

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

C6 OCaml : aspects fonctionnels

1. Généralités

Quelques caractéristiques

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

C6 OCaml : aspects fonctionnels

1. Généralités

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 **récurtivité** 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.

C6 OCaml : aspects fonctionnels

1. Généralités

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 **récurtivité** 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.

C6 OCaml : aspects fonctionnels

1. Généralités

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 **récurtivité** 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**.

C6 OCaml : aspects fonctionnels

1. Généralités

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 **récurtivité** 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.

C6 OCaml : aspects fonctionnels

1. Généralités

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 **récurtivité** 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

- ④ Fonction qui pour un entier n , renvoie $n(n - 1)$.

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

Fonctions sur les entiers et les flottants

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

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

- ⚠ Dans le paradigme fonctionnel, on écrit des **expressions** et pas des **instructions** (paradigme impératif). Contrairement à une instruction, une expression est évaluée et renvoie toujours une valeur.

Fonctions sur les entiers et les flottants

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

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

- ⚠ Dans le paradigme fonctionnel, on écrit des **expressions** et pas des **instructions** (paradigme impératif). Contrairement à une instruction, une expression est évaluée et renvoie toujours une valeur.
- On remarquera la proximité avec $f : n \mapsto n(n - 1)$ (et l'absence de **return**).

Fonctions sur les entiers et les flottants

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

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

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

Fonctions sur les entiers et les flottants

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

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

- ⚠ Dans le paradigme fonctionnel, on écrit des **expressions** et pas des **instructions** (paradigme impératif). Contrairement à une instruction, une expression est évaluée et renvoie toujours une valeur.
- On remarquera la proximité avec $f : n \mapsto n(n - 1)$ (et l'absence de **return**).
- Les opérateurs *****, **-** portent sur les entiers et permettent d'inférer le type de l'argument et du résultat.
- Pour calculer $f(5)$: **f 5 ;;**

C6 OCaml : aspects fonctionnels

2. Quelques exemples

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 fonctionnel, on écrit des **expressions** et pas des **instructions** (paradigme impératif). Contrairement à une instruction, une expression est évaluée et renvoie toujours une valeur.
- On remarquera la proximité avec $f : n \mapsto n(n - 1)$ (et l'absence de **return**).
- Les opérateurs $*$, $-$ portent sur les entiers et permettent d'inférer le type de l'argument et du résultat.
- Pour calculer $f(5)$: **f 5 ;;**

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

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

C6 OCaml : aspects fonctionnels

2. Quelques exemples

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 fonctionnel, on écrit des **expressions** et pas des **instructions** (paradigme impératif). Contrairement à une instruction, une expression est évaluée et renvoie toujours une valeur.
- On remarquera la proximité avec $f : n \mapsto n(n - 1)$ (et l'absence de **return**).
- Les opérateurs $*$, $-$ portent sur les entiers et permettent d'inférer le type de l'argument et du résultat.
- Pour calculer $f(5) : f\ 5 ;;$

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

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

- Les opérateurs $+. , *. , -.$ concernent les flottants.

C6 OCaml : aspects fonctionnels

2. Quelques exemples

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 fonctionnel, on écrit des **expressions** et pas des **instructions** (paradigme impératif). Contrairement à une instruction, une expression est évaluée et renvoie toujours une valeur.
- On remarquera la proximité avec $f : n \mapsto n(n - 1)$ (et l'absence de **return**).
- Les opérateurs $*$, $-$ portent sur les entiers et permettent d'inférer le type de l'argument et du résultat.
- Pour calculer $f(5) : \text{f } 5 ;;$

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

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

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

C6 OCaml : aspects fonctionnels

2. Quelques exemples

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);;
```

C6 OCaml : aspects fonctionnels

2. Quelques exemples

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.

```
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**.
- L'appel s'effectue en donnant les paramètres séparés par des espaces :
`deux_egaux 4 6 4;;`

C6 OCaml : aspects fonctionnels

2. Quelques exemples

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**.
- L'appel s'effectue en donnant les paramètres séparés par des espaces :
`deux_egaux 4 6 4;;`

- ② Terme suivant de la suite de Syracuse :

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

C6 OCaml : aspects fonctionnels

2. Quelques exemples

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**.
- L'appel s'effectue en donnant les paramètres séparés par des espaces :
`deux_egaux 4 6 4;;`

- ② Terme suivant de la suite de Syracuse :

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

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

C6 OCaml : aspects fonctionnels

2. Quelques exemples

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**.
- L'appel s'effectue en donnant les paramètres séparés par des espaces :
`deux_egaux 4 6 4;;`

- ② Terme suivant de la suite de syracuse :

```
1 let syracuse n =  
2   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.

C6 OCaml : aspects fonctionnels

2. Quelques exemples

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**.
- L'appel s'effectue en donnant les paramètres séparés par des espaces :
`deux_egaux 4 6 4;;`

- ② Terme suivant de la suite de syracuse :

```
1 let syracuse n =  
2   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`.

