TP1 Caml

Marin Bougeret

13 septembre 2010

1 Posons le décors

Ocaml est un langage fonctionnel (Vs programmation dite "impérative", par exemple le C). Pour caricaturer, un algorithme en Ocaml est la composition de l'application de plusieurs fonctions, alors qu'en impératif on exécute une séquence d'instructions qui peuvent chacune changer l'état de la mémoire. Remarquons que le Ocaml permet tout de même d'écrire de l'impératif (c'est un langage fonctionnel impur), via par exemple les tableaux (ou appelés vecteur) ou les boucles (for et while), dont on s'interdit l'utilisation aujourd'hui !

Voici maintenant quelques éléments pour survivre.

2 Getting started : lancer Ocaml

Je suppose que vous pouvez vous identifier sur votre machine, et ouvrir un terminal (si il y a un problème n'hésitez pas à demander aux encadrants présents !).

Un programme est une succession de "phrases caml", séparée par ";;". Il y a deux façons d'écrire des programmes. La première (je vous la recommande pour aujourd'hui) consiste à lancer l'interpréteur en lançant la commande "ocaml" (on dira alors que l'on est dans le mode "intéractif"), puis à écrire les lignes de caml directement dans l'interpréteur. Dans ce mode, chaque phrase caml est analysée immédiatement par l'interpréteur. Cela est pratique pour faire des "petits tests" uniquement, car une fois que l'on quitte ocaml (en tappant #quit ;;), tout est perdu ! Une solution pratique est bien sûr d'ouvrir en parallèle un éditeur de texte (par exemple vi/emacs/gedit) pour pouvoir sauvegarder vos oeuvres, puis

- (brutal) soit de copier coller votre code entre l'éditeur et l'interprète ocaml
- soit d'utiliser la commande "#use "algo1.ml"", où le fichier "algo1.ml" contient bien sûr votre tp !

La deuxième solution consiste à écrire dans un fichier quelconque (fichier que l'on nommera cependant avec une extension ".ml", comme par exemple "algo1