

Programmation Fonctionnelle

Cours 01 et 02

Michele Pagani



Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

`michele.pagani@pps.univ-paris-diderot.fr`

14 et 21 septembre 2015

Organisation

Intervenants:

CM: Michele Pagani,
michele.pagani@pps.univ-paris-diderot.fr,
lundi 10h30-12h30, Amphi 3B

TP INFO1: Arthur Milchior,
arthur@milchior.fr,
vendredi 9h30-12h30, Salle 2003

TP INFO2: Peter Habermehl,
peter.habermehl@liafa.univ-paris-diderot.fr,
mardi 9h30-12h30, Salle 2031

TP INFO3: Mathieu Laurière,
mathieu.lauriere@gmail.com,
mercredi 9h30-12h30, Salle 2001

TP MI: Antonio Bucciarelli,
antonio.bucciarelli@pps.univ-paris-diderot.fr,
mardi 14h30-17h30, Salle 2031

Contrôle de connaissances

- Première session :

$$\frac{1}{2} * \text{exam1} + \frac{1}{2} * \text{projet}$$

- Deuxième session :

$$\max(\text{exam2}, \frac{1}{2} * \text{exam2} + \frac{1}{2} * \text{projet})$$

Attention ! Le projet est obligatoire et compte dans les deux sessions !

Contrôle de connaissances

- Première session :

$$\frac{1}{2} * \text{exam1} + \frac{1}{2} * \text{projet}$$

- Deuxième session :

$$\max(\text{exam2}, \frac{1}{2} * \text{exam2} + \frac{1}{2} * \text{projet})$$

Attention ! Le projet est obligatoire et compte dans les deux sessions !

Contrôle continu

- PROJET :
- à rendre sur DiDel le mercredi 30 décembre
 - À faire en binôme, mais. . .
attention: soutenance et note individuelles !
 - plagats ou absence = 0
 - Plus sur l'organisation du projet : voir les TP

Organisation

- Page web: sur DidEL: PF5 - Programmation Fonctionnelle

Inscrivez vous !

- Support : copies des transparents
 - ce n'est pas un livre !
 - il ne suffit pas. . .
- Il est indispensable d'assister au cours et au TP, et de faire le projet (**obligatoire**).
- Il y a des ressources en ligne (voir plus tard)

Organisation

- Page web: sur [DidEL: PF5 - Programmation Fonctionnelle](#)

Inscrivez vous !

- Support : copies des transparents
 - ce n'est pas un livre !
 - il ne suffit pas. . .
- Il est indispensable d'assister au cours et au TP, et de faire le projet (**obligatoire**).
- Il y a des ressources en ligne (voir plus tard)

Organisation

- Page web: sur [DidEL: PF5 - Programmation Fonctionnelle](#)

Inscrivez vous !

- Support : copies des transparents
 - ce n'est pas un livre !
 - il ne suffit pas. . .
- Il est indispensable d'assister au cours et au TP, et de faire le projet (**obligatoire**).
- Il y a des ressources en ligne (voir plus tard)

Organisation

- Page web: sur [DidEL: PF5 - Programmation Fonctionnelle](#)

Inscrivez vous !

- Support : copies des transparents
 - ce n'est pas un livre !
 - il ne suffit pas. . .
- Il est indispensable d'assister au cours et au TP, et de faire le projet (**obligatoire**).
- Il y a des ressources en ligne (voir plus tard)

Qu'est-ce que la programmation fonctionnelle ?

Deux styles face à face

Style impératif

```
1 static int gcd(int m, int n){
2   int aux;
3   while (m != 0) {
4     aux = m;
5     m = n % m;
6     n = aux;
7   }
8   return n;
9 }
```

- **programme**: séquence structurée d'instructions modifiant l'état de la machine
- **évaluer**: exécuter les instructions les unes après les autres jusqu'au point de sortie

Style fonctionnel

```
1 let rec gcd(m,n) =
2   if m=0 then n
3   else gcd(n mod m, m)
```

- **programme**: expression définissant une fonction
- **évaluer**: réécrire une expression jusqu'à obtenir une valeur

comme en math:
 $(2 + 3) * 4 = 5 * 4 = 20$

Deux styles face à face

Style impératif

```
1 static int gcd(int m, int n){
2   int aux;
3   while (m != 0) {
4     aux = m;
5     m = n % m;
6     n = aux;
7   }
8   return n;
9 }
```

- **programme**: séquence structurée d'instructions modifiant l'état de la machine
- **évaluer**: exécuter les instructions les unes après les autres jusqu'au point de sortie

Style fonctionnel

```
1 let rec gcd(m,n) =
2   if m=0 then n
3   else gcd(n mod m, m)
```

- **programme**: expression définissant une fonction
- **évaluer**: réécrire une expression jusqu'à obtenir une valeur

comme en math:
 $(2 + 3) * 4 = 5 * 4 = 20$

Deux styles face à face

Style impératif

```
1 static int gcd(int m, int n){
2   int aux;
3   while (m != 0) {
4     aux = m;
5     m = n % m;
6     n = aux;
7   }
8   return n;
9 }
```

- **programme**: séquence structurée d'instructions modifiant l'état de la machine
- **évaluer**: exécuter les instructions les unes après les autres jusqu'au point de sortie

Style fonctionnel

```
1 let rec gcd(m, n) =
2   if m=0 then n
3   else gcd(n mod m, m)
```

