INTRODUCTION À OCAML

slides copyright 2017, 2018 David Walker, Amy Felty
permission granted to reuse these slides for non-commercial educational purposes

Alonzo Church, 1903-1995

En 1936, Alonzo Church a inventé le lambda calcul. Il l'a appelé une logique, mais c'était un langage fonctionnel pure - le premier langage de programmation au monde.

Il a dit:

« Il peut y avoir d'autres applications du système que son utilisation en tant que logique. »

"*There may, indeed, be other applications of the system than its use as a logic.*"

En 1936, Alonzo Church a

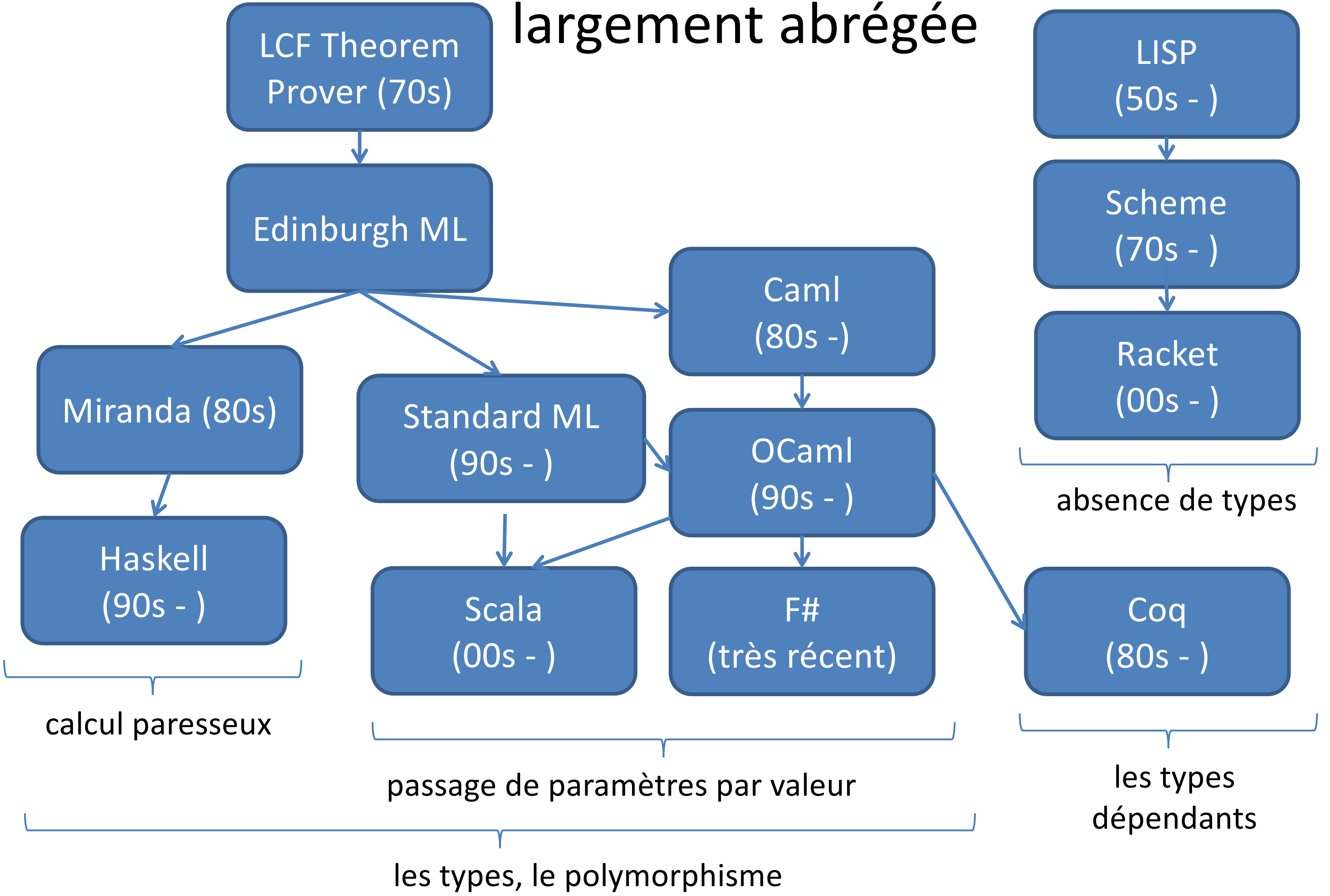
Un peu trop modeste?
Or the greatest technological understatement of the 20th century?