Récurivité exemple 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);;
```

Récurtivité exemple 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 réursive.

Récurtivité exemple 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)

Réversivité exemple 2

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

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


Récursivité exemple 2

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

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

- Regroupement de la clause du `else` dans un bloc délimité par `()`.
On peut de façon équivalente délimiter par `begin` et `end`

Récursivité exemple 2

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

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

- Regroupement de la clause du `else` dans un bloc délimité par `()`.
On peut de façon équivalente délimiter par `begin` et `end`
- Les résultats des affichages, sont aussi des expressions et donc renvoient une valeur. Ce sont des valeurs de type `unit` (et on tous la même valeur : `()`).

Récursivité exemple 2

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

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

- Regroupement de la clause du `else` dans un bloc délimité par `()`.
On peut de façon équivalente délimiter par `begin` et `end`
- Les résultats des affichages, sont aussi des expressions et donc renvoient une valeur. Ce sont des valeurs de type `unit` (et on tous la même valeur : `()`).
- Les `;` séparent les différents affichages, ils permettent d'ignorer la valeur renvoyée par les différents affichages. L'évaluation de l'expression se poursuit donc jusqu'à l'appel récursif.

C6 OCaml : aspects fonctionnels

2. Quelques exemples



A retenir !

- Un programme OCaml consiste en l'évaluation d'une ou plusieurs expressions, **il n'y a pas d'instructions.**

C6 OCaml : aspects fonctionnels

2. Quelques exemples

A retenir !

- Un programme OCaml consiste en l'évaluation d'une ou plusieurs expressions, **il n'y a pas d'instructions.**

Une structure `if ... then ... else` est évaluée et renvoie une expression. En impératif, ce sont des instructions conditionnelles.

⚠ A retenir !

- Un programme OCaml consiste en l'évaluation d'une ou plusieurs expressions, **il n'y a pas d'instructions.**

Une structure `if ... then ... else` est évaluée et renvoie une expression. En impératif, ce sont des instructions conditionnelles.

- Les types n'ont pas à être spécifiés, ils sont déterminés automatiquement par le mécanisme d'inférence.

Les opérateurs sont donc associés à des types précis, par exemple `+` est l'addition sur les entiers, `+.` est l'addition sur les flottants, `@` est l'opérateur de concaténation des listes, `^` est l'opérateur de concaténation des chaînes de caractères.

C6 OCaml : aspects fonctionnels

2. Quelques exemples



A retenir !

- Un programme OCaml consiste en l'évaluation d'une ou plusieurs expressions, **il n'y a pas d'instructions.**

Une structure `if ... then ... else` est évaluée et renvoie une expression. En impératif, ce sont des instructions conditionnelles.

- Les types n'ont pas à être spécifiés, ils sont déterminés automatiquement par le mécanisme d'inférence.

Les opérateurs sont donc associés à des types précis, par exemple `+` est l'addition sur les entiers, `+.` est l'addition sur les flottants, `@` est l'opérateur de concaténation des listes, `^` est l'opérateur de concaténation des chaînes de caractères.

- Toute expression a une valeur, la détermination de cette valeur s'appelle l'évaluation.

C6 OCaml : aspects fonctionnels

3. Définitions et types de base

Types de base

Type	Opérations	Commentaires

C6 OCaml : aspects fonctionnels

3. Définitions et types de base

Types de base

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

C6 OCaml : aspects fonctionnels

3. Définitions et types de base

Types de base

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

C6 OCaml : aspects fonctionnels

3. Définitions et types de base

Types de base

Type	Opérations	Commentaires
<code>int</code>	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>mod</code> , <code>abs</code>	Entiers signés sur 64 bits valeurs dans $\llbracket -2^{62}; 2^{62} - 1 \rrbracket$
<code>float</code>	<code>+. , -. , *. , /. , **.</code>	Correspond au type double de la norme IEEE-754. Fonctions mathématiques usuelles (<code>sin</code> , <code>exp</code> , ...)
<code>bool</code>	<code>&&</code> , <code> </code> , <code>not</code>	Evaluations paresseuses. Les valeurs sont notées <code>true</code> et <code>false</code> .

C6 OCaml : aspects fonctionnels

3. Définitions et types de base

Types de base

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

C6 OCaml : aspects fonctionnels

3. Définitions et types de base

Types de base

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

C6 OCaml : aspects fonctionnels

3. Définitions et types de base

Types de base

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

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

C6 OCaml : aspects fonctionnels

3. Définitions et types de base

Types de base

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

- 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*.

C6 OCaml : aspects fonctionnels

3. Définitions et types de base

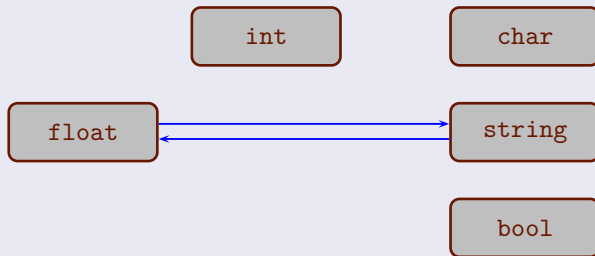
Conversion de types



C6 OCaml : aspects fonctionnels

3. Définitions et types de base

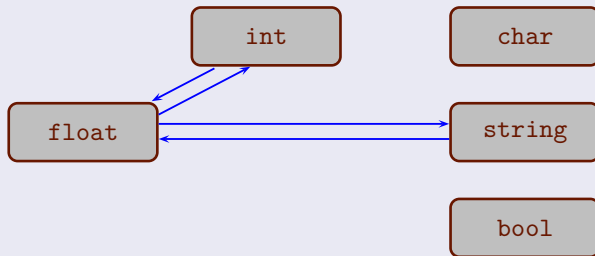
Conversion de types



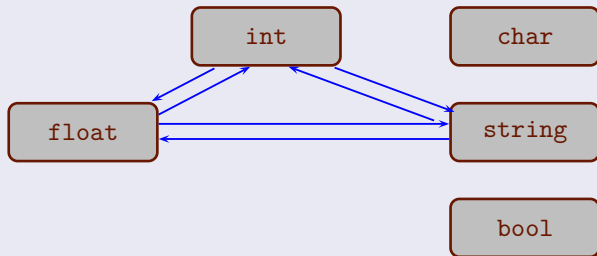
C6 OCaml : aspects fonctionnels

3. Définitions et types de base

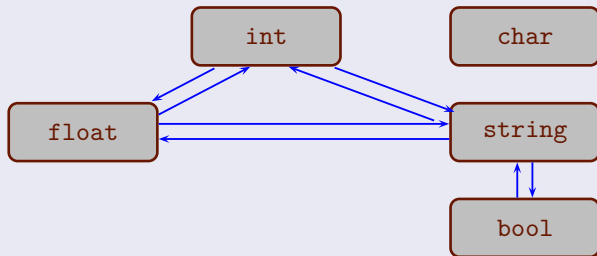
Conversion de types



Conversion de types



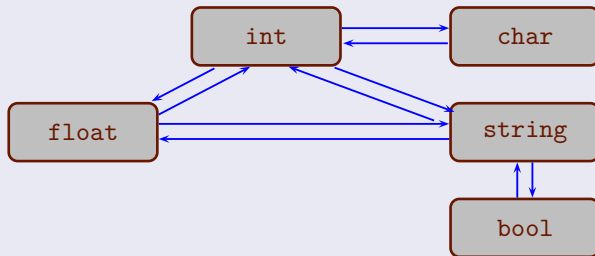
Conversion de types



C6 OCaml : aspects fonctionnels

3. Définitions et types de base

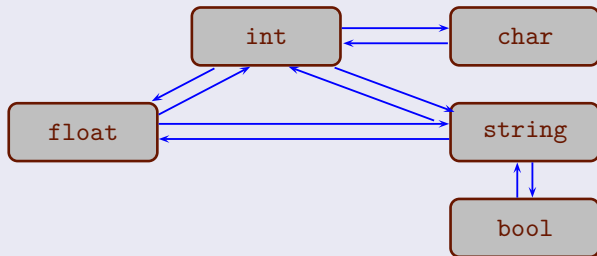
Conversion de types



C6 OCaml : aspects fonctionnels

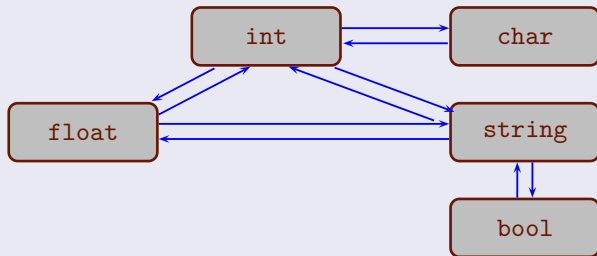
3. Définitions et types de base

Conversion de types



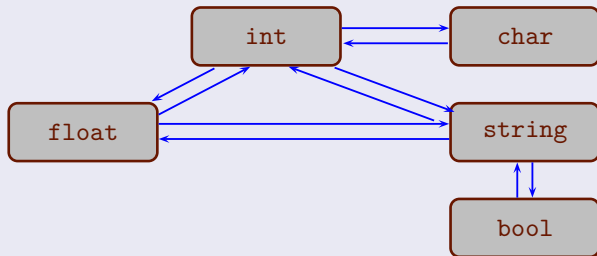
- Les fonctions de conversion sont de la forme `<type1>_of_<type2>` par exemple, `string_of_float`.

Conversion de types



- Les fonctions de conversion sont de la forme `<type1>_of_<type2>` par exemple, `string_of_float`.
- Il n'y a *pas* de conversion implicite.

Conversion de types



- Les fonctions de conversion sont de la forme `<type1>_of_<type2>` par exemple, `string_of_float`.
- Il n'y a *pas* de conversion implicite.
- L'affichage s'obtient avec `print_<type>` par exemple `print_string` (excepté pour les booléens).

Syntaxe et évaluation

- En Ocaml, on parlera d'**expressions conditionnelles** (et pas d'instructions conditionnelles).

Syntaxe et évaluation

- En Ocaml, on parlera d'**expressions conditionnelles** (et pas d'instructions conditionnelles).
- La syntaxe d'une expression conditionnelle est :

