Introduction à CAML

2 février 2015

1 Principes généraux

CAML est un langage créé par l'INRIA (dernière version : CAML-light 0.75), qui est une simplification d'un langage plus complet et évolué nommé OCAML. Diffusé de manière libre et gratuite, il est disponible pour de multiples plates-forme (UNIX, Linux, Mac, Windows,...) à l'adresse suivante :

`http://caml.inria.fr`

CAML est un langage compilé (comme le langage C) : l'utilisateur écrit un code appelé code source, qui est transformé par le compilateur en code exécutable par la machine, puis qui peut être utilisé. A la différence des langages interprétés (comme celui de Python), chaque modification du code source oblige à une recompilation complète.

Pour éviter ce genre de manipulations fastidieuses dans le cadre d'une mise au point ou d'un apprentissage, CAML est fourni avec une interface interactive (écrite en code CAML puis compilée) qui simule un langage interprété.

Malheureusement, cette interface est rudimentaire et en mode console. C'est pourquoi plusieurs logiciels intermédiaires viennent s'intercaler pour offrir des interfaces graphiques plus ou moins complexes. Celle du lycée (nommée "camleditor") est écrite elle-même en OCAML, elle bénéficie du label "Fait maison" et est disponible sous forme de paquet debian ou rpm pour machines Linux récentes. Un de vos prédécesseurs (mr Huguenin) a écrit une version Windows en Java et la propose librement.

2 Premiers pas

Pour lancer la phase d'évaluation dans l'interface interactive, on envoie au compilateur du code syntaxiquement correct, formés de *phrases*. Un phrase se termine toujours par un double point-virgule `;;`.

Testez les commandes suivantes et remarquez la façon de répondre de CAML (envoyez la demande d'évaluation au compilateur à chaque double point-virgule) :

```
5 + 7;;
2.0 +. 6.2;;
-3.14 / 2;;
```

2.1 CAML est un langage fortement typé

Tout objet CAML a un type (par exemple, entier, flottant, booléen, etc) et à chaque type sont associés les opérations et instructions permises. Et réciproquement, à chaque instruction est associée la liste des types utilisables (voir plus loin).

La première et principale cause d'erreurs dans la programmation CAML est l'erreur de typage.

Par exemple, les opérateurs `+ - * /` sont les opérations usuelles de l'arithmétique des entiers et de ceux-là seulement. Toute utilisation de ses symboles avec autre chose que des entiers entraîne un message d'erreur.

2.2 Définition des symboles de constantes

Tout ce que fait CAML est par essence un calcul et que comme tout calcul, cela produit un résultat typé.

Pour calculer, CAML utilise les objets dits natifs (les nombres explicites, les caractères explicites et tout autre symbole constant explicite) ainsi que des *constantes symboliques*. Avant tout calcul faisant intervenir une constante symbolique, il faut la déclarer (comme le font les mathématiciens dans des constructions de la forme « Soit f la fonction $x \mapsto xe^x$ »). Pour cela, on utilise la construction : **let** *symbole* = *valeur*. Cette déclaration s'appelle une liaison.

(Envoyez la demande d'évaluation à chaque double point-virgule)

```
let x = 2;;
let y = x + 4;;
let a = 6 and b = 2;;
```

On peut comme dans la dernière ligne précédente définir plusieurs symboles simultanément, à la condition que ces symboles soient indépendants. La définition suivante est interdite :

```
let f = 2 and g = f + 3;;
```

Toutes les liaisons précédentes sont globales : les constantes ainsi créées sont reconnues pendant toute la session.

Pour créer des symboles locaux, on utilise la construction :
let *symbole* = *valeur* **in** *calcul*

Le symbole ainsi défini n'est connu que dans le calcul qui suit.

(Ce qui est encadré entre (∗ et ∗) est un commentaire : il est inutile de le recopier)

```
let a = 2. in
  a *. 3.5;;
let d =
  (* d est le resultat de ce calcul : le dernier de la suite d'instructions *)
  let c = 5 in
  6 * c;;
d;;
c;;
```

Le symbole c est localement défini dans le bloc qui suit : il est inaccessible dans l'environnement global.

Dans le cas d'un même symbole représentant des constantes différentes (une locale et une globale par exemple), lors d'un calcul, le compilateur remplace les symboles de constante par la dernière définition en cours au moment du calcul.

```
let c = 10;;
let d =
  let c = 5 in
  c * 6;;
d;;
c;;
```

2.3 CAML est un langage fonctionnel

Le cadre naturel de calcul de CAML est celui des fonctions.

Pour définir une fonction, on utilise la construction : **let** *symbole var var ...* = *valeur*, comme dans les exemples suivants.

```
let f x = 2 * x;;
let g x y = x +. y;;
```

f est une fonction d'une seule variable, g de deux variables.