Alonzo Church, 1903-1995

« Il peut y avo ’autres applications du système que son utilisation en tant que logique. »

"*There may, indeed, be other applications of the system than its use as a logic.*"

Généalogie de programmation fonctionelle largement abrégée



Généalogie de programmation fonctionelle largement abrégée

- LCF Theorem Prover (70s) → Edinburgh ML
- Edinburgh ML → Miranda (80s), Standard ML (90s -), Caml (80s -)
- Miranda (80s) → Haskell (90s -)
- Caml (80s -) → OCaml (90s -)
- Standard ML (90s -) → OCaml (90s -), Scala (00s -)
- OCaml (90s -) → Scala (00s -), F# (très récent), Coq (80s -)
- LISP (50s -) → Scheme (70s -) → Racket (00s -)

Groupes :

- calcul paresseux : Haskell (90s -)
- passage de paramètres par valeur : Scala (00s -), F# (très récent)
- les types, le polymorphisme : Haskell, Scala, F#
- absence de types : LISP, Scheme, Racket
- les types dépendants : Coq (80s -)

Langages fonctionnels: qui les utilise?

map-reduce dans leurs centres de données

Scala pour la certification, la maintenance, la flexibilité

Erlang pour la concurrence, Haskell pour la gestion de PHP

F# utilisé dans Visual Studio

les mathématiciens

re-preuve du théorème 4 couleurs en Coq

Haskell pour synthétiser du matériel

Haskell pour la finance

www.artima.com/scalazine/articles/twitter_on_scala.html
www.janestreet.com/technology/
msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/visual-fsharp
ai.google/research/pubs/pub62
www.haskell.org/haskellwiki/Haskell_in_industry
blogs.wsj.com/cio/2014/02/24/facebook-whatsapp-messaging-service-written-in-exotic-erlang/

Langages fonctionnels: qui les utilise?

Des éléments de programmation fonctionnelle apparaissent partout

- F# utilisé dans Microsoft Visual Studio
- Scala combine ML (un langage fonctionnel) avec des objets
  - exécute sur la JVM
- C# inclut “delegates”
  - delegates == fonctions
- Python inclut les “lambdas”
  - lambdas == encore, les fonctions
- Javascript
  - On peut trouver des tutoriels en ligne sur l'utilisation des techniques de programmation fonctionnelle afin d’écrire du code plus élégant
- C++ les modules pours map-reduce
  - a permis le parallélisme fonctionnel à Google
- Java contient les « generics » et la ramassage de miettes
- ...

Révision de la programmation fonctionnelle

Penser en termes de programmation fonctionnelle

En Java ou C, la plupart du travail se fait en *changeant* quelque chose

```
temp = pair.x;
pair.x = pair.y;
pair.y = temp;
```

les commandes *modifient* une structure de données existante (comme une paire)

En ML, la plupart du travail se fait en *produisant de nouvelles données*

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

On *analyse* les données existantes (comme la paire) et on *produit* de nouvelles données (y, x)

Penser en termes de programmation fonctionnelle

code fonctionnel :

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

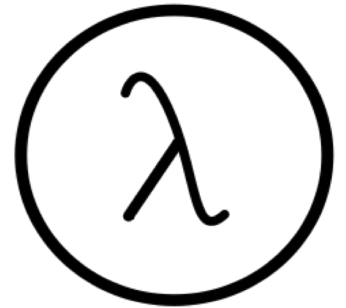
- *la sortie est très importante!*
- *la sortie est <u>une fonction</u> de l'entrée*
- *les propriétés des données sont stables*
- *répétable*
- *le parallélisme est apparent*
- *plus facile à tester*
- *plus facile à composer*

code imperatif :

```
temp = pair.x;
pair.x = pair.y;
pair.y = temp;
```

- *la sortie n'est pas pertinente!*
- *la sortie n'est pas une fonction de l'entrée*
- *les propriétés des données changent*
- *pas répétable*
- *le parallélisme n'est pas apparent*
- *plus difficile à tester*
- *plus difficile à composer*

Présentation des fonctionnalités d'OCaml



Petit noyau basé sur le *lambda calcul*.