```
if expr_bool then expr1 else expr2;;
```

C6 OCaml : aspects fonctionnels

4. Expressions conditionnelles

Syntaxe et évaluation

- En Ocaml, on parlera d'**expressions conditionnelles** (et pas d'instructions conditionnelles).
- La syntaxe d'une expression conditionnelle est :

```
if expr_bool then expr1 else expr2;;
```

- Cette expression est évaluée de la façon suivante :

Syntaxe et évaluation

- En Ocaml, on parlera d'**expressions conditionnelles** (et pas d'instructions conditionnelles).
- La syntaxe d'une expression conditionnelle est :

```
if expr_bool then expr1 else expr2;;
```

- Cette expression est évaluée de la façon suivante :
 - On évalue `expr_bool`

Syntaxe et évaluation

- En Ocaml, on parlera d'**expressions conditionnelles** (et pas d'instructions conditionnelles).

- La syntaxe d'une expression conditionnelle est :

```
if expr_bool then expr1 else expr2;;
```

- Cette expression est évaluée de la façon suivante :
 - On évalue `expr_bool`
 - Si le résultat est vrai alors la valeur de l'expression conditionnelle est `expr1`

Syntaxe et évaluation

- En Ocaml, on parlera d'**expressions conditionnelles** (et pas d'instructions conditionnelles).

- La syntaxe d'une expression conditionnelle est :

```
if expr_bool then expr1 else expr2;;
```

- Cette expression est évaluée de la façon suivante :
 - On évalue `expr_bool`
 - Si le résultat est vrai alors la valeur de l'expression conditionnelle est `expr1`
 - Sinon c'est `expr2`

Syntaxe et évaluation

- En Ocaml, on parlera d'**expressions conditionnelles** (et pas d'instructions conditionnelles).

- La syntaxe d'une expression conditionnelle est :

```
if expr_bool then expr1 else expr2;;
```

- Cette expression est évaluée de la façon suivante :
 - On évalue `expr_bool`
 - Si le résultat est vrai alors la valeur de l'expression conditionnelle est `expr1`
 - Sinon c'est `expr2`
- `expr1` et `expr2` doivent toujours être du même type.

Syntaxe et évaluation

- En Ocaml, on parlera d'**expressions conditionnelles** (et pas d'instructions conditionnelles).

- La syntaxe d'une expression conditionnelle est :

```
if expr_bool then expr1 else expr2;;
```

- Cette expression est évaluée de la façon suivante :
 - On évalue `expr_bool`
 - Si le résultat est vrai alors la valeur de l'expression conditionnelle est `expr1`
 - Sinon c'est `expr2`
- `expr1` et `expr2` doivent toujours être du même type.
- On doit donc toujours écrire la clause `else`

Syntaxe et évaluation

- En Ocaml, on parlera d'**expressions conditionnelles** (et pas d'instructions conditionnelles).
- La syntaxe d'une expression conditionnelle est :

```
if expr_bool then expr1 else expr2;;
```

- Cette expression est évaluée de la façon suivante :
 - On évalue `expr_bool`
 - Si le résultat est vrai alors la valeur de l'expression conditionnelle est `expr1`
 - Sinon c'est `expr2`
- `expr1` et `expr2` doivent toujours être du même type.
- On doit donc toujours écrire la clause `else`
Sauf en fait dans le cas où `expr1` est de type `unit`, dans ce cas en cas d'omission, `expr2` est par défaut `()` (seule valeur du type `unit`)

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 .

C6 OCaml : aspects fonctionnels

4. Expressions conditionnelles

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 .

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

C6 OCaml : aspects fonctionnels

4. Expressions conditionnelles

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 .

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

- ③ Evaluer l'expression suivante et donner la valeur de b si a vaut -3

```
let b = 3 * if a < 0 then 2 else 5;;
```

C6 OCaml : aspects fonctionnels

4. Expressions conditionnelles

Exemple

- ❶ 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;;
```

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

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

- ❸ Evaluer l'expression suivante et donner la valeur de b si a vaut -3

```
let b = 3 * if a < 0 then 2 else 5;;  
la valeur de b est 6.
```

C6 OCaml : aspects fonctionnels

5. Structure d'un programme OCaml

Structure générale d'un programme OCaml

- Un programme OCaml repose sur l'évaluation d'une ou plusieurs expressions :
 expr1 ;;
 expr2 ;;
 ...

C6 OCaml : aspects fonctionnels

5. Structure d'un programme OCaml

Structure générale d'un programme OCaml

- Un programme OCaml repose sur l'évaluation d'une ou plusieurs expressions :
 `expr1 ;;`
 `expr2 ;;`
 ...
- Les expressions sont évaluées dans un **environnement** (qui lie des identificateurs à des valeurs)
 `a + b` est une expression qui sera évaluée à 10 si `a` est lié à la valeur 4 et `b` à la valeur 6.

