Programmation Fonctionnelle
Cours 01

Michele Pagani

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

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

18 septembre 2014

Organisation

- Intervenants:

  CM: Michele Pagani,
  michele.pagani@pps.univ-paris-diderot.fr,
  jeudi 10h30-12h30, Amphi 1A

  TD Groupe 1: Antonio Bucciarelli,
  buccia@pps.univ-paris-diderot.fr,
  mardi 8h30-10h30, Salle 2031

  TD Groupe 2: Peter Habermehl,
  peter.habermehl@liafa.univ-paris-diderot.fr,
  lundi 8h30-10h30, Salle 2032

  TD Groupe 3: Juliusz Chroboczek,
  jch@pps.univ-paris-diderot.fr,
  mardi 8h30-10h30, Salle 2001

- Période des examens : en janvier
- Il y a un projet de programmation, mais pas de partiel

Organisation

- Intervenants:

  CM: Michele Pagani,
  michele.pagani@pps.univ-paris-diderot.fr,
  jeudi 10h30-12h30, Amphi 1A

  TD Groupe 1: Antonio Bucciarelli,
  buccia@pps.univ-paris-diderot.fr,
  mardi 8h30-10h30, Salle 2031

  TD Groupe 2: Peter Habermehl,
  peter.habermehl@liafa.univ-paris-diderot.fr,
  lundi 8h30-10h30, Salle 2032

  TD Groupe 3: Juliusz Chroboczek,
  jch@pps.univ-paris-diderot.fr,
  mardi 8h30-10h30, Salle 2001
- Période des examens : en janvier
- Il y a un projet de programmation, mais pas de partiel

Organisation

- Intervenants:

  CM: Michele Pagani,
  `michele.pagani@pps.univ-paris-diderot.fr`,
  jeudi 10h30-12h30, Amphi 1A

  TD Groupe 1: Antonio Bucciarelli,
  `buccia@pps.univ-paris-diderot.fr`,
  mardi 8h30-10h30, Salle 2031

  TD Groupe 2: Peter Habermehl,
  `peter.habermehl@liafa.univ-paris-diderot.fr`,
  lundi 8h30-10h30, Salle 2032

  TD Groupe 3: Juliusz Chroboczek,
  `jch@pps.univ-paris-diderot.fr`,
  mardi 8h30-10h30, Salle 2001

- Période des examens : en janvier
- Il y a un projet de programmation, mais pas de partiel

Contrôle de connaissances

- Première session :

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

- Deuxième session :

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

Le projet

- ???
- À faire en binôme, mais. . .

  attention: soutenance et note individuelles !
- Plagiats = absence à la premiere session !
- Plus sur l'organisation du projet : voir les TP

Le projet

- ???
- À faire en binôme, mais. . .

  attention: soutenance et note individuelles !
- Plagiats = absence à la premiere session !
- Plus sur l'organisation du projet : voir les TP

Le projet

- ???
- À faire en binôme, mais. . .

  attention: soutenance et note individuelles !
- Plagiats = absence à la premiere session !
- Plus sur l'organisation du projet : voir les TP

Le projet

- ???
- À faire en binôme, mais. . .

  attention: soutenance et note individuelles !
- Plagiats = absence à la premiere session !
- Plus sur l'organisation du projet : voir les TP

Organisation

- Page web: sur DidEL: PF521 - 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 la page web du cours)

Organisation

- Page web: sur DidEL: PF521 - 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 la page web du cours)

Qu'est-ce que la programmation fonctionnelle ?

Style impératif

```
1  static void qSort(int tab[], int start, int end)
2   {
3    int pivindex;
4    if (start>=end) return;
5
6    pivindex=split(tab[start], start, end, tab);
7
8    qSort(tab, start, pivindex);
9    qSort(tab, pivindex+1, end);
10  }
11
12 static int split(int pivot, int start, int end,int tab[])
13  {
14   int store = start;
15   swap (tab, start, end);
16   for (int i=start; i<end; i++){
17    if (tab[i]<pivot){
18     swap (tab,i,store);
19     store++;
20    }
21   }
22   swap (tab,store,end);  //start..store < pivot <+ store..end
23   return store;
24  }
```

Style fonctionnel

```
let rec qSort list =
 match list with
 | [] -> []
 | pivot::rest ->
  split pivot [] [] rest
and split pivot left right list =
 match list with
 | [] -> (qSort left)@( pivot :: (qSort right))
 | hd :: tl ->
  if hd <= pivot then split pivot (hd :: left) right tl
  else split pivot left (hd :: right) tl;;
```

Les deux styles face à face

Impératif

- programme: séquence structurée d'instructions à la machine
- variable: nom pour un emplacement mémoire qui contient une valeur
- affectation:

  x:=2+1;

  change la valeur de la case de mémoire associée à x
- usage intensif de l'itération

Fonctionel

- programme: définition d'une fonction à partir des fonctions plus élémentaires
- variable: comme en mathématiques, une inconnue sur un ensemble des valeurs
- application des fonctions:

```
(fun x -> x+1)2;;
```

  expression représentant la valeur 3
- usage intensif de de la récursion

Les deux styles face à face

Impératif

- programme: séquence structurée d'instructions à la machine
- variable: nom pour un emplacement mémoire qui contient une valeur
- affectation:

  x:=2+1;

  change la valeur de la case de mémoire associée à x
- usage intensif de l'itération

Fonctionel

- programme: définition d'une fonction à partir des fonctions plus élémentaires
- variable: comme en mathématiques, une inconnue sur un ensemble des valeurs
- application des fonctions:

  ```
  (fun x -> x+1)2;;
  ```

  expression représentant la valeur 3
- usage intensif de de la récursion

Les deux styles face à face

Impératif

- programme: séquence structurée d'instructions à la machine
- variable: nom pour un emplacement mémoire qui contient une valeur
- affectation:

  ```
  x:=2+1;
  ```

  change la valeur de la case de mémoire associée à x
- usage intensif de l'itération

Fonctionel

- programme: définition d'une fonction à partir des fonctions plus élémentaires
- variable: comme en mathématiques, une inconnue sur un ensemble des valeurs
- application des fonctions:

  ```
  (fun x -> x+1)2;;
  ```

  expression représentant la valeur 3
- usage intensif de de la récursion

Les deux styles face à face

Impératif

- programme: séquence structurée d'instructions à la machine
- variable: nom pour un emplacement mémoire qui contient une valeur
- affectation:

  ```
  x:=2+1;
  ```

  change la valeur de la case de mémoire associée à x
- usage intensif de l'itération

Fonctionel

- programme: définition d'une fonction à partir des fonctions plus élémentaires
- variable: comme en mathématiques, une inconnue sur un ensemble des valeurs
- application des fonctions:

  ```
  (fun x -> x+1)2;;
  ```

  expression représentant la valeur 3
- usage intensif de de la récursion

Pour quoi la programmation fonctionnelle ?

> *Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function.*
>
> John Carmack

- 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 verification (même automatique) des comportements des programmes
- ... et cela souvent sans perte d'efficacité

Pour quoi la programmation fonctionnelle ?

> *Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function.*
>
> John Carmack

- 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 verification (même automatique) des comportements des programmes
- ... et cela souvent sans perte d'efficacité

Pour quoi la programmation fonctionnelle ?

> *Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function.*
>
> John Carmack

- 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 verification (même automatique) des comportements des programmes
- ... et cela souvent sans perte d'efficacité

Pour quoi la programmation fonctionnelle ?

> *Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function.*
>
> John Carmack

- 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 verification (même automatique) des comportements des programmes
- ... et cela souvent sans perte d'efficacité

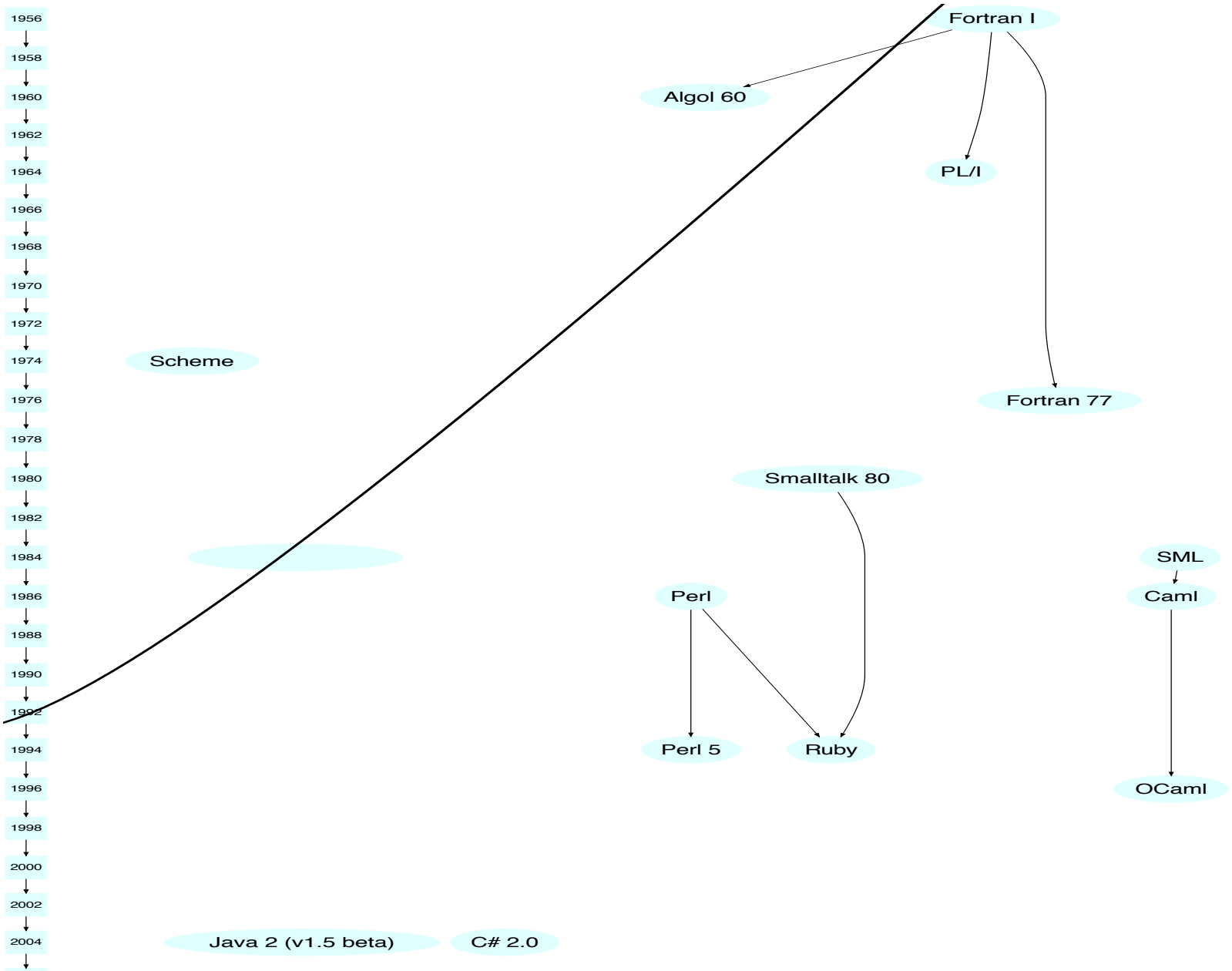
Pour quoi la programmation fonctionnelle ?

> *Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function.*
>
> John Carmack

- 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 verification (même automatique) des comportements des programmes
- ... et cela souvent sans perte d'efficacité

Le langage OCaml

Où on est


1956
1958
1960
1962
1964
1966
1968
1970
1972
1974
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
2004
2006
Fortran I
Algol 60
PL/I
Fortran 77
Scheme
Smalltalk 80
SML
Caml
Perl
Perl 5
Ruby
OCaml
Java 2 (v1.5 beta)
C# 2.0

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, librairies partagées, etc.

2003 Modules récursifs, private types, etc.

Qui utilise OCaml?

Enseignement

- ici, par exemple !

Recherche

- vérification: Coq, Astrée, SLAM,...
- outils pour le web: Ocsigen, Mirage,...

Communauté

- Marionnet, Unison, MLDonkey, ...

Industrie

- Citrix, Dassault, Esterel, Lexifi, Jane Street Capital, OcamlPro, Baretta, Merjis, RedHat, ...

Qui utilise OCaml?

Enseignement

- ici, par exemple !

Recherche

- vérification: Coq, Astrée, SLAM,...
- outils pour le web: Ocsigen, Mirage,...

Communauté

- Marionnet, Unison, MLDonkey, ...

Industrie

- Citrix, Dassault, Esterel, Lexifi, Jane Street Capital, OcamlPro, Baretta, Merjis, RedHat, ...

Qui utilise OCaml?

Enseignement

- ici, par exemple !

Recherche

- vérification: Coq, Astrée, SLAM,...
- outils pour le web: Ocsigen, Mirage,...

Communauté

- Marionnet, Unison, MLDonkey, ...

Industrie

- Citrix, Dassault, Esterel, Lexifi, Jane Street Capital, OcamlPro, Baretta, Merjis, RedHat, ...

Qui utilise OCaml?

Enseignement

- ici, par exemple !

Recherche

- vérification: Coq, Astrée, SLAM,...
- outils pour le web: Ocsigen, Mirage,...

Communauté

- Marionnet, Unison, MLDonkey, ...

Industrie

- Citrix, Dassault, Esterel, Lexifi, Jane Street Capital, OcamlPro, Baretta, Merjis, RedHat, ...

Pourquoi utiliser 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

Pourquoi utiliser 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

Pourquoi utiliser 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

Pourquoi utiliser 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

Bibliographie

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

- Xavier Leroy et al :
  The Objective Caml system
  http://caml.inria.fr/pub/docs/manual-ocaml/

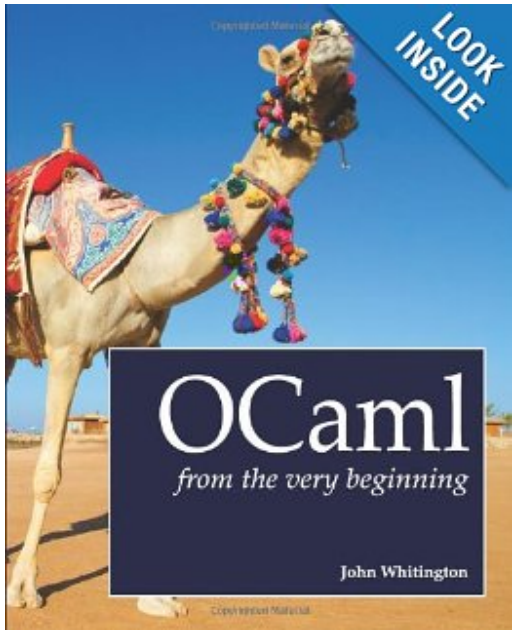
- Emmanuel Chailloux, Pascal Manoury et Bruno Pagano :
  Développement d'Applications avec Objective Caml
  O'Reilly, 2000 disponible en ligne

Bibliographie

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

- Xavier Leroy et al :
  The Objective Caml system
  http://caml.inria.fr/pub/docs/manual-ocaml/
- Emmanuel Chailloux, Pascal Manoury et Bruno Pagano :
  Développement d'Applications avec Objective Caml
  O'Reilly, 2000 disponible en ligne

Vient de paraître



John Whitington : *OCaml from the Very Beginning*
S'adresse plutôt à des débutants.
(a été commandé pour la bibliothèque centrale)

Autres ouvrages

Guy Cousineau et Michel Mauny
Approche fonctionnelle de la programmation
Dunod, 1995

Pierre Weis et Xavier Leroy
Le langage Caml
Dunod, 1999

Bibliographie

Catherine Dubois et Valérie Ménissier-Morain
Apprentissage de la programmation avec OCaml. Hermès, 2004

Louis Gacogne
Programmation par l'exemple en Caml
Ellipse, 2004

Philippe Nardel
Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml
Vuibert, 2005

Modes de compilation

Deux façons de travailler avec OCaml

1. compiler (comme en Java, C, .

Deux façons de travailler avec OCaml

1. compiler (comme en Java, C, ...) :
   - permet d'obténir 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*(4+1)-7;;

(* syntax error *)
17 +;;

(* erreur de typage *)
42 + "hello";;

2+3*1;;

5/2;;

-5 mod 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

Les types

- un des traits principaux du langage OCaml
- synthétisés automatiquement par interpréteur/compilateur
  - ☺ combiner commodité des langages non-typés avec la sûreté des langages typés !
- L'ensemble des types OCaml est très riche:
  - types de base : `int`, `float`, `bool`, ...
  - types fonctionnels : `int -> float`, `(int -> int) -> int`, ...
  - types structurés (voir plus tard)
  - objets et classes (pas dans ce cours)
- Grande flexibilité dûe à la polymorphie (voir plus tard)

```
val qSort :  'a list -> 'a list = <fun>
```

- aussi petits inconvénients dues à inférence de type
  - ☹ par ex. pas de conversion automatique entre types

Les types

- un des traits principaux du langage OCaml
- synthétisés automatiquement par interpréteur/compilateur
  - ☺ combiner commodité des langages non-typés avec la sûreté des langages typés !
- L'ensemble des types OCaml est très riche:
  - types de base : `int`, `float`, `bool`, ...
  - types fonctionnels : `int -> float`, `(int -> int) -> int`, ...
  - types structurés (voir plus tard)
  - objets et classes (pas dans ce cours)
- Grande flexibilité dûe à la polymorphie (voir plus tard)

```
val qSort :  'a list -> 'a list = <fun>
```

- aussi petits inconvénients dues à inférence de type
  - ☹ par ex. pas de conversion automatique entre types

Les types

- un des traits principaux du langage OCaml
- synthétisés automatiquement par interpréteur/compilateur
  - ☺ combiner commodité des langages non-typés avec la sûreté desnon-typ

Les types

- un des traits principaux du langage OCaml
- synthétisés automatiquement par interpréteur/compilateur
  - ☺ combiner commodité des langages non-typés avec la sûreté des langages typés !
- L'ensemble des types OCaml est très riche:
  - types de base : `int`, `float`, `bool`, ...
  - types fonctionnels : `int -> float`, `(int -> int) -> int`, ...
  - types structurés (voir plus tard)
  - objets et classes (pas dans ce cours)
- Grande flexibilité dûe à la polymorphie (voir plus tard)

```
val qSort :  'a list -> 'a list = <fun>
```

- aussi petits inconvénients dues à inférence de type
  - ☹ par ex. pas de conversion automatique entre types

Les types

- un des traits principaux du langage OCaml
- synthétisés automatiquement par interpréteur/compilateur
  - ☺ combiner commodité des langages non-typés avec la sûreté des langages typés !
- L'ensemble des types OCaml est très riche:
  - types de base : `int`, `float`, `bool`, ...
  - types fonctionnels : `int -> float`, `(int -> int) -> int`, ...
  - types structurés (voir plus tard)
  - objets et classes (pas dans ce cours)
- Grande flexibilité dûe à la polymorphie (voir plus tard)

```
val qSort :  'a list -> 'a list = <fun>
```

- aussi petits inconvénients dues à inférence de type
  - ☹ par ex. pas de conversion automatique entre types

int

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

opérateurs:

- $+$: addition (infixe)
- $-$: soustraction (infixe)
- $*$: moltiplication (infixe)
- $/$: division entière (infixe)
- mod : reste de la division entière (infixe)
- ...

int (exemples)

```
# 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: arithmetiques

+., −., ∗., /.

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 (exemples)

```
# 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: arithmetiques

+., −., ∗., /.

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 (exemples)

```
# sin (2.0/.3.0);;
-
  #
```

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 conditionel
- seulement une des deux branches e1 et e2 est évaluée.

bool (exemples)

```
# 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 conditionel
- seulement une des deux branches e1 et e2 est évaluée.

bool (exemples)

```
# 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
```

types fonctionnels

Le type d'une fonction n'est plus un type de base.

une façon d'introduire les fonctions est à travers:

fun var1 . . . varn > expr

(on peut utiliser aussi:

function var > expr

dans le cas de fonctions à un seul argument)

pour évaluer une fonction il faut lui donner des augments

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

l'argument d'une fonction peut être une fonction

types fonctionnels (exemples)

```
# 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
```

types fonctionnels

Le type d'une fonction n'est plus un type de base.

- une façon d'introduire les fonctions est à travers:

  fun var1 . . . varn −> expr

  (on peut utiliser aussi:

  function var −> expr

  dans le cas de fonctions à un seul argument)

- pour évaluer une fonction il faut lui donner des augments
- le résultat de l'évaluation d'une fonction peut être une autre fonction (évaluation partielle)
- l'argument d'une fonction peut être une fonction

types fonctionnels (exemples)

```
# 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
```

Déclaration des valeurs

let nom = expr

- associe à nom la valeur de l'expression expr pour la réutiliser après:

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

- En particulier, on peut donner un nom aux fonctions

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

- OCaml fournit aussi une syntaxe simplifiée:

let f x y = expr est équivalent à **let** f = **fun** x y -> expr

```
# let g f x = f (f (x * 2));;
val g : (int -> int) -> int -> int = <fun>
# let sq x = x * x;;
val sq : int -> int = <fun>
# let h = g sq;;
val h : int -> int = <fun>
# h 1;;
- : int = 16
```

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 **in** expr2

- permet une définition 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
```

Utilité **let in**

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

- la définition suivante est très inefficace car elle évalue deux fois 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 majore efficacité quand on appelle plusieurs fois la même expression.

- la définition suivante est très inefficace car elle évalue deux fois 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>
```

Visibilité des liaisons

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

o $x = 1$

o $x = 2$

o $x = 3$

o $x = 2$

o $x = 1$

Seulement la liaison la plus locale est visible.

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'evaluation de `even 4` enchaîne une alternance d'evaluations:

$$\texttt{even}\,4 \mapsto \texttt{odd}\,3 \mapsto \texttt{even}\,2 \mapsto \texttt{odd}\,1 \mapsto \texttt{even}\,0 \mapsto \texttt{true}$$

Doggy bag

- organisation du cours
- introduction programmation fonctionnelle
- introduction OCaml
- modes de compilation
  - compilation: ocamlc ou ocamlopt
  - boucle interprétation: ocaml
- types
  - int , float , bool
  - types fonctionnels : **fun** var1 ... varn –> expr
- déclaration des valeurs
  - **let** nom = expr
  - **let** **rec** nom = **fun** arg1 arg 2... –> expr
  - **let** nom = expr1 **in** expr2
  - **let** nom1 = expr1 **and** ... **and** nomn = exprn