- Le contrôle est basé sur des fonctions (récursives).
- Au lieu de for-loops, while-loops, do-loops, iterators, etc.
  - peut être défini à partir de primitives
- Facilite la définition de la sémantique

Procédures *« première classes », de portée lexicale et d'ordre supérieur*

- les fonctions « première classes » ou les clôtures ou les « lambdas »
- « première classes » : les fonctions sont des valeurs de données comme toute autre valeur de données
  - comme les nombres, elles peuvent être stockées, définies de manière anonyme,...
- portée lexicale : signification des variables est déterminée statiquement
- d'ordre supérieur : fonctions utilisées comme arguments et résultats
  - programmes comme arguments aux programmes; généré à partir de programmes

Ces fonctionnalités se trouvent également dans Racket, Haskell, SML, F#, Clojure, ...

Présentation des fonctionnalités d'OCaml

Typé statiquement: aide au débogage et aux tests

- Le compilateur détecte de nombreuses erreurs simples avant de pouvoir exécuter le code.
    - Un type vaut mille tests (à partir de 6h20):
        - https://www.youtube.com/watch?v=q1Yi-WM7XqQ
- Java est également fortement typé statiquement.
- Scheme, Python, Javascript, etc. sont tous fortement *typés dynamiquement*, les erreurs de type sont découvertes pendant l'exécution du code

Fortement typé : le compilateur applique l'abstraction de type.

- ne peut pas convertir un entier en un enregistrement, une fonction, une chaîne, etc.
- C/C++ sont des langages *faiblement typés* (statiquement typés). Le compilateur vous permet de faire quelque chose d'intelligent (*plus souvent stupide*).

Inférence de types : le compilateur déduit les types automatiquement

Exécution d'OCaml

- OCaml est installé avec des compilateurs:
  - "ocamlc" – compilateur rapide de bytecode
  - "ocamlopt" – fait des optimisations, compilateur de code natif
  - "ocamlbuild – un « wrapper » qui calcule les dépendances (pas utilisé dans ce cours)
- Un shell interactif (interpréteur):
  - utile pour essayer du code et pour déboguer de petits programmes
  - "ocaml"
  - *Vous utiliserez probablement cet interpréteur la plupart du temps*
- Bien d'autres outils
  - par exemple, un débogueur, un générateur de dépendance, un profileur, etc.
- Nous allons l'utiliser via VCL. Vous pouvez essayer de l'installer vous-même.
  - Voir OCaml.org

Les éditeurs

- De nombreuses options:
  - Emacs
    - ce que je vais utiliser en classe et ce qui sera utilisé dans les laboratoires
    - bon support pour OCaml avec le mode Emacs « Touareg »
    - puissant éditeur de texte, peut exécuter OCaml
    - extensions écrites en elisp - un langage fonctionnel!)
  - OCaml IDE
    - environnement de développement intégré écrit en Ocaml
    - Je ne l'ai pas utilisé, donc je ne peux pas commenter.
  - Eclipse
    - Je ne l'ai pas essayé mais d'autres le recommandent.
  - Sublime, atom
    - Je ne l'ai pas essayé, mais souvent utilisé par les étudiants.