C6 OCaml : aspects fonctionnels

5. Structure d'un programme Ocaml

Structure générale d'un programme OCaml

- Un programme OCaml repose sur l'évaluation d'une ou plusieurs expressions :
`expr1 ;;`
`expr2 ;;`
`...`
- Les expressions sont évaluées dans un **environnement** (qui lie des identificateurs à des valeurs)
`a + b` est une expression qui sera évaluée à 10 si `a` est lié à la valeur 4 et `b` à la valeur 6.
- On manipule l'environnement **global** grâce au mot clef **let** :
`let id = expr;;`

C6 OCaml : aspects fonctionnels

5. Structure d'un programme Ocaml

Structure générale d'un programme OCaml

- Un programme OCaml repose sur l'évaluation d'une ou plusieurs expressions :
`expr1 ;;`
`expr2 ;;`
`...`
- Les expressions sont évaluées dans un **environnement** (qui lie des identificateurs à des valeurs)
`a + b` est une expression qui sera évaluée à 10 si `a` est lié à la valeur 4 et `b` à la valeur 6.
- On manipule l'environnement **global** grâce au mot clef **let** :
`let id = expr;;`
- Le mot clef **in** permet de créer un environnement **local** :
`let id = expr1 in expr2;;`
Si `id` existe déjà dans l'environnement courant il est provisoirement masqué.

C6 OCaml : aspects fonctionnels

5. Structure d'un programme Ocaml

Fonctions

- L'appel d'une fonction f à n arguments s'écrit en OCaml : $f \ x1 \ x2 \ \dots \ xn$

⚠ Attention au parenthésage !

$f \ 2 \ * \ 3$ est équivalent à $(f \ 2) \ * \ 3$ et vaut $f(2) \times 3$

$f \ (2*3)$ vaut $f(2 \times 3)$.

C6 OCaml : aspects fonctionnels

5. Structure d'un programme OCaml

Fonctions

- L'appel d'une fonction f à n arguments s'écrit en OCaml : $f \ x1 \ x2 \ \dots \ xn$

⚠ Attention au parenthésage !

$f \ 2 \ * \ 3$ est équivalent à $(f \ 2) \ * \ 3$ et vaut $f(2) \times 3$

$f \ (2*3)$ vaut $f(2 \times 3)$.

- OCaml traite les fonctions à plusieurs variables comme une fonction à un argument qui renvoie une fonction sur le reste des variables. C'est ce qu'on appelle la **curryfication** (du nom du mathématicien américain Haskell Curry).

c'est-à-dire que par exemple :

$$\begin{array}{ccc} f : E \times F & \leftarrow & G \\ (x, y) & \mapsto & z \end{array} \quad \text{s'interprète comme} \quad \begin{array}{ccc} f : E & \leftarrow & \mathcal{A}(F, G) \\ x & \mapsto & (y \mapsto z) \end{array}$$

C6 OCaml : aspects fonctionnels

5. Structure d'un programme OCaml

Fonctions

- L'appel d'une fonction f à n arguments s'écrit en OCaml : $f \ x1 \ x2 \ \dots \ xn$

⚠ Attention au parenthésage !

$f \ 2 \ * \ 3$ est équivalent à $(f \ 2) \ * \ 3$ et vaut $f(2) \times 3$

$f \ (2*3)$ vaut $f(2 \times 3)$.

- OCaml traite les fonctions à plusieurs variables comme une fonction à un argument qui renvoie une fonction sur le reste des variables. C'est ce qu'on appelle la **curryfication** (du nom du mathématicien américain Haskell Curry). c'est-à-dire que par exemple :

$$\begin{array}{ccc} f : E \times F \leftarrow G & & f : E \leftarrow \mathcal{A}(F, G) \\ (x, y) \mapsto z & \text{s'interprète comme} & x \mapsto (y \mapsto z) \end{array}$$

- On donne donc la signature d'une fonction en OCaml, en listant les types des arguments séparés par \rightarrow et ensuite le type du résultat.

C6 OCaml : aspects fonctionnels

5. Structure d'un programme OCaml

Fonctions

- L'appel d'une fonction f à n arguments s'écrit en OCaml : $f \ x1 \ x2 \ \dots \ xn$

⚠ Attention au parenthésage !

$f \ 2 \ * \ 3$ est équivalent à $(f \ 2) \ * \ 3$ et vaut $f(2) \times 3$

$f \ (2*3)$ vaut $f(2 \times 3)$.

- OCaml traite les fonctions à plusieurs variables comme une fonction à un argument qui renvoie une fonction sur le reste des variables. C'est ce qu'on appelle la **curryfication** (du nom du mathématicien américain Haskell Curry). c'est-à-dire que par exemple :

$$\begin{array}{ccc} f : E \times F \leftarrow G & & f : E \leftarrow \mathcal{A}(F, G) \\ (x, y) \mapsto z & \text{s'interprète comme} & x \mapsto (y \mapsto z) \end{array}$$

- On donne donc la signature d'une fonction en OCaml, en listant les types des arguments séparés par \rightarrow et ensuite le type du résultat.

Par exemple une fonction `min3` renvoyant le minimum de trois entiers s'écrit `min3 : int -> int -> int -> int`.

C6 OCaml : aspects fonctionnels

5. Structure d'un programme OCaml

Fonctions

- L'appel d'une fonction f à n arguments s'écrit en OCaml : `f x1 x2 ... xn`

⚠ Attention au parenthésage !

`f 2 * 3` est équivalent à `(f 2) * 3` et vaut $f(2) \times 3$

`f (2*3)` vaut $f(2 \times 3)$.

- OCaml traite les fonctions à plusieurs variables comme une fonction à un argument qui renvoie une fonction sur le reste des variables. C'est ce qu'on appelle la **curryfication** (du nom du mathématicien américain Haskell Curry). c'est-à-dire que par exemple :

$$\begin{array}{ccc} f : E \times F \leftarrow G & & f : E \leftarrow \mathcal{A}(F, G) \\ (x, y) \mapsto z & \text{s'interprète comme} & x \mapsto (y \mapsto z) \end{array}$$

- On donne donc la signature d'une fonction en OCaml, en listant les types des arguments séparés par `->` et ensuite le type du résultat.

Par exemple une fonction `min3` renvoyant le minimum de trois entiers s'écrit `min3 : int -> int -> int -> int`.

- L'évaluation d'une fonction sans donner tous ses arguments est une fonction.