- **programme**: expression définissant une fonction
- **évaluer**: réécrire une expression jusqu'à obtenir une valeur

comme en math:
 $(2 + 3) * 4 = 5 * 4 = 20$

Style impératif

```
1 static int gcd(int m, int n){  
2   int aux;  
3   while (m != 0) {  
4     aux = m;  
5     m = n % m;  
6     n = aux;  
7   }  
8   return n;  
9 }
```

PC	m	n	
1	4	6	aux
2	4	6	0
3	4	6	0
4	4	6	4
5	2	6	4
6	2	4	4
7	2	4	4
3	2	4	4
4	2	4	2
5	0	4	2
6	0	2	2
7	0	2	2
3	0	2	2
8	0	2	2

$$\begin{aligned}
&\text{gcd}(4,6) \xrightarrow[m=4, n=6]{} \text{if } m=0 \text{ then } n \text{ else gcd}(n \bmod m, m) \\
&\xrightarrow[m=4, n=6]{} \text{if } 4=0 \text{ then } n \text{ else gcd}(n \bmod m, m) \\
&\xrightarrow[m=4, n=6]{} \text{if false then } n \text{ else gcd}(n \bmod m, m) \\
&\xrightarrow[m=4, n=6]{} \text{gcd}(n \bmod m, m) \\
&\xrightarrow[m=4, n=6]{} \text{gcd}(6 \bmod 4, 4) \\
&\xrightarrow[m=4, n=6]{} \text{gcd}(2, 4) \\
&\xrightarrow[m=2, n=4]{} \text{if } m=0 \text{ then } n \text{ else gcd}(n \bmod m, m) \\
&\xrightarrow[m=2, n=4]{} \text{if } 2=0 \text{ then } n \text{ else gcd}(n \bmod m, m) \\
&\xrightarrow[m=2, n=4]{} \text{if false then } n \text{ else gcd}(n \bmod m, m) \\
&\xrightarrow[m=2, n=4]{} \text{gcd}(n \bmod m, m) \\
&\xrightarrow[m=2, n=4]{} \text{gcd}(4 \bmod 2, 2) \\
&\xrightarrow[m=2, n=4]{} \text{gcd}(0, 2) \\
&\xrightarrow[m=0, n=2]{} \text{if } 0=0 \text{ then } n \text{ else gcd}(n \bmod m, m) \\
&\xrightarrow[m=0, n=2]{} n \xrightarrow[m=0, n=2]{} 2
\end{aligned}$$

Style fonctionnel

```

1  let rec gcd(m,n) =
2    if m=0 then n
3    else gcd(n mod m, m)

```

Qu'est-ce qu'on gagne ?

- Les fonctions sont des données de première classe: elles peuvent être retournées en résultat et passées en argument à d'autres fonctions
- Permet d'écrire des programmes qui sont plus proche au problème donné
- Favorise la vérification (même automatique) des comportements des programmes
- ... et cela souvent sans perte d'efficacité

Qu'est-ce qu'on gagne ?

- Les fonctions sont des données de première classe: elles peuvent être retournées en résultat et passées en argument à d'autres fonctions
- Permet d'écrire des programmes qui sont plus proche au problème donné
- Favorise la vérification (même automatique) des comportements des programmes
- ... et cela souvent sans perte d'efficacité

Qu'est-ce qu'on gagne ?

- Les fonctions sont des données de première classe: elles peuvent être retournées en résultat et passées en argument à d'autres fonctions
- Permet d'écrire des programmes qui sont plus proche au problème donné
- Favorise la vérification (même automatique) des comportements des programmes
- ... et cela souvent sans perte d'efficacité

Qu'est-ce qu'on gagne ?

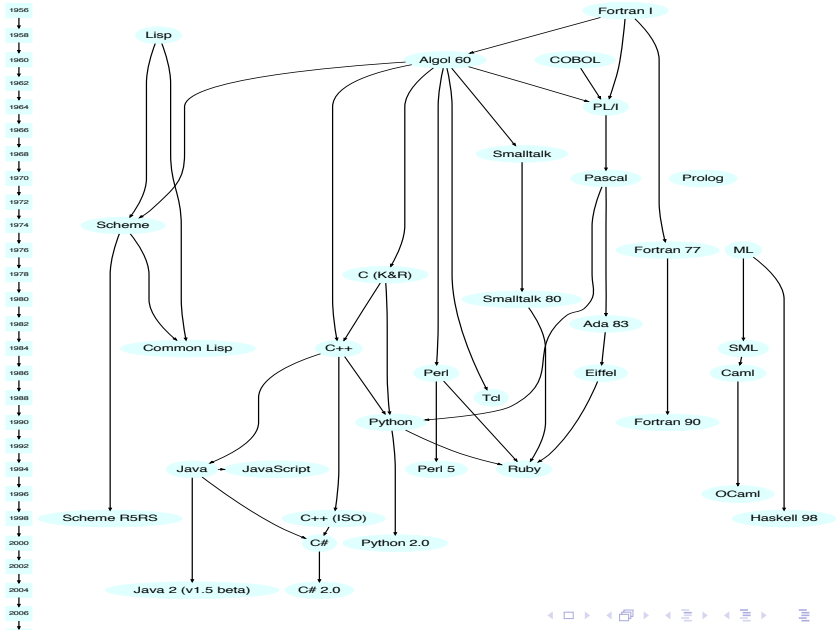
- Les fonctions sont des données de première classe: elles peuvent être retournées en résultat et passées en argument à d'autres fonctions
- Permet d'écrire des programmes qui sont plus proche au problème donné
- Favorise la vérification (même automatique) des comportements des programmes
- ... et cela souvent sans perte d'efficacité

Qu'est-ce qu'on gagne ?