UNE INTRODUCTION: QUELQUES EXEMPLES DE PROGRAMMES

Compilateur et interprète OCaml

- Demo:
  - emacs
  - les fichiers .ml
  - des programmes simples: bonjour.ml, sumTo.ml
  - débogage simple et tests unitaires
  - Compilateur ocamlc

Un premier programme OCaml

`bonjour.ml:`

```
print_string ”Bonjour CSI 3520!!\n";;
```

Un premier programme OCaml

bonjour.ml:

```
print_string ”Bonjour CSI 3520!!\n"
```

une fonction

son argument: une chaîne inclus dans "..."

un programme peut être rien plus qu’une seule expression (mais c'est rare)

pas de parentheses, appelle une fonction comme ci-dessous :

```
f arg
```

(les parentheses sont seulement utilisés pour le groupement, la priorité, si nécessaire)

Un premier programme OCaml

bonjour.ml:

```
print_string ”Bonjour CSI 3520!!\n"
```

l'interprète appliqué à bonjour.ml:

```
$ ocaml
         Objective Caml Version 4.05.0
# 3 + 1;;
- : int = 4
#
```

Un premier programme OCaml

bonjour.ml:

```
print_string ”Bonjour CSI 3520!!\n"
```

l'interprète appliqué à bonjour.ml:

```
$ ocaml
        Objective Caml Version 4.05.0
# 3 + 1;;
- : int = 4
# #use ”bonjour.ml";;
Bonjour CSI 3520!!
- : unit = ()
#
```

Un premier programme OCaml

bonjour.ml:

```
print_string ”Bonjour CSI 3520!!\n"
```

l'interprète appliqué à bonjour.ml:

```
$ ocaml
          Objective Caml Version 4.05.0
# 3 + 1;;
- : int = 4
# #use ”bonjour.ml";;
Bonjour CSI 3520!!
- : unit = ()
# #quit;;
$
```

l'interpréteur OCaml

Un premier programme OCaml

bonjour.ml:

```
print_string "Bonjour CSI 3520!!\n"
```

compilation et exécution de bonjour.ml:

```
$ ocamlc bonjour.ml
$
```

Un premier programme OCaml

bonjour.ml:

```
print_string ”Bonjour CSI 3520!!\n"
```

compilation et exécution de bonjour.ml:

```
$ ocamlc bonjour.ml
$ ocaml
        Objective Caml Version 4.05.0
#
```

Un premier programme OCaml

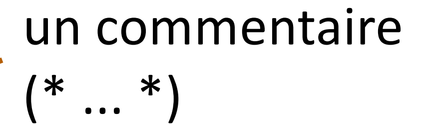
bonjour.ml:

```
print_string ”Bonjour CSI 3520!!\n"
```

compilation et exécution de bonjour.ml:

```
$ ocamlc bonjour.ml
$ ocaml
        Objective Caml Version 4.05.0
# #load ”bonjour.cmo";;
Bonjour CSI 3520!!
#
```

Un deuxième programme OCaml



sumTo.ml:

```
(* additionner les nombres de 0 à n

*)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)
let _ =
  print_int (sumTo 8);
  print_newline()
```

Un deuxième programme OCaml

le nom de la fonction définie

sumTo.ml:

```
(* additionner les nombres de 0 à n

*)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)
let _ =
  print_int (sumTo 8);
  print_newline()
```

le mot-clé «let» commence une définition; le mot clé «rec» indique la récursion

Un deuxième programme OCaml

sumTo.ml:

```
(* additionner les nombres de 0 à n

*)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)
let _ =
  print_int (sumTo 8);
  print_newline()
```

type de résultat int

argument nommé n a le type int

Un deuxième programme OCaml

sumTo.ml:

```
(* additionner les nombres de 0 à n

*)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)
let _ =
  print_int (sumTo 8);
  print_newline()
```

Un deuxième programme OCaml

Chaque branche de l'instruction conditionnelle construit un résultat

sumTo.ml:

```
(* additionner les nombres de 0 à n

*)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)
let _ =
  print_int (sumTo 8);
  print_newline()
```

construction du résultat 0

construction du résultat en utilisant un appel recursif à sumTo

Un deuxième programme OCaml

sumTo.ml:

```
(* additionner les nombres de 0 à n

*)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)
let _ =
  print_int (sumTo 8);
  print_newline()
```

imprimer le résultat de l'appel sumTo sur argument 8

imprimer une nouvelle ligne

Un deuxième programme Ocaml (Nouvelle Version)

newSumTo.ml:

```
(* additionner les nombres de 0 à n
   pré condition : n doit être un nombre
   naturel
*)
let rec sumTo (n:int) : int =
  match n with
    0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
```