C6 OCaml : aspects fonctionnels

5. Structure d'un programme Ocaml

Exemple

```
1  let aire_rectangle long larg = long*larg;;
2
3  let aire_cercle r =
4      let pi = 3.1415 in 2.*.pi*.r;;
5
6  print_int (aire_rectangle 4 10) ;;
7  print_newline() ;;
8  print_float (aire_cercle 5.);;
9  print_newline() ;;
```

C6 OCaml : aspects fonctionnels

5. Structure d'un programme Ocaml

Exemple

```
1  let distance x1 y1 x2 y2 =  
2    sqrt((x1-.x2)**2.-.(y1-.y2)**2.) ;;  
3  
4  (* Curryfication *)  
5  let distance_a_zero = distance 0. 0. ;;  
6  
7  print_float (distance 0. 0. 10. 7.);;  
8  print_newline();;  
9  print_float (distance_a_zero 10. 7.);;  
10 print_newline();;
```

C6 OCaml : aspects fonctionnels

6. Les types construits

Principe

Les trois mécanismes suivants permettent de construire de nouveaux types en OCaml à partir des types de base (`int`, `float`, `bool`, ...)

Principe

Les trois mécanismes suivants permettent de construire de nouveaux types en OCaml à partir des types de base (`int`, `float`, `bool`, ...)

- Les couples, triplets et plus généralement les n -uplets, définissent des types de la forme (x_1, \dots, x_n) où chacun des x_i peut avoir son propre type de base.

C6 OCaml : aspects fonctionnels

6. Les types construits

Principe

Les trois mécanismes suivants permettent de construire de nouveaux types en OCaml à partir des types de base (`int`, `float`, `bool`, ...)

- Les couples, triplets et plus généralement les n-uplets, définissent des types de la forme (x_1, \dots, x_n) ou chacun des x_i peut avoir son propre type de base.

Ex : `type point = float * float;;` définit un couple de flottant.

C6 OCaml : aspects fonctionnels

6. Les types construits

Principe

Les trois mécanismes suivants permettent de construire de nouveaux types en OCaml à partir des types de base (`int`, `float`, `bool`, ...)

- Les couples, triplets et plus généralement les n-uplets, définissent des types de la forme (x_1, \dots, x_n) ou chacun des x_i peut avoir son propre type de base.

Ex : `type point = float * float;;` définit un couple de flottant.

Les fonctions `fst` et `snd` permettent d'accéder au premier et au second élément d'un n-uplet.

C6 OCaml : aspects fonctionnels

6. Les types construits

Principe

Les trois mécanismes suivants permettent de construire de nouveaux types en OCaml à partir des types de base (`int`, `float`, `bool`, ...)

- Les couples, triplets et plus généralement les n-uplets, définissent des types de la forme (x_1, \dots, x_n) où chacun des x_i peut avoir son propre type de base.
Ex : `type point = float * float;;` définit un couple de flottant.
Les fonctions `fst` et `snd` permettent d'accéder au premier et au second élément d'un n-uplet.
- Les types enregistrements (ou type produits).

C6 OCaml : aspects fonctionnels

6. Les types construits

Principe

Les trois mécanismes suivants permettent de construire de nouveaux types en OCaml à partir des types de base (`int`, `float`, `bool`, ...)

- Les couples, triplets et plus généralement les n-uplets, définissent des types de la forme (x_1, \dots, x_n) où chacun des x_i peut avoir son propre type de base.

Ex : `type point = float * float;;` définit un couple de flottant.

Les fonctions `fst` et `snd` permettent d'accéder au premier et au second élément d'un n-uplet.

- Les types enregistrements (ou type produits).

Ex : `type date = {jour : int; mois : string; annee : int}`

C6 OCaml : aspects fonctionnels

6. Les types construits

Principe

Les trois mécanismes suivants permettent de construire de nouveaux types en OCaml à partir des types de base (`int`, `float`, `bool`, ...)

- Les couples, triplets et plus généralement les n-uplets, définissent des types de la forme (x_1, \dots, x_n) où chacun des x_i peut avoir son propre type de base.

Ex : `type point = float * float;;` définit un couple de flottant.

Les fonctions `fst` et `snd` permettent d'accéder au premier et au second élément d'un n-uplet.

- Les types enregistrements (ou type produits).

Ex : `type date = {jour : int; mois : string; annee : int}`

Accès aux champs avec la notation `.` (comme en C).

C6 OCaml : aspects fonctionnels

6. Les types construits

Principe

Les trois mécanismes suivants permettent de construire de nouveaux types en OCaml à partir des types de base (`int`, `float`, `bool`, ...)

- Les couples, triplets et plus généralement les n-uplets, définissent des types de la forme (x_1, \dots, x_n) où chacun des x_i peut avoir son propre type de base.
Ex : `type point = float * float;;` définit un couple de flottant.
Les fonctions `fst` et `snd` permettent d'accéder au premier et au second élément d'un n-uplet.
- Les types enregistrements (ou type produits).
Ex : `type date = {jour : int; mois : string; annee : int}`
Accès aux champs avec la notation `.` (comme en C).
- Les types unions (ou encore types sommes) où on liste les valeurs possibles séparés par des `|`.

C6 OCaml : aspects fonctionnels

6. Les types construits

Principe

Les trois mécanismes suivants permettent de construire de nouveaux types en OCaml à partir des types de base (`int`, `float`, `bool`, ...)

- Les couples, triplets et plus généralement les n-uplets, définissent des types de la forme (x_1, \dots, x_n) ou chacun des x_i peut avoir son propre type de base.

Ex : `type point = float * float;;` définit un couple de flottant.

Les fonctions `fst` et `snd` permettent d'accéder au premier et au second élément d'un n-uplet.

- Les types enregistrements (ou type produits).

Ex : `type date = {jour : int; mois : string; annee : int}`

Accès aux champs avec la notation `.` (comme en C).

- Les types unions (ou encore types sommes) où on liste les valeurs possibles séparés par des `|`.

Ex : `type signe = Positif | Negatif | Nul;;`

C6 OCaml : aspects fonctionnels

6. Les types construits

Principe

Les trois mécanismes suivants permettent de construire de nouveaux types en OCaml à partir des types de base (`int`, `float`, `bool`, ...)

- Les couples, triplets et plus généralement les n-uplets, définissent des types de la forme (x_1, \dots, x_n) où chacun des x_i peut avoir son propre type de base.

Ex : `type point = float * float;;` définit un couple de flottant.

Les fonctions `fst` et `snd` permettent d'accéder au premier et au second élément d'un n-uplet.

- Les types enregistrements (ou type produits).

Ex : `type date = {jour : int; mois : string; annee : int}`

Accès aux champs avec la notation `.` (comme en C).

- Les types unions (ou encore types sommes) où on liste les valeurs possibles séparés par des `|`.