- Les fonctions sont des données de première classe: elles peuvent être retournées en résultat et passées en argument à d'autres fonctions
- Permet d'écrire des programmes qui sont plus proche au problème donné
- Favorise la vérification (même automatique) des comportements des programmes
- ... et cela souvent sans perte d'efficacité

Le langage OCaml

Où on est



Qui a créé OCaml?

OCaml (Objective Caml) est le fruit de développements continus:

- 1975 R. Milner propose ML comme métalangage pour l'assistant de preuve LCF OCaml
- 1980 Projet Formel à l'INRIA (G. Huet), Categorical Abstract Machine (P. L. Curien) OCaml
- 1985 Développement de Caml à l'INRIA et, en parallèle, de Standard ML à Édimbourg, de SML à New-Jersey, de Lazy ML à Chalmers, de Haskell à Glasgow, etc.
- 1990 Implantation de Caml-Light par X. Leroy et D. Doligez
- 1995 Compilateur vers du code natif + système de modules
- 1996 Object et classes OCaml
- 2002 Méthodes polymorphes, bibliothèques partagées, etc.
- 2003 Modules récursifs, private types, etc.

Spécificités d'OCaml

Haut niveau

- fonctions fournissent mécanisme d'abstraction puissant
- gestion mémoire automatique (comme JAVA)

Multi-paradigme

- styles fonctionnel, impératif, orientée objet
- graphisme, applications réseaux, ...

Système de typage

- capture beaucoup d'erreurs
- types synthétisés automatiquement par le compilateur
- permet reutilisation du code grâce au polymorphisme

Performance

- génération de code efficace

Spécificités d'OCaml

Haut niveau

- fonctions fournissent mécanisme d'abstraction puissant
- gestion mémoire automatique (comme JAVA)

Multi-paradigme

- styles fonctionnel, impératif, orientée objet
- graphisme, applications réseaux, ...

Système de typage

- capture beaucoup d'erreurs
- types synthétisés automatiquement par le compilateur
- permet reutilisation du code grâce au polymorphisme

Performance

- génération de code efficace

Spécificités d'OCaml

Haut niveau

- fonctions fournissent mécanisme d'abstraction puissant
- gestion mémoire automatique (comme JAVA)

Multi-paradigme

- styles fonctionnel, impératif, orientée objet
- graphisme, applications réseaux, ...

Système de typage

- capture beaucoup d'erreurs
- types synthétisés automatiquement par le compilateur
- permet reutilisation du code grâce au polymorphisme

Performance

- génération de code efficace

Spécificités d'OCaml

Haut niveau

- fonctions fournissent mécanisme d'abstraction puissant
- gestion mémoire automatique (comme JAVA)

Multi-paradigme

- styles fonctionnel, impératif, orientée objet
- graphisme, applications réseaux, ...

Système de typage

- capture beaucoup d'erreurs
- types synthétisés automatiquement par le compilateur
- permet reutilisation du code grâce au polymorphisme

Performance

- génération de code efficace

Ressources et Bibliographie

<http://ocaml.org/index.fr.html>



Apprendre

Documentation

Contributions

Communauté

Rechercher



OCaml est un langage de programmation de niveau industriel supportant les styles fonctionnel, impératif et orienté-objet

Installer OCaml



Apprendre

Une description d'OCaml, ses utilisateurs, des exemples de code, des tutoriaux à lire et bien plus.



Contributions

Le gestionnaire de paquets OPAM vous donne accès aux multiples versions de centaines de paquets.



Documentation

Installer OCaml, trouver des docs de paquets, accéder au Manuel, obtenir des mémentos et bien plus.



Communauté

Lire les fils de news, discuter et échanger, obtenir du support, rencontrer d'autres utilisateurs et trouver OCaml sur le web.

Nouvelles



OCaml 2015
4 septembre 2015



Weekly News
1 septembre 2015



Weekly News
25 août 2015



Tenth OCaml compiler hacking evening a...
20 août 2015



No (functional) experience required
19 août 2015



Haskell Engineer at Wagon (Full-time)
18 août 2015



Plus...



Apprendre OCaml dans son navigateur avec TryOCaml

MOOC sur OCAML

<https://www.france-universite-numerique-mooc.fr/courses/parisdiderot/56002/session01/about>



CONNEXION

DÉCOUVRIR, APPRENDRE ET RÉUSSIR

QU'EST-CE QUE FUN ?

ACTUALITÉS

LES COURS

LES ÉTABLISSEMENTS

S'INSCRIRE MAINTENANT

Introduction to functional programming in OCaml

S'INSCRIRE À INTRODUCTION TO FUNCTIONAL PROGRAMMING IN OCAML

PARIS
DIDEROT

présentation

ABOUT THIS COURSE

Functional programming is a programming paradigm which is rapidly attracting interest from a broad range of developers because it allows to write expressive, concise and elegant programs.

