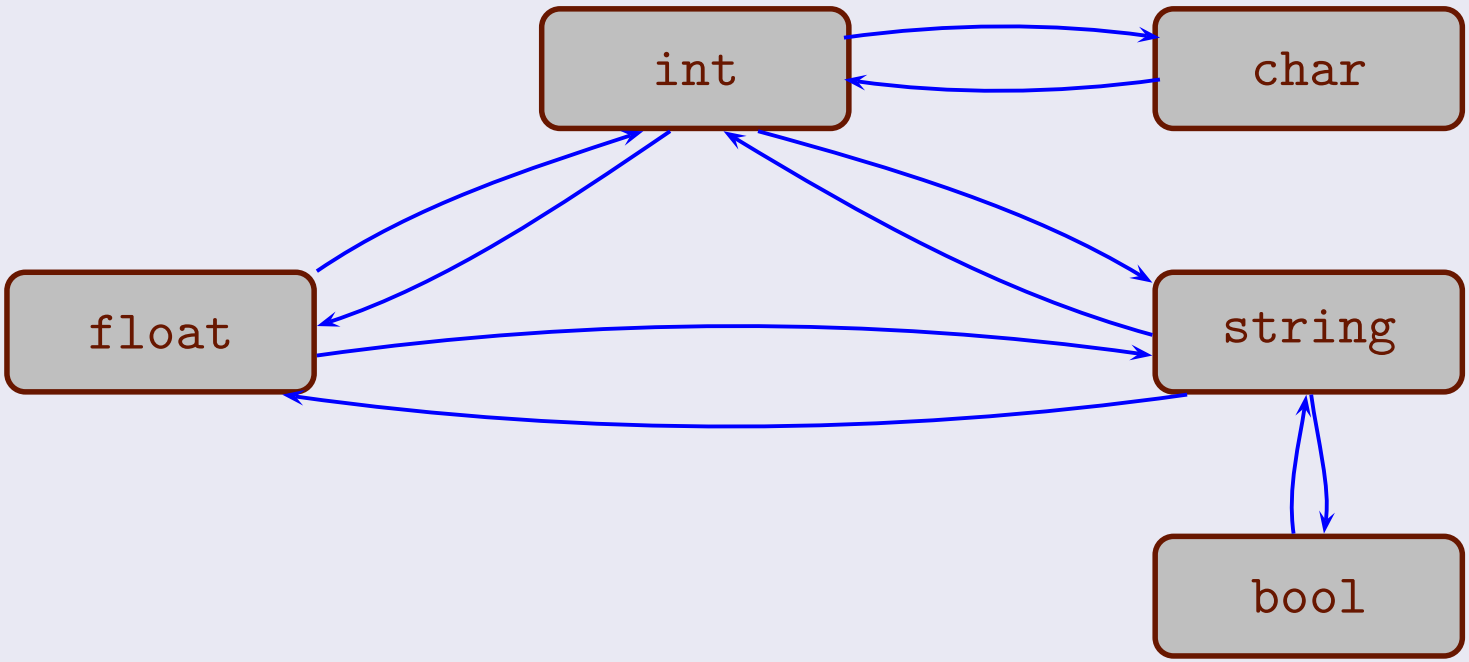


Conversion de types



- Les fonctions de conversion sont de la forme `<type1>_of_<type2>` par exemple, `string_of_float`.

Conversion de types



- Les fonctions de conversion sont de la forme `<type1>_of_<type2>` par exemple, `string_of_float`.
- L'affichage s'obtient avec `print_<type>` par exemple `print_string` (excepté booléen).