Ex : `type signe = Positif | Negatif | Nul;;`

Ex : `type carte = Roi | Dame | Valet | Nombre of int;;`

C6 OCaml : aspects fonctionnels

6. Les types construits

Principe

Les trois mécanismes suivants permettent de construire de nouveaux types en OCaml à partir des types de base (`int`, `float`, `bool`, ...)

- Les couples, triplets et plus généralement les n-uplets, définissent des types de la forme (x_1, \dots, x_n) où chacun des x_i peut avoir son propre type de base.

Ex : `type point = float * float;;` définit un couple de flottant.

Les fonctions `fst` et `snd` permettent d'accéder au premier et au second élément d'un n-uplet.

- Les types enregistrements (ou type produits).

Ex : `type date = {jour : int; mois : string; annee : int}`

Accès aux champs avec la notation `.` (comme en C).

- Les types unions (ou encore types sommes) où on liste les valeurs possibles séparés par des `|`.

Ex : `type signe = Positif | Negatif | Nul;;`

Ex : `type carte = Roi | Dame | Valet | Nombre of int;;`

⚠ Les constructeurs commencent par une majuscule.

Exemple

- 1 Définir le type complexe comme couple de flottant, écrire la fonction module pour ce type.

Exemple

- 1 Définir le type complexe comme couple de flottant, écrire la fonction module pour ce type.
- 2 Définir le type menu comme un type enregistrement composé des champs entree (string), plat (string), dessert(string) et prix (float).


Exemple

- 1 Définir le type complexe comme couple de flottant, écrire la fonction module pour ce type.
- 2 Définir le type menu comme un type enregistrement composé des champs entree (string), plat (string), dessert(string) et prix (float).
- 3 Définir le type rbarre ($\overline{\mathbb{R}}$), comme un type somme pouvant être Plusinfini, Moinsinfini et les flottants.

Exemple : correction

① Type produit complexe

```
1  type complexe = float * float;;  
2  
3  let modu (z:complexe) =  
4      let x = (fst z) in  
5      let y = (snd z) in  
6      sqrt(x**2.+y**2.);;
```

 On utilise l'annotation de type pour spécifier que l'argument est de type complexe;

Exemple : correction

① Type produit complexe

```
1  type complexe = float * float;;
2
3  let modu (z:complexe) =
4      let x = (fst z) in
5      let y = (snd z) in
6      sqrt(x**2.+y**2.);;
```

⚠ On utilise l'annotation de type pour spécifier que l'argument est de type complexe;

② Type menu enregistrement

```
1  type menu = {entree : string; plat : string; dessert : string;
  ↪   prix : float};;
```

Exemple : correction

① Type produit complexe

```
1  type complexe = float * float;;
2
3  let modu (z:complexe) =
4      let x = (fst z) in
5      let y = (snd z) in
6      sqrt(x**2.+y**2.);;
```

⚠ On utilise l'annotation de type pour spécifier que l'argument est de type complexe;

② Type menu enregistrement

```
1  type menu = {entree : string; plat : string; dessert : string;
   ↪ prix : float};;
```

③ Type somme rbarre

```
1  type rbarre = Plusinfini | Moinsinfini | Nombre of float;;
```

Définition

Définition

- Le filtrage par motif (*pattern matching*) est un mécanisme permettant de travailler efficacement sur les types construits en les décomposant et les analysant suivant leur structure. On peut ainsi indiquer l'expression à évaluer suivant la forme spécifique de l'entrée.

Définition

- Le filtrage par motif (*pattern matching*) est un mécanisme permettant de travailler efficacement sur les types construits en les décomposant et les analysant suivant leur structure. On peut ainsi indiquer l'expression à évaluer suivant la forme spécifique de l'entrée.

- La syntaxe générale est :


```
match expr with  
| motif1 -> expr1  
  
| :  
  
| motifn -> exprn
```

Définition

- Le filtrage par motif (*pattern matching*) est un mécanisme permettant de travailler efficacement sur les types construits en les décomposant et les analysant suivant leur structure. On peut ainsi indiquer l'expression à évaluer suivant la forme spécifique de l'entrée.


- La syntaxe générale est :

```
match expr with  
| motif1 -> expr1  
  
| :  
| motifn -> exprn
```

 Le filtrage doit être exhaustif, le caractère spécial `_` indique un motif qui correspond à toutes les entrées.


Définition

- Le filtrage par motif (*pattern matching*) est un mécanisme permettant de travailler efficacement sur les types construits en les décomposant et les analysant suivant leur structure. On peut ainsi indiquer l'expression à évaluer suivant la forme spécifique de l'entrée.
- La syntaxe générale est :
`match` `expr` `with`
| `motif1` -> `expr1`

| `:`
| `motifn` -> `exprn`
 Le filtrage doit être exhaustif, le caractère spécial `_` indique un motif qui correspond à toutes les entrées.
- L'ordre du filtrage est important car si deux motifs correspondent à une entrée c'est l'expression du premier rencontré qui sera évalué.


Définition

- Le filtrage par motif (*pattern matching*) est un mécanisme permettant de travailler efficacement sur les types construits en les décomposant et les analysant suivant leur structure. On peut ainsi indiquer l'expression à évaluer suivant la forme spécifique de l'entrée.
- La syntaxe générale est :
`match` `expr` `with`
| `motif1` -> `expr1`

| `:`
| `motifn` -> `exprn`
 Le filtrage doit être exhaustif, le caractère spécial `_` indique un motif qui correspond à toutes les entrées.
- L'ordre du filtrage est important car si deux motifs correspondent à une entrée c'est l'expression du premier rencontré qui sera évalué.
- Chaque identifiant apparaît une fois au plus dans un motif.

Définition

- Le filtrage par motif (*pattern matching*) est un mécanisme permettant de travailler efficacement sur les types construits en les décomposant et les analysant suivant leur structure. On peut ainsi indiquer l'expression à évaluer suivant la forme spécifique de l'entrée.
- La syntaxe générale est :
`match` `expr` `with`
| `motif1` -> `expr1`

| `:`
| `motifn` -> `exprn`
 Le filtrage doit être exhaustif, le caractère spécial `_` indique un motif qui correspond à toutes les entrées.
- L'ordre du filtrage est important car si deux motifs correspondent à une entrée c'est l'expression du premier rencontré qui sera évalué.
- Chaque identifiant apparaît une fois au plus dans un motif.
- On peut filter un n-uplet sur un n-uplet de motif.