Voir la vidéo de présentation du cours



Disponibles gratuitement

- Xavier Leroy et al :
The Objective Caml system
<http://caml.inria.fr/pub/docs/manual-ocaml/>
(Le manuel officiel)
- Emmanuel Chailloux, Pascal Manoury et Bruno Pagano :
Développement d'Applications avec Objective Caml
O'Reilly, 2000
<http://caml.inria.fr/pub/docs/oreilly-book/>

Autres ouvrages

- John Whittington
OCaml from the Very Beginning
Coherent Press, 2013
(S'adresse plutôt à des débutants)
- Guy Cousineau et Michel Mauny
Approche fonctionnelle de la programmation
Dunod, 1995
- Pierre Weis et Xavier Leroy
Le langage Caml
Dunod, 1999

... enfin, une liste de livres sur la programmation fonctionnelle est maintenue par Alex Ott:

<http://alexott.net/en/fp/books/>

Modes de compilation

Deux façons de travailler avec OCaml

① **compiler** (comme en Java, C, ...) :

- permet d'obtenir des exécutables autonomes
- nécessite une certaine maîtrise du langage
- existe aussi en OCaml (voir plus tard)

`ocamlc` compilateur en ligne de code-octet

`ocamlrun` interprète de code-octet

`ocamlopt` compilateur en ligne de code natif

`js_of_ocaml` compilateur vers JavaScript

② **interpréter** :

- permet d'expérimenter avec le langage, et d'observer les effets des requêtes une par une
- plus adapté pour apprendre un langage

`ocaml` lance la boucle d'interprétation

Deux façons de travailler avec OCaml

① **compiler** (comme en Java, C, ...) :

- permet d'obtenir des exécutables autonomes
- nécessite une certaine maîtrise du langage
- existe aussi en OCaml (voir plus tard)

`ocamlc` compilateur en ligne de code-octet

`ocamlrun` interprète de code-octet

`ocamlopt` compilateur en ligne de code natif

`js_of_ocaml` compilateur vers JavaScript

② **interpréter** :

- permet d'expérimenter avec le langage, et d'observer les effets des requêtes une par une
- plus adapté pour apprendre un langage

`ocaml` lance la boucle d'interprétation

Comment lancer l'interpréteur ?

- Dans une shell: `ocaml`
- Mieux: lancer l'interpréteur avec un éditeur de ligne (comme `rlwrap` ou `ledit`)
`ledit ocaml`
- Encore mieux: dans `emacs` (ou `xemacs`, ou `aquamacs`): utiliser le mode `tuareg`

La boucle d'interprétation

- L'utilisateur tape une **requête** (on dit encore une **phrase**) OCaml : une **expression** terminée par deux points-virgules consécutifs.
- OCaml analyse la syntaxe, affiche un message d'erreur si cette syntaxe est incorrecte.
- Si la syntaxe est correcte, l'interpréteur **infère** (c'est-à-dire calcule) le **type** de l'expression, affiche un message d'erreur si l'expression est mal typée.
- Si l'expression est bien typée, l'interpréteur **évalue** l'expression, puis affiche le type calculé et la valeur obtenue.
- Boucle **Read-Eval-Print** introduite par LISP.

La boucle d'interprétation

- L'utilisateur tape une **requête** (on dit encore une **phrase**) OCaml : une **expression** terminée par deux points-virgules consécutifs.
- OCaml analyse la syntaxe, affiche un message d'erreur si cette syntaxe est incorrecte.
- Si la syntaxe est correcte, l'interpréteur **infère** (c'est-à-dire calcule) le **type** de l'expression, affiche un message d'erreur si l'expression est mal typée.
- Si l'expression est bien typée, l'interpréteur **évalue** l'expression, puis affiche le type calculé et la valeur obtenue.
- Boucle **Read-Eval-Print** introduite par LISP.

La boucle d'interprétation

- L'utilisateur tape une **requête** (on dit encore une **phrase**) OCaml : une **expression** terminée par deux points-virgules consécutifs.
- OCaml analyse la syntaxe, affiche un message d'erreur si cette syntaxe est incorrecte.
- Si la syntaxe est correcte, l'interpréteur **infère** (c'est-à-dire calcule) le **type** de l'expression, affiche un message d'erreur si l'expression est mal typée.
- Si l'expression est bien typée, l'interpréteur **évalue** l'expression, puis affiche le type calculé et la valeur obtenue.
- Boucle **Read-Eval-Print** introduite par LISP.

La boucle d'interprétation

- L'utilisateur tape une **requête** (on dit encore une **phrase**) OCaml : une **expression** terminée par deux points-virgules consécutifs.
- OCaml analyse la syntaxe, affiche un message d'erreur si cette syntaxe est incorrecte.
- Si la syntaxe est correcte, l'interpréteur **infère** (c'est-à-dire calcule) le **type** de l'expression, affiche un message d'erreur si l'expression est mal typée.
- Si l'expression est bien typée, l'interpréteur **évalue** l'expression, puis affiche le type calculé et la valeur obtenue.
- Boucle **Read-Eval-Print** introduite par LISP.

La boucle d'interprétation

- L'utilisateur tape une **requête** (on dit encore une **phrase**) OCaml : une **expression** terminée par deux points-virgules consécutifs.
- OCaml analyse la syntaxe, affiche un message d'erreur si cette syntaxe est incorrecte.
- Si la syntaxe est correcte, l'interpréteur **infère** (c'est-à-dire calcule) le **type** de l'expression, affiche un message d'erreur si l'expression est mal typée.
- Si l'expression est bien typée, l'interpréteur **évalue** l'expression, puis affiche le type calculé et la valeur obtenue.
- Boucle **Read-Eval-Print** introduite par LISP.

