Objectifs didactiques

Découverte d'un nouveau paradigme de programmation : après les langages de script, la programmation structurée et à objets, présenter la programmation fonctionnelle illustrée par le langage OCaml.

Le temps imparti étant très court, on n'abordera que les traits fonctionnels et distinctifs d'OCaml.

L'approche sera très différente, voire déroutante, par rapport à celles que vous avez déjà pratiquées.

À cause de la nouveauté et du peu de temps, il sera donc essentiel de travailler en dehors des heures de classe.

Domaines d'usage de OCaml

Usage général

Domaines de prédilection

- calcul symbolique : preuves mathématiques, compilation, interprétation, analyses de programmes ;
- prototypage rapide et langages dédiés (*Domain Specific Languages*).

Enseignement et recherche

Classes préparatoires, grandes universités (Europe, USA, Japon).

Industrie

CEA, EDF, France Télécom, Simulog.

Gros logiciels

Coq, Ensemble, Unison.

Quelques mots-clés

Le langage OCaml est dit :

- *fonctionnel* : les fonctions peuvent être directement passées à d'autre fonctions et être retournées ;
- *à gestion de mémoire automatique* (comme Java) ;
- *fortement typé* : le système de types garanti l'absence d'erreurs à l'exécution dues à des incohérences sur les données ;
- *avec inférence de types statique* : les annotations de types sont facultatives car inférées par le compilateur ;
- *compilé* ou *interactif* (interactif comme une calculette) ;
- *à objets* et *à modules*.

Mini-ML

Commençons pas présenter un sous-ensemble du noyau fonctionnel de OCaml : mini-ML.

Une *phrase* est définie par les cas suivants, où e dénote une expression, x et f sont des variables et **let**et **rec**sont des mots-clés :

- définition globale $e;;$
  $\textbf{let}\ x = e;;$
- définition globale récursive $\textbf{let rec}\ f = e;;$

Les phrases sont terminées par « ;; » et un *programme* est une suite de phrases.

Méta-variables et variables

Lorsque l'on écrit que *e* est une expression, on veut dire que *e* désigne une portion de phrase OCaml qu'on classe syntaxiquement (c.-à-d. d'après leur forme) dans les expressions.

On dit que *e* est une *méta-variable* car, étant un nom c'est une variable, mais cette variable n'existe pas dans le langage décrit (OCaml) : elle existe dans le langage de *description*. Autrement dit, ce n'est pas du OCaml mais une notation pour décrire des fragments de OCaml (éventuellement une infinité).

Ainsi la méta-variable *x* désigne un ensemble (peut-être infini) de variables OCaml, et il ne faut pas la confondre avec la variable OCaml x. De même, la méta-variable e_1 désigne une infinité d'expressions.

Les expressions

Une expression e est définie *récursivement* par les cas suivants :

• variable	f, g, h (fonctions) ; x, y, z (autres).
• fonction (ou abstraction)	**fun** $x \to e$
• appel (ou application)	$e_1\ e_2$
• opérateur arithmétique	(+) (-) (/) (*)
• opération arithmétique	e_1 + e_2 ou e_1 - e_2 ou e_1 / e_2 ou e_1 * e_2
• unité ou constante entière	() ou 0 ou 1 ou 2 etc.
• parenthèse	(e)
• définition locale	**let** $x = e_1$ **in** e_2

Remarques Ce que nous écrivons joliment $\to$ s'écrit -> en ascii.

Un programme correct

```
let x = 0;;
let id = fun x -> x;;
let y = 2 in id (y);;
let x = (fun x -> fun y -> x + y) 1 2;;
x+1;;
```

Remarques

- Les variables doivent débuter par une minuscule.
- La flèche est associative à *droite* : l'expression
  fun $x_1 \to$ **fun** $x_2 \to \ldots \to$ **fun** $x_n \to e$ est équivalente à
  fun $x_1 \to$ (**fun** $x_2 \to$ ($\ldots \to$ (**fun** $x_n \to e$)) $\ldots$)

Un programme correct (suite)

- L'appel de fonction est associatif à *gauche* : l'expression $e_1\ e_2\ e_3\ \ldots\ e_n$ est équivalente à $((\ldots(e_1\ e_2)\ e_3)\ldots)\ e_n)$
- L'application des fonctions est prioritaire par rapport à celle des opérateurs :

  `f 3 + 4` équivaut à `(f 3) + 4`
- L'application des opérateurs est prioritaire par rapport à l'abstraction :

  `fun x -> x + y` équivaut à `fun x -> (x + y)`

Phrase unique

Un programme, c.-à-d. une suite de définitions globales, peut toujours se réécrire en une seule phrase à l'aide de définitions locales imbriquées.

L'exemple précédent est équivalent à

```
let x = 0 in
  let id = fun x -> x in
  let _ = let y = 2 in id (y) in
  let x = (fun x -> fun y -> x + y) 1 2
in x+1;;
```

Le symbole _ désigne une variable dont le nom est unique et différent de toutes les autres. Sans perte de généralité, nous étudierons donc les programmes réduits à une seule expression. Le résultat du programme est l'*évaluation* de `x+1`.

Un autre programme correct

OCaml est dit fonctionnel car ses programmes sont bâtis sur des fonctions qui modélisent les fonctions mathématiques calculables.

Par exemple :

```
let compose = fun f -> fun g -> fun x -> f (g x) in
  let square = fun f -> compose f f in
  let double = fun x -> x + x in
  let quad = square double
in square quad;;
```

Interprétation mathématique

$$
\begin{aligned}
\texttt{compose} &\equiv f \mapsto (g \mapsto (x \mapsto f(g(x)))) \equiv f \mapsto (g \mapsto (x \mapsto f \circ g(x))) \\
&\equiv f \mapsto (g \mapsto f \circ g) \quad \text{car} \quad \forall h.(x \mapsto h(x) \equiv h) \\
\texttt{square} &\equiv f \mapsto f \circ f \equiv f \mapsto f^2 \\
\texttt{double} &\equiv x \mapsto x + x \equiv x \mapsto 2x \\
\texttt{quad} &\equiv (f \mapsto f^2)(x \mapsto 2x) \equiv (x \mapsto 2x)^2 \equiv x \mapsto 4x \\
\texttt{square}\,(\texttt{quad}) &\equiv (f \mapsto f^2)(x \mapsto 4x) \equiv (x \mapsto 4x)^2 \equiv x \mapsto 16x
\end{aligned}
$$

Donc le résultat du programme (`square quad`) est une fonction.