C6 OCaml : aspects fonctionnels

7. Filtrage par motifs

Exemple

On a déjà rencontré le type carte :

```
type carte = Roi | Dame | Valet | Nombre of int;;
```

On peut associer chaque carte à sa valeur (à la belote) à l'aide d'un pattern matching :

```
1  type carte = As | Roi | Dame | Valet | Nombre of int;;
2
3  let valeur c =
4      match c with
5      | As -> 11
6      | Roi -> 4
7      | Dame -> 3
8      | Valet -> 2
9      | Nombre 10 -> 10
10     | _ -> 0
```

Exemple

On peut aussi filtrer un couple de carte afin de détecter une éventuelle paire (deux cartes identiques) :

```
1  let est_paire c1 c2 =  
2      match (c1, c2) with  
3      | (Roi, Roi) -> true  
4      | (Dame, Dame) -> true  
5      | (Valet, Valet) -> true  
6      | (Nombre x, Nombre y) -> x = y  
7      | (_, _) -> false
```

Exercice

- 1 Définir un type somme signe avec les valeurs Négatif, Positif et Nul.

Exercice

- 1 Définir un type somme signe avec les valeurs `Negatif`, `Positif` et `Nul`.

```
1 type signe = Negatif | Nul | Positif
```

Exercice

- 1 Définir un type somme signe avec les valeurs `Negatif`, `Positif` et `Nul`.

```
1 type signe = Negatif | Nul | Positif
```

- 2 Ecrire une fonction `produit` qui renvoie le signe d'un produit

Exercice

- ① Définir un type somme signe avec les valeurs `Negatif`, `Positif` et `Nul`.

```
1  type signe = Negatif | Nul | Positif
```

- ② Ecrire une fonction `produit` qui renvoie le signe d'un produit

```
1  let produit s1 s2 =  
2      match s1, s2 with  
3      | Nul, _ -> Nul  
4      | _, Nul -> Nul  
5      | Positif, Positif -> Positif  
6      | Negatif, Negatif -> Positif  
7      | _, _ -> Negatif
```


Exemple

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

Exemple

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

Cette fonction peut prendre en entrée trois entiers, trois flottants, trois caractères, ... sa signature est `deux_egaux : 'a -> 'a -> 'a -> bool`. On dit qu'elle est polymorphe car elle accepte des entrées de types différents, le type des entrées est donc noté par OCaml `'a`.

C6 OCaml : aspects fonctionnels

8. Polymorphisme

Exemple

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

Cette fonction peut prendre en entrée trois entiers, trois flottants, trois caractères, ... sa signature est `deux_egaux : 'a -> 'a -> 'a -> bool`. On dit qu'elle est polymorphe car elle accepte des entrées de types différents, le type des entrées est donc noté par OCaml `'a`.

Définition

Le **polymorphisme** est un mécanisme permettant d'écrire des fonctions prenant en entrée des valeurs de types différents. Le type des arguments est alors noté `'a`, `'b`,

le type `list`

- Le type `list` est prédéfini en OCaml et permet de représenter des listes de valeurs **de même type**. Une liste est **non mutable**.

le type `list`

- Le type `list` est prédéfini en OCaml et permet de représenter des listes de valeurs **de même type**. Une liste est **non mutable**.
- On peut définir une liste en mettant les éléments entre crochets et en les séparant par des points virgules.

le type `list`

- Le type `list` est prédéfini en OCaml et permet de représenter des listes de valeurs **de même type**. Une liste est **non mutable**.
- On peut définir une liste en mettant les éléments entre crochets et en les séparant par des points virgules.

`let` `p` = `[2; 3; 5; 7]` : une liste d'entiers

le type `list`

- Le type `list` est prédéfini en OCaml et permet de représenter des listes de valeurs **de même type**. Une liste est **non mutable**.
- On peut définir une liste en mettant les éléments entre crochets et en les séparant par des points virgules.

`let` `p` = `[2; 3; 5; 7]` : une liste d'entiers

`let` `lang` = `["Python"; "Ada"; "C"]` : une liste de chaînes de caractères

le type `list`

- Le type `list` est prédéfini en OCaml et permet de représenter des listes de valeurs **de même type**. Une liste est **non mutable**.
- On peut définir une liste en mettant les éléments entre crochets et en les séparant par des points virgules.

`let` `p` = `[2; 3; 5; 7]` : une liste d'entiers

`let` `lang` = `["Python"; "Ada"; "C"]` : une liste de chaînes de caractères

`let` `bug` = `[3.14; 2.71; 42; 0.57]` : erreur car liste non homogène

le type `list`

- Le type `list` est prédéfini en OCaml et permet de représenter des listes de valeurs **de même type**. Une liste est **non mutable**.
- On peut définir une liste en mettant les éléments entre crochets et en les séparant par des points virgules.
`let p = [2; 3; 5; 7] : une liste d'entiers`
`let lang = ["Python"; "Ada"; "C"] : une liste de chaînes de caractères`
`let bug = [3.14; 2.71; 42; 0.57] : erreur car liste non homogène`
- Une liste est un type récursif, en effet :

le type `list`

- Le type `list` est prédéfini en OCaml et permet de représenter des listes de valeurs **de même type**. Une liste est **non mutable**.
- On peut définir une liste en mettant les éléments entre crochets et en les séparant par des points virgules.
`let p = [2; 3; 5; 7] : une liste d'entiers`
`let lang = ["Python"; "Ada"; "C"] : une liste de chaînes de caractères`
`let bug = [3.14; 2.71; 42; 0.57] : erreur car liste non homogène`
- Une liste est un type récursif, en effet :
 - Elle est vide (noté `[]`) ou

le type `list`

- Le type `list` est prédéfini en OCaml et permet de représenter des listes de valeurs **de même type**. Une liste est **non mutable**.
- On peut définir une liste en mettant les éléments entre crochets et en les séparant par des points virgules.
`let p = [2; 3; 5; 7] : une liste d'entiers`
`let lang = ["Python"; "Ada"; "C"] : une liste de chaînes de caractères`
`let bug = [3.14; 2.71; 42; 0.57] : erreur car liste non homogène`
- Une liste est un type récursif, en effet :
 - Elle est vide (noté `[]`) ou
 - Elle est constituée d'un élément (appelée tête) et d'une liste (appelée queue)

le type `list`

- Le type `list` est prédéfini en OCaml et permet de représenter des listes de valeurs **de même type**. Une liste est **non mutable**.
- On peut définir une liste en mettant les éléments entre crochets et en les séparant par des points virgules.
`let p = [2; 3; 5; 7] : une liste d'entiers`
`let lang = ["Python"; "Ada"; "C"] : une liste de chaînes de caractères`
`let bug = [3.14; 2.71; 42; 0.57] : erreur car liste non homogène`
- Une liste est un type récursif, en effet :
 - Elle est vide (noté `[]`) ou
 - Elle est constituée d'un élément (appelée tête) et d'une liste (appelée queue)
- Le filtrage par motif est la technique standard pour opérer sur les listes. On utilise alors le constructeur `::` pour "séparer" la tête et la queue.