CAML est muni d'une propriété dite d'inférence de type, c'est-à-dire qu'il analyse le code source pour en déduire le type des variables si c'est possible ou tout au moins les relations entre les types des variables. Dans les exemples précédents, CAML note l'utilisation du symbole ∗ réservé aux entiers, donc il déduit que le symbole x est forcément un entier. De même, le symbole +. est utilisé dans la définition de g, or il est associé au type

flottant, donc CAML déduit que x et y sont des flottants. Remarquez comment CAML affiche le résultat de l'inférence de types.

Quand CAML ne peut pas décider du type précis des variables lors de la compilation, il les considère d'un pseudo-type noté 'a, 'b,... (lire α, β,...). On dit que le symbole est *polymorphe*. Cependant, il analyse les rapports entre les pseudo-types rendus nécessaires par la cohérence du code source.

```
let h x y =
  x = y;;
```

Pour utiliser les fonctions, on les appelle selon la même syntaxe :

```
f 5;;
g 9. 6.;;
g 8. (-10.);;
g 8. -10.;;
```

Remarque. On peut définir les fonctions avec les parenthèses, mais dans ce cas, il faut indiquer les parenthèses lors de l'appel.

```
let h (x,y) = x - y;;
h(2,5);;
h 2 5;;
```

Il y a en fait une différence algébrique entre ses deux sortes de définitions, mais à isomorphisme prés, cela donne le même résultat.

On peut définir des fonctions d'ordres supérieurs :

```
let somme f g =
  let h x = f x + g x
  in
  h;;

let f x = 2 * x and g x = x + 5;;
let h = somme f g;;
h 3;;

let compose f g =
  let h x = f (g x)
  in
  h;;
let k = compose f g in k 3;;
```

On peut aussi définir des fonctions partielles :

```
let prod x y = x * y;;

let p = prod 40;;

somme p f 5;;
```

2.4 Recommandations syntaxiques

Des commentaires peuvent être ajoutés au sein du code source en les encadrant par les symboles (* et *).

CAML est particulièrement sensible à la précédence des opérateurs et instructions. Il ne faut pas hésiter à encadrer entre parenthèses les sous-formules ou à les encadrer par les mots-clefs **begin** et **end**.

Enfin, les concepteurs de CAML proposent les conventions suivantes :

— on insère un espace avant et après les symboles d'opérations binaires (+ * - / = < etc)

— on insère un espace aprés les séparateurs (la virgule , séparateur dans les n-uplets, le point-virgule ; séparateur d'instructions dans les séquences d'instructions, le double pont-virgule ;; fin de phrase, etc), mais pas avant
— on indente les blocs d'instructions

3 Principaux types et opérateurs associés

3.1 Types constants

a) **Entiers** (`int`).
Ils sont codés sur 31 bits donc compris entre -2^{30} et $2^{30}-1$ (le premier bit sert à indiquer le signe). CAML ne signale pas les débordements de mémoire (perte de chiffres au-delà des bornes indiquées). Autrement dit, CAML fait du calcul modulo 2^{31}. Pour utiliser des entiers plus grands, il faut utiliser un module spécialisé de CAML.
Les opérateurs associés sont $+ - *$, les symboles de division euclidienne / et mod qui calculent respectivement le quotient et le reste. A noter qu'il n'y a pas de symbole d'exponentiation sur les entiers. La fonction print_int sert à afficher un entier.

b) **Flottants** (`float`).
Ce sont les nombres à virgule, dont l'étendue est beaucoup plus large que celle des entiers, mais avec les imprécisions dues aux erreurs d'arrondi.
Les opérateurs associés sont +. −. *. /.
Le symbole d'exponentiation sur les flottants est noté **. On dispose des fonctions mathématiques usuelles sqrt, sin, cos, tan, acos, asin, atan, exp, log, sinh, cosh, tanh. La fonction print_float sert à afficher un flottant.
Si nécessaire, il existe des fonctions pour transformer un entier en flottant ou vice-versa.

c) **Booléens** (`bool`).
Les booléens ne peuvent prendre que deux valeurs : true, false.
On dispose des opérateurs logiques usuels : et (noté &&), ou (noté ||), non (noté **not**).
Les comparaisons (grâce aux opérateurs polymorphes = < > <> >= <=) sont des calculs de type booléen.

d) **Produits cartésiens de type**
On peut définir des couples, des n-uplets d'objets en utilisant la notation des mathématiques (avec parenthèses et virgule).

```
let couple = (2, 3.);;
let (x, y) = couple;;
x;;
y;;
```

Remarquez la façon de récupérer les éléments du couple (par correspondance de motif).

e) **Caractères** (`char`).
Un caractère est tout symbole d'écriture encadré par deux apostrophes inversées (ou accents graves, obtenus par la combinaison des touches `Alt Gr + 7`), comme `'a'`, `'2'`, etc.

f) `unit`
Comme tout est calcul, même les affichages à l'écran, les modifications de références, les ouvertures de fichier, etc ont un type : le `unit`, qui n'a qu'une seule valeur représentée par le symbole ().