Un deuxième programme Ocaml (Nouvelle Version)

déconstruire la valeur n
en utilisant le filtrage sur l'argument
« pattern matching »

newSumTo.ml:

```
(* additionner les nombres de 0 à n
   pré condition : n doit être un nombre
   naturel
*)
let rec sumTo (n:int) : int =
  match n with
    0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
```

les données à
déconstruire
apparaît
entre
mots clés
« match » et
« with »

Un deuxième programme Ocaml (Nouvelle Version)

barre verticale "|" sépare les alternatifs du filtrage

newSumTo.ml:

```
(* additionner les nombres de 0 à n
   pré condition : n doit être un nombre
   naturel
*)
let rec sumTo (n:int) : int =
  match n with
    0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
```

les données déconstruites correspondent à l'un des 2 cas:
(i) un « match » à 0 ou (ii) un « match » au variable n

Un deuxième programme Ocaml (Nouvelle Version)

Chaque branche de l'instruction « match » construit un résultat

newSumTo.ml:

```
(* additionner les nombres de 0 à n
   pré condition : n doit être un nombre
   naturel
let rec sumTo (n:int) : int =
  match n with
    0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
```

comme avant, construction du résultat 0

comme avant, construction du résultat en utilisant un appel recursif à sumTo

LES BASES D'OCAML: EXPRESSIONS, VALEURS, TYPES SIMPLES

Terminologie: expressions, valeurs, types

- Les expressions sont des calculs
  - 2 + 3 est un calcul
- Les valeurs sont les résultats de calculs
  - 5 est un valeur
- Les types décrivent des collections de valeurs et les calculs qui génèrent ces valeurs
  - int est un type
  - les valeurs de type int incluent
    - 0, 1, 2, 3, …, max_int
    - -1, -2, …, min_int

Quelques types simples, valeurs, expressions

Type:	Valeurs:	Expressions:
`int`	`-2, 0, 42`	`42 * (13 + 1)`
`float`	`3.14, -1., 2e12`	`(3.14 +. 12.0) *. 10e6`
`char`	`'a', 'b', '&'`	`int_of_char 'a'`
`string`	`"moo", "cow"`	`"moo" ^ "cow"`
`bool`	`true, false`	`if true then 3 else 4`
`unit`	`()`	`print_int 3`

Pour plus de types et de fonctions primitifs, consultez le manuel de référence OCaml:

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html

Toutes les expressions n'ont pas de valeur

Expression:

`42 * (13 + 1)`	**évalue à**	`588`
`(3.14 +. 12.0) *. 10e6`	↦	`151400000.`
`int_of_char ’a’`	↦	`97`
`"moo" ^ "cow"`	↦	`“moocow"`
`if true then 3 else 4`	↦	`3`
`print_int 3`	↦	`()`

`1 + ”bonjour"` **n'évalue pas!**

LES BASES D'OCAML: SYNTAXE DES EXPRESSION DE BASE

Syntaxe des expressions de base

Les expressions OCaml les plus simples e sont:

- valeurs — *nombres, chaînes de caractère booléens, ...*
- id — *variables (x, foo, ...)*
- e_1 op e_2 — *les opérateurs (x+3, ...)*
- id e_1 e_2 … e_n — *appel de fonction (foo 3 42)*
- **let** id = e_1 **in** e_2 — *déclaration de variable locale*
- **if** e_1 **then** e_2 **else** e_3 — *une expression conditionnelle*
- (e) — *une expression entre parenthèses*
  (e : t) — *une expression avec son type*

Une remarque sur les parenthèses

Dans la plupart des langues, les arguments sont entre parenthèses et séparés par des virgules :

```
f(x,y,z)      sum(3,4,5)
```

En OCaml, on n'écrit pas les parenthèses ni les virgules:

```
f x y z       sum 3 4 5
```

Mais regroupement est important. Par exemple,

```
f x y z
f x (y z)
```

Dans le premier, f a 3 arguments (x, y et z).

Dans la seconde, f a 2 arguments (x et le résultat de l'application de la fonction y à z.)