Exemple de filtrage par motif sur les listes

- fonction qui renvoie la longueur d'une liste

Exemple de filtrage par motif sur les listes

- fonction qui renvoie la longueur d'une liste

```
1  let rec taille liste =  
2    match liste with  
3    | [] -> 0  
4    | h::t -> 1 + taille t
```

Exemple de filtrage par motif sur les listes

- fonction qui renvoie la longueur d'une liste

```
1  let rec taille liste =  
2    match liste with  
3    | [] -> 0  
4    | h::t -> 1 + taille t
```

- fonction qui renvoie **true** si un élément est dans une liste et **false** sinon.

Exemple de filtrage par motif sur les listes

- fonction qui renvoie la longueur d'une liste

```
1  let rec taille liste =  
2    match liste with  
3    | [] -> 0  
4    | h::t -> 1 + taille t
```

- fonction qui renvoie **true** si un élément est dans une liste et **false** sinon.

```
1  let rec est_dans elt liste =  
2    match liste with  
3    | [] -> false  
4    | h::t -> elt=h || est_dans elt t
```


Fonctions usuelles sur les listes

- @ est l'opérateur de concaténation.

Fonctions usuelles sur les listes

- @ est l'opérateur de concaténation.

Par exemple, `let ex1 = [1; -2] @ [-3; 4]` va créer la liste
`ex1 = [1; -2; -3; 4]`.

Fonctions usuelles sur les listes

- `@` est l'opérateur de concaténation.

Par exemple, `let ex1 = [1; -2] @ [-3; 4]` va créer la liste
`ex1 = [1; -2; -3; 4]`.

- `List.hd` et `List.tl` permettent de récupérer respectivement la tête et la queue de la liste.

Fonctions usuelles sur les listes

- `@` est l'opérateur de concaténation.

Par exemple, `let ex1 = [1; -2] @ [-3; 4]` va créer la liste
`ex1 = [1; -2; -3; 4]`.

- `List.hd` et `List.tl` permettent de récupérer respectivement la tête et la queue de la liste.

Par exemple, `List.hd ex1` renvoie l'entier 1 et `List.tl ex1` renvoie la liste
`[-2; -3; 4]`

Fonctions usuelles sur les listes

- `@` est l'opérateur de concaténation.

Par exemple, `let ex1 = [1; -2] @ [-3; 4]` va créer la liste
`ex1 = [1; -2; -3; 4]`.

- `List.hd` et `List.tl` permettent de récupérer respectivement la tête et la queue de la liste.

Par exemple, `List.hd ex1` renvoie l'entier 1 et `List.tl ex1` renvoie la liste
`[-2; -3; 4]`

- `List.length` renvoie la longueur de la liste.

Fonctions usuelles sur les listes

- `@` est l'opérateur de concaténation.

Par exemple, `let ex1 = [1; -2] @ [-3; 4]` va créer la liste
`ex1 = [1; -2; -3; 4]`.

- `List.hd` et `List.tl` permettent de récupérer respectivement la tête et la queue de la liste.

Par exemple, `List.hd ex1` renvoie l'entier 1 et `List.tl ex1` renvoie la liste
`[-2; -3; 4]`

- `List.length` renvoie la longueur de la liste.

Par exemple, `List.length ex1` renvoie 4

Fonctions usuelles sur les listes

- `@` est l'opérateur de concaténation.

Par exemple, `let ex1 = [1; -2] @ [-3; 4]` va créer la liste
`ex1 = [1; -2; -3; 4]`.

- `List.hd` et `List.tl` permettent de récupérer respectivement la tête et la queue de la liste.

Par exemple, `List.hd ex1` renvoie l'entier 1 et `List.tl ex1` renvoie la liste
`[-2; -3; 4]`

- `List.length` renvoie la longueur de la liste.

Par exemple, `List.length ex1` renvoie 4

- `List.map` renvoie la liste obtenu en appliquant la fonction donnée en paramètre à chaque élément de la liste.

Fonctions usuelles sur les listes

- `@` est l'opérateur de concaténation.

Par exemple, `let ex1 = [1; -2] @ [-3; 4]` va créer la liste
`ex1 = [1; -2; -3; 4]`.

- `List.hd` et `List.tl` permettent de récupérer respectivement la tête et la queue de la liste.

Par exemple, `List.hd ex1` renvoie l'entier 1 et `List.tl ex1` renvoie la liste
`[-2; -3; 4]`

- `List.length` renvoie la longueur de la liste.

Par exemple, `List.length ex1` renvoie 4

- `List.map` renvoie la liste obtenu en appliquant la fonction donnée en paramètre à chaque élément de la liste.

Par exemple, `List.map abs ex1` renvoie `[1; 2; 3; 4]`

Fonctions usuelles sur les listes

- `@` est l'opérateur de concaténation.

Par exemple, `let ex1 = [1; -2] @ [-3; 4]` va créer la liste
`ex1 = [1; -2; -3; 4]`.

- `List.hd` et `List.tl` permettent de récupérer respectivement la tête et la queue de la liste.

Par exemple, `List.hd ex1` renvoie l'entier 1 et `List.tl ex1` renvoie la liste
`[-2; -3; 4]`

- `List.length` renvoie la longueur de la liste.

Par exemple, `List.length ex1` renvoie 4

- `List.map` renvoie la liste obtenu en appliquant la fonction donnée en paramètre à chaque élément de la liste.

Par exemple, `List.map abs ex1` renvoie `[1; 2; 3; 4]`

- `List.fold_left` et `List.fold_right` permettent d'effectuer des opérations entre les éléments d'une liste. On doit fournir l'opération et l'opérande de départ.

Fonctions usuelles sur les listes

- `@` est l'opérateur de concaténation.

Par exemple, `let ex1 = [1; -2] @ [-3; 4]` va créer la liste
`ex1 = [1; -2; -3; 4]`.

- `List.hd` et `List.tl` permettent de récupérer respectivement la tête et la queue de la liste.

Par exemple, `List.hd ex1` renvoie l'entier 1 et `List.tl ex1` renvoie la liste
`[-2; -3; 4]`

- `List.length` renvoie la longueur de la liste.

Par exemple, `List.length ex1` renvoie 4

- `List.map` renvoie la liste obtenu en appliquant la fonction donnée en paramètre à chaque élément de la liste.

Par exemple, `List.map abs ex1` renvoie `[1; 2; 3; 4]`

- `List.fold_left` et `List.fold_right` permettent d'effectuer des opérations entre les éléments d'une liste. On doit fournir l'opération et l'opérande de départ.

Par exemple, `List.fold_left (*) 1 ex1` renvoie 24.