Exemples (code/examples1.ml)

(correct *)*

$3 \times (4 + 1) - 7;;$

(syntax error *)*

$17 +;;$

(erreur de typage *)*

$42 + \text{"hello"};;$

$2 + 3 \times 1;;$

$5 / 2;;$

$-5 \bmod 3;;$

Remarques

- C'est le ;; qui termine la requête, pas le saut de ligne. Les sauts de ligne à l'intérieur d'une requête sont des espaces comme les autres.
- Les ;; ne font pas partie de la syntaxe du langage *Caml* lui-même mais sont spécifiques à l'interpréteur.
- Commentaires : entre (* et *), éventuellement sur plusieurs lignes.

Remarques

- C'est le ;; qui termine la requête, pas le saut de ligne. Les sauts de ligne à l'intérieur d'une requête sont des espaces comme les autres.
- Les ;; ne font pas partie de la syntaxe du langage *Caml* lui-même mais sont spécifiques à l'interpréteur.
- Commentaires : entre (* et *), éventuellement sur plusieurs lignes.

Remarques

- C'est le ;; qui termine la requête, pas le saut de ligne. Les sauts de ligne à l'intérieur d'une requête sont des espaces comme les autres.
- Les ;; ne font pas partie de la syntaxe du langage *Caml* lui-même mais sont spécifiques à l'interpréteur.
- Commentaires : entre (* et *), éventuellement sur plusieurs lignes.

Premiers pas en OCaml

Premiers pas en OCaml

① types de base

- `int`, `float`, `bool`, `char`, `string`

② fonctions

- **function** `var` \rightarrow `expr`
- **fun** `var1` ... `varn` \rightarrow `expr`

③ déclaration des valeurs

- **let** `nom` = `expr`
- **let** `nom` = `expr1` **in** `expr2`
- **let** `f` `var1` ... `varn` = `expr`
- **let rec** `f` `var1` ... `varn` = `expr`
- **let** `nom1` = `expr1` **and** ... **and** `nomn` = `exprn`

Premiers pas en OCaml

① types de base

- `int`, `float`, `bool`, `char`, `string`

② fonctions

- **`function`** `var` \rightarrow `expr`
- **`fun`** `var1` ... `varn` \rightarrow `expr`

③ déclaration des valeurs

- **`let`** `nom` = `expr`
- **`let`** `nom` = `expr1` **`in`** `expr2`
- **`let`** `f` `var1` ... `varn` = `expr`
- **`let`** **`rec`** `f` `var1` ... `varn` = `expr`
- **`let`** `nom1` = `expr1` **`and`** ... **`and`** `nomn` = `exprn`

int

valeurs: $\dots, -2, -1, 0, 1, 2, 3, \dots$

- opérateurs:
- $+$: addition (infixe)
 - $-$: soustraction (infixe)
 - $*$: multiplication (infixe)
 - $/$: division entière (infixe)
 - mod : reste de la division entière (infixe)
 - \dots

int (examples)

```
# 3 + 4 * 2;;  
- : int = 11
```

```
# (3 + 4) * 2;;  
- : int = 14
```

```
# mod (3+4) 2;;  
  ^^^
```

Error: Syntax error

```
# (3+4) mod 2;;  
- : int = 1
```

float

valeurs: ..., -2.0, 3.14, 5e3, 6e-9 ...

opérateurs: arithmétiques

+, -, *, /.

attention ! opérateurs typés: sur float les opérateurs arithmétiques s'écrivent avec un point

réels

sin, sqrt, log, ceil, floor, ...

conversion: il y a des fonctions de conversion entre int et float

```
# float_of_int;;  
- : int -> float = <fun>  
# (float);;  
- : int -> float = <fun>  
# int_of_float;;  
- : float -> int = <fun>
```

float (examples)

```
# sin (2.0/.3.0);;  
- : float = 0.618369803069737  
# 3.0 +.2.5;;  
- : float = 5.5  
# 3.0 + 2.5;;  
^^^
```

Error: This expression has **type** float but an expression was expected **of type** int

```
# int_of_float 3.0 + int_of_float 2.5;;  
- : int = 5
```

```
# 3+.2.5;;  
^
```

Error: This expression has **type** int but an expression was expected **of type** float

```
# float_of_int 3 +. 2.5;;  
- : float = 5.5
```

float

valeurs: ..., -2.0, 3.14, 5e3, 6e-9 ...

opérateurs: arithmétiques

+, -, *, /.

attention ! opérateurs typés: sur float les opérateurs arithmétiques s'écrivent avec un point

réels

sin, sqrt, log, ceil, floor, ...

conversion: il y a des fonctions de conversion entre int et float

```
# float_of_int;;  
- : int -> float = <fun>  
# (float);;  
- : int -> float = <fun>  
# int_of_float;;  
- : float -> int = <fun>
```

float (examples)

```
# sin (2.0/.3.0);;  
- : float = 0.618369803069737  
# 3.0 +.2.5;;  
- : float = 5.5  
# 3.0 + 2.5;;  
  ^^^
```

Error: This expression has **type** float but an expression was expected **of type** int

```
# int_of_float 3.0 + int_of_float 2.5;;  
- : int = 5
```

```
# 3+.2.5;;  
  ^
```

Error: This expression has **type** int but an expression was expected **of type** float

```
# float_of_int 3 +. 2.5;;  
- : float = 5.5
```

bool

valeurs: true, false

opérateurs: logiques

- **not** : négation
- **&&**, **&** : et séquentiel (infixe)
- **||**, **or** : ou séquentiel (infixe)

comparaison

- **=** : égalité (infixe, à détaillé plus tard)
- **>**, **>=** : plus grand, plus grand ou égale (infixe)
- **<**, **<=** : plus petit, plus petit ou égale (infixe)

conditionnel: **if** cond **then** e1 **else** e2

- cond est une expression de type bool
- e1 et e2 sont deux expressions de même type, qui est aussi le type du conditionnel
- seulement une des deux branches e1 et e2 est évaluée.

bool (examples)

```
# not false && false;;  
- : bool = false  
# not (false && false);;  
- : bool = true  
# true = false;;  
- : bool = false  
# 3 = 3;;  
- : bool = true  
# 4 + 5 >= 10;;  
- : bool = false  
# 2.0 *. 4.0 >= 7.0;;  
- : bool = true  
  
# if (3<4) then 1 else 0;;  
- : int = 1  
# if (4<3) then 1 else 0;;  
- : int = 0
```


bool

valeurs: true, false

opérateurs: logiques

- **not** : négation
- **&&**, **&** : et séquentiel (infixe)
- **||**, **or** : ou séquentiel (infixe)

comparaison

- **=** : égalité (infixe, à détaillé plus tard)
- **>**, **>=** : plus grand, plus grand ou égale (infixe)
- **<**, **<=** : plus petit, plus petit ou égale (infixe)

conditionnel: **if** cond **then** e1 **else** e2

- cond est une expression de type bool
- e1 et e2 sont deux expressions de même type, qui est aussi le type du conditionnel
- seulement une des deux branches e1 et e2 est évaluée.

bool (examples)

```
# not false && false;;  
- : bool = false  
# not (false && false);;  
- : bool = true  
# true = false;;  
- : bool = false  
# 3 = 3;;  
- : bool = true  
# 4 + 5 >= 10;;  
- : bool = false  
# 2.0 *. 4.0 >= 7.0;;  
- : bool = true  
  
# if (3<4) then 1 else 0;;  
- : int = 1  
# if (4<3) then 1 else 0;;  
- : int = 0
```

char

valeurs: caractères ASCII (écrits entre apostrophes '
(*American Standard Code Information Interchange*)
'a', 'z', ' ', 'W'

échappement:

- `\\` : antislash (`\`)
- `\n` : saut de ligne (line feed)
- `\t` : tabulation
- `\ddd` : le caractère avec le code ASCII *ddd* en décimal
- `\'` : apostrophe (`'`)

fonct. convers.:

- `Char.code`: `char` \rightarrow `int`
- `Char.chr`: `int` \rightarrow `char`
- `Char.lowercase`: `char` \rightarrow `char`
- `Char.uppercase`: `char` \rightarrow `char`

Voir manual (module `Char`) pour une liste complète.

char (exemples)

```
# 'a';;  
- : char = 'a'
```

```
# Char.code 'a';;  
- : int = 97
```

```
# '\097';;  
- : char = 'a'
```

```
# '\97';;  
  ^^
```

Error: Illegal backslash escape in string or character (\9)

```
# Char.uppercase 'a';;  
- : char = 'A'
```

```
# Char.uppercase '[';;  
- : char = '['
```

string

valeurs: chaînes de caractères (écrites entre guillemets "
"Hello", "a", " ", "\097est a"

string \neq char: # "Hello".[1];;
- : char = 'e'

"Hell" ^ 'o' ;;
 ^^^

Error: This expression has **type** char but an
expression was expected **of type** string

concatenation: # "Hello" ^ "World" ;;
- : string = "HelloWorld"

autres fonct.:

- String.length: string \rightarrow int
- String.get: string \rightarrow int \rightarrow char
- String.make: int \rightarrow char \rightarrow string
- String.sub: string \rightarrow int \rightarrow int \rightarrow string

Voir manual (module String) pour une liste complète.

string (examples)

```
# "\097est_a";;  
- : string = "aest_a"
```

```
# "\097\"".[1];;  
- : char = '"'
```

```
# String.length "hello";;  
- : int = 5
```

```
# "hello".[0];;  
- : char = 'h'
```

```
# "hello".[5];;  
Exception: Invalid_argument "index_out_of_bounds".
```

```
# String.make 10 'a';;  
- : string = "aaaaaaaaaa"
```

Premiers pas en OCaml

① types de base

- `int`, `float`, `bool`, `char`, `string`

② fonctions

- **`function`** `var` \rightarrow `expr`
- **`fun`** `var1` ... `varn` \rightarrow `expr`

③ déclaration des valeurs

- **`let`** `nom` = `expr`
- **`let`** `nom` = `expr1` **`in`** `expr2`
- **`let`** `f` `var1` ... `varn` = `expr`
- **`let`** **`rec`** `f` `var1` ... `varn` = `expr`
- **`let`** `nom1` = `expr1` **`and`** ... **`and`** `nomn` = `exprn`

function var \rightarrow expr

- définit une fonction qui prend un argument, dénoté par le paramètre var, et qui renvoie comme résultat la valeur de l'expression expr.

```
# function x  $\rightarrow$  x+1;;  
- : int  $\rightarrow$  int = <fun>
```

- l'application d'une fonction à un argument évalue dans le résultat obtenu en remplaçant var dans expr avec la valeur de l'argument.

```
# (function x  $\rightarrow$  x+1)2;;  
- : int = 3
```

- un argument peut être une expression complexe:

```
# (function x  $\rightarrow$  x+1)(3+2-1);;  
- : int = 5  
# (function x  $\rightarrow$  x+1)((function x  $\rightarrow$  x+1)1);;  
- : int = 3
```

- attention, l'application associe à gauche:

```
# (function x  $\rightarrow$  x+1)(function x  $\rightarrow$  x+1)1;;  
      ^^^
```

Error: This **function** has **type** int \rightarrow int
It is applied **to** too many arguments.

function var \rightarrow expr (types)

- le type d'une fonction est $t1 \rightarrow t2$, où $t1$ est le type de l'argument et $t2$ est le type du résultat de la fonction.

```
# function x -> x+1;;  
- : int -> int = <fun>
```

- le type est déterminé en utilisant des informations dans l'expression expr, et peut être polymorphe. (\Rightarrow plus tard)

```
# function x -> x +. 1.;;  
- : float -> float = <fun>  
# function x -> x > 0.;;  
- : float -> bool = <fun>  
# function x -> x > 0;;  
- : int -> bool = <fun>  
# function x -> x;;  
- : 'a -> 'a = <fun>
```

- bien sur, l'application doit respecter le type de la fonction.

```
# (function x -> x+1)2. ;;  
      ^^^
```

Error: This expression has **type** float but an expression was expected **of type** int

et si on a plusieurs arguments ?

- on les aligne !

```
# function x -> function y -> x+y;;  
- : int -> int -> int = <fun>
```

```
# (function x -> function y -> x+y)2 3;;  
- : int = 5
```

- OCaml fournit aussi une syntaxe simplifiée:

fun var1 ... varn -> expr

est équivalent à

function var1 -> ... **function** varn -> expr

```
# fun x y -> x+y;;  
- : int -> int -> int = <fun>
```

```
# (fun x y -> x+y)2 3;;  
- : int = 5
```

les fonctions: données de première classe !

- l'argument d'une fonction peut être une fonction.

```
# function f -> f(f(2));;  
- : (int -> int) -> int = <fun>  
# (function f -> f(f(2)))(function x -> x+1);;  
- : int = 4
```

- l'implication associe à droite: attention aux parenthèses !

```
# fun x y -> x + y;;  
- : int -> int -> int = <fun>
```

- le résultat de l'évaluation d'une fonction peut être une autre fonction (évaluation partielle)

```
# (fun x y -> x + y) 1;;  
- : int -> int = <fun>
```

Examples

```
# fun x -> x*2;;  
- : int -> int = <fun>  
# function x -> x*2;;  
- : int -> int = <fun>  
# fun x y -> x*y;;  
- : int -> int -> int = <fun>  
# (fun x -> x*2) 3;;  
- : int = 6  
# (fun x y -> x*y) 3;;  
- : int -> int = <fun>  
# (fun x y -> x*y) 3 2;;  
- : int = 6  
# fun f -> (f(f2));;  
- : (int -> int) -> int = <fun>  
# (fun f x -> f (f x)) (fun x -> x*x);;  
- : int -> int = <fun>  
# (fun f x -> f (f x)) (fun x -> x*x) 2;;  
- : int = 16
```

Premiers pas en OCaml

① types de base

- `int`, `float`, `bool`, `char`, `string`

② fonctions

- **function** `var` \rightarrow `expr`
- **fun** `var1` ... `varn` \rightarrow `expr`

③ déclaration des valeurs

- **let** `nom` = `expr`
- **let** `nom` = `expr1` **in** `expr2`
- **let** `f` `var1` ... `varn` = `expr`
- **let rec** `f` `var1` ... `varn` = `expr`
- **let** `nom1` = `expr1` **and** ... **and** `nomn` = `exprn`

let nom = expr

- est une déclaration qui associe à nom la valeur de l'expression expr, qui peut donc être réutilisé dans les expressions suivantes:

```
# let x = 2 + 3;;  
val x : int = 5  
# x * 3;;  
- : int = 15  
# let y = x+1;;  
val y : int = 6
```

- attention, le nom est associé à la valeur et pas à l'expression:

```
# let z = 6-1;;  
val z : int = 5  
# x = z;;  
- : bool = true
```

- une déclaration peut être écrasée par une autre déclaration:

```
#let x = 0;;  
val x : int = 0  
# (*attention, la valeur de y ne change pas*)  
y;;  
- : int = 6  
# let z = x+1;;  
val z : int = 1
```

let nom = expr1 in expr2

- permet une déclaration locale de **nom** à l'intérieur de **expr2**:

```
# let x = 3 in x+4;;  
- : int = 7  
# let y=x+4;;  
    ^^^
```

Error: Unbound **value** x

- un nom déclaré par **let** (ou **let rec**) est par contre connu dans toutes les expressions qui suivent la déclaration:

```
# let x = 3;;  
val x : int = 3  
# x+4;;  
- : int = 7  
# let y=x+4;;  
val y : int = 7
```

- une déclaration locale peut redéfinir localement un nom global

```
# let x = 2;;  
val x : int = 2  
# let x = 3 in x;;  
- : int = 3  
# x;;  
- : int = 2
```

Visibilité des liaisons

```
let x = 1;;  
:  
let x = 2 in  
  :  
  let x = 3 in  
    :  
    :  
  :  
  :  
:
```

Diagram illustrating variable binding and visibility in a nested `let` expression. The code defines `x` at three different scopes. To the right of the code, curly braces indicate the scope of each binding, with the value of `x` shown next to the brace. The innermost binding (`x = 3`) is visible in the scope it defines and the scope immediately above it. The middle binding (`x = 2`) is visible in the scope it defines and the scope immediately above it. The outermost binding (`x = 1`) is visible in the scope it defines and the scope immediately above it.

Seulement la liaison la plus locale est visible.

Déclarer les fonctions

- on peut donner, en particulier, un nom aux fonctions

```
# let f = (fun x -> x * 2);;  
val f : int -> int = <fun>  
# f 3;;  
- : int = 6  
# let g = fun y -> let f = fun x -> x*x in f (f y);;  
val g : int -> int = <fun>  
# g 2;;  
- : int = 16
```

- OCaml fournit aussi une syntaxe simplifiée:

`let nom var1 ... varn = expr`

est équivalent à

`let nom = fun var1 ... varn -> expr`

```
# let f x = x * 2;;  
val f : int -> int = <fun>  
# let g y = let f x = x*x in f (f y);;  
val g : int -> int = <fun>  
# g (f 2);;  
- : int = 256
```

Déclarer les fonctions

- on peut donner, en particulier, un nom aux fonctions

```
# let f = (fun x -> x * 2);;  
val f : int -> int = <fun>  
# f 3;;  
- : int = 6  
# let g = fun y -> let f = fun x -> x*x in f (f y);;  
val g : int -> int = <fun>  
# g 2;;  
- : int = 16
```

- OCaml fournit aussi une syntaxe simplifiée:

let nom var1 ... varn = expr

est équivalent à

let nom = **fun** var1 ... varn -> expr

```
# let f x = x * 2;;  
val f : int -> int = <fun>  
# let g y = let f x = x*x in f (f y);;  
val g : int -> int = <fun>  
# g (f 2);;  
- : int = 256
```

let rec nom = fun arg1 arg 2... -> expr

- permet une définition récursive d'une fonction nom, c.à-d. nom peut être utilisé dans expr:

```
# let rec fact = fun x -> if (x=0) then 1 else x*fact(x-1);;  
val fact : int -> int = <fun>
```

- si on utilise simplement **let** on obtient un error:

```
# let fact = fun x -> if (x=0) then 1 else x*fact(x-1);;  
                                     ^^^  
Error: Unbound value fact
```

let définit en effet fact dans toutes les expressions qui suivent la déclaration **mais** pas dans l'expression à droite de **let fact = fun x ->...**

- On peut utiliser la syntaxe simplifiée, comme pour **let**

```
# let rec fact x = if (x=0) then 1 else x*fact(x-1);;  
val fact : int -> int = <fun>
```

let nom = expr1 and nom = expr2

- permet la définition simultanée de plusieurs expressions, séparées par le mot clef **and**:

```
# let a = 3 and b = 3*2 and c = 2.0;;  
val a : int = 3  
val b : int = 6  
val c : float = 2.
```

- En particulier utile pour les définitions de fonctions récursives:

```
let rec even x =  
  if (x=0) then true else odd (x-1)  
and odd x =  
  if (x=0) then false else even (x-1);;  
val even : int -> bool = <fun>  
val odd : int -> bool = <fun>
```

```
# even 4;;  
- : bool = true
```

- en effet, l'évaluation de even 4 enchaîne une alternance d'évaluations:

even 4 \mapsto odd 3 \mapsto even 2 \mapsto odd 1 \mapsto even 0 \mapsto true

Utilité **let in**

Une déclaration locale permet une majeure efficacité quand on appelle plusieurs fois la même expression.

- la définition suivante est très inefficace car elle évalue deux fois $\text{exp } (x/2)$ dans chaque branche du cas $x > 0$:

```
# let rec exp x =  
  if (x = 0) then 1  
  else if (x mod 2 = 1) then (exp (x/2)) * (exp (x/2)) * 2  
    else (exp (x/2)) * (exp (x/2));;  
val exp : int -> int = <fun>
```

- Pour éviter ce type de problèmes on peut définir des noms locaux

```
# let rec exp x =  
  if (x = 0) then 1  
  else let h = exp (x/2) in  
    if (x mod 2 = 1) then h * h * 2  
    else h * h;;  
val exp : int -> int = <fun>
```

Utilité **let in**

Une déclaration locale permet une majeure efficacité quand on appelle plusieurs fois la même expression.

- la définition suivante est très inefficace car elle évalue deux fois $\text{exp } (x/2)$ dans chaque branche du cas $x > 0$:

```
# let rec exp x =  
  if (x = 0) then 1  
  else if (x mod 2 = 1) then (exp (x/2)) * (exp (x/2)) * 2  
    else (exp (x/2)) * (exp (x/2));;  
val exp : int -> int = <fun>
```

- Pour éviter ce type de problèmes on peut définir des noms locaux

```
# let rec exp x =  
  if (x = 0) then 1  
  else let h = exp (x/2) in  
    if (x mod 2 = 1) then h * h * 2  
    else h * h;;  
val exp : int -> int = <fun>
```