4 Conditionnelle

CAML dispose d'une « instruction » conditionnelle classique :

if *booléen* **then** *résultat1* **else** *résultat2*

qui est en fait un calcul, donc *résultat1* et *résultat2* doivent être de *même type*.

Tout ça fait de la conditionnelle de CAML quelque chose de bien plus puissant qu'un simple branchement logique.

```
let abs x =
  if x > 0 then x
  else -x;;

let range a b =
  let (c,d) =
    if a > b then (a,b)
    else (b,a)
  in
  (c,d);;

range 3 5;;

let h x =
  if x > 0 then print_string "positif"
  else print_string "negatif";;

h 1;;
h (-8);;

let g x =
  if x > 0 then print_string "positif";;

g 1;;
g (-8);;

let k x =
  if x > 0 then 1;;
```

Finalement, on peut voir une instruction conditionnelle comme une fonction de `bool -> 'a`, ce qui permet d'écrire des phrases comme

```
let y =
  let x = exp 4. in
  2 +. (if x > 60. then x
              else x *. x);;
```

Exercices

- Écrivez une fonction `s` à trois paramètres a, b, c qui résout l'équation du second degré $ax^2 + bx + c = 0$.
- Écrivez une fonction somme qui a pour paramètre un triplet de nombres et en calcule la somme.
- On représente un nombre rationnel $\frac{p}{q}$ par le couple (p, q) : écrivez des fonctions somrat, prodrat qui calculent la somme, le produit de deux rationnels.
- Écrivez une fonction compose qui a deux paramètres fonctions et qui calcule la fonction composée.
- Écrivez une fonction cube qui calcule le cube d'un entier, puis une fonction qui calcule la puissance 81 d'un entier.

5 Programmation impérative

CAML est un langage fonctionnel (basé sur la théorie mathématique du λ-calcul) mais il peut mimer les principaux aspects de la programmation impérative (c'est-à-dire celle qui s'occupe de l'état de la machine).

On appelle séquence toute succession de calculs séparés par des ; Les calculs se font dans l'ordre de leur écriture, les uns après les autres : le symbole ; correspond donc au mot français « puis ».

Dans une séquence, les calculs sont tous de type `unit` (sinon ils sont ignorés, et à quoi pourrait bien servir de faire des calculs dont les résultats sont perdus ?), sauf éventuellement le dernier. La valeur de la séquence est la valeur du dernier calcul.

```
let t = make_vect 3 1 in
  t.(1) <- 5;
  t.(2) <- t.(2) - t.(1);
  t;;
```

Pour éviter les ambiguïtés, on peut clairement délimiter les séquences par des parenthèses ou les mots-clefs **begin** ... **end**.

5.1 Types mutables

a) **Chaînes de caractères** (`string`).

Une chaîne de caractères est une suite de caractères, on la note encadrée par deux guillemets comme dans "mot", "polygone".

L'opérateur ^ sert à concaténer les chaînes. La longueur d'une chaîne est donnée par la fonction string_length *ch*. Chaque caractère de la chaîne est accessible par son indice (attention, les indices commencent à 0) et est modifiable.

```
let ch =
  let ch1 = "lap" and ch2 = "ins" in
  ch1 ^ ch2;;
string_length ch;;
ch.[0];;
ch.[5];;
ch.[1] <- 'u';;
ch;;
```

b) **Tableaux (ou vecteurs)** (`vect`).

Un tableau est une suite d'objets de même type, rangées de manière consécutive dans la mémoire de l'ordinateur. Un tableau d'entiers est de `int vect` etc.

On définit un tableau par la notation $[|1.;\ 2.;\ -8.|]$ (ici un tableau de flottants de `float vect`).

La longueur du tableau est donnée par la fonction vect_length. Chaque élément du tableau a un indice (comme pour les chaînes, les indices vont de 0 à vect_length $t - 1$) et est modifiable (sous réserve de respecter la cohérence des types).

make_vect n *obj* construit un tableau de n objets identiques à *obj*.

```
let t = make_vect 5 0.;;
vect_length t;;
t.(1);;
t.(4) <- 1.23;;
t;;
```

c) **Références** (`ref`).

Les symboles introduits par le mot-clef **let** sont des constantes. Or il arrive qu'on ait besoin de variables, comme dans la plupart des autres langages de programmation. On définit alors une constante qui désigne non pas la valeur proprement dite, mais l'adresse en mémoire où cette valeur a été stockée. On définit une référence en faisant suivre le symbole = par le mot-clef ref.

```
let x = ref 0;;
x;;
!x;;
x := 1;;
x := !x + 10;;
```

Si x désigne une référence à une valeur d'un certain type, alors !x représente la valeur référencée. On peut modifier la valeur référencée grâce au symbole d'affectation :=, mais comme toujours, il faut respecter le type déclaré lors de la définition.

Remarque. En fait, les types tableau et chaîne sont aussi des références, ce qui explique leur caractère mutable. Les tableaux, chaînes passés en paramètre à une fonction le sont par référence : toute modification à l'intérieur de la fonction affecte l'objet lui-même.

5.2 Instruction conditionnelle de branchement

La conditionnelle de CAML peut jouer le rôle d'une instruction conditionnelle classique de branchement, comme dans la plupart des langages.

On notera qu'il n'y a pas de délimiteur explicite de fin d'instruction, donc en cas de doute, il ne faut pas hésiter à clairement délimiter à l'aide de parenthèses ou de **begin** ... **end**. Si on a donc plusieurs actions à enchaîner, il faut donc les regrouper comme indiqué précédemment. Dans l'exemple qui suit, devinez le résultat des calculs et testez pour voir si vous avez raison.

```
let f t =
  if (t.(0) mod 2) = 0 then
        t.(0) <- t.(0) / 2
  else
        t.(0) <- 3 * t.(0) + 1;
        t.(1) <- 2 * t.(1);
  t;;

f [|6; 8|];; f [|3; 5|];;

let g t =
  if (t.(0) mod 2) = 0 then
    t.(0) <- t.(0) / 2
  else begin
    t.(0) <- 3 * t.(0) + 1;
    t.(1) <- 2 * t.(1)
  end;
  t;;

g [|6 ; 8|];; g [|3; 5|];;

let h t =
  if (t.(0) mod 2) = 0 then
    t.(0) <- t.(0) / 2;
    t.(1) <- 2 * t.(1);
  else
    t.(0) <- 3 * t.(0) + 1;
    t.(1) <- 2 * t.(1);
  t;;

h [|6; 8|];; h [|3; 5|];;
```

La partie **else** *résultat2* est obligatoire, sauf dans un seul cas : si le type de *résultat1* est unit, on peut omettre la partie finale du test.

if *booléen* **then** *résultat1* ; est correctement typé si et seulement si type de *résultat1* est unit.

5.3 Boucles indexées

CAML fournit une boucle indexée classique : **for** *var* = *val-début* **to** *val-fin* **do** *instructions* **done**

On notera que *val-début*, *val-fin* doivent être de type entier et que le pas est toujours de 1. De plus, le compteur de boucle n'est pas une référence, mais une constante entière, donc ne peut être modifiée dans le corps de la boucle.

On dispose aussi d'une version décroissante en remplaçant **to** par **downto**.

Il n'y a pas d'instructions comme break ou return pour interrompre prématurément une boucle indexée (comme dans d'autres langages de programmation plus « sales »...). Une boucle indexée va donc toujours passer par toutes les valeurs prévues à l'avance. Si on doit interrompre une boucle indexée, c'est que le programme a été mal construit. Il faut alors utiliser une boucle conditionnelle.

5.4 Boucles conditionnelles

De même, on dispose d'une boucle conditionnelle : **while** *booléen* **do** *instructions* **done**.

5.5 Exceptions

Un calcul ne peut pas toujours être mené à bien même s'il est correctement typé (exemple : on ne peut pas diviser par 0). CAML dispose d'un type particulier qui s'appelle les exceptions. Quand une erreur survient, CAML lève une exception (expression consacrée) qui peut être interceptée par le programme en cours si cette interception (et donc la possibilité d'apparition de l'erreur) a été prévue. Dans ce cas, le programme réagit à la survenue de l'erreur selon ce que le programmeur lui a demandé de faire. Sinon, CAML interrompt le calcul en cours : le résultat du calcul est alors une exception non rattrapée.

Vous-même pouvez empêcher le déroulement anormal d'un calcul par la levée d'une exception non rattrapée. Pour interrompre prématurément un calcul, on utilise l'instruction : failwith *chaîne-avertissement* (elle interrompt le calcul, lève une exception et affiche éventuellement un message d'avertissement)

```
let f x y =
  if x = 0 then failwith "division␣par␣zero"
  else y / x
;;
```

Il ne faut pas croire que cette fonction failwith permet d'interrompre une boucle et de rendre un résultat ! Elle peut effectivement interrompre une boucle, mais le résultat sera une exception et rien d'autre, donc inutilisable pour récupérer une valeur. Il faudrait intercepter l'exception pour éviter l'arrêt de la fonction et retourner une valeur, mais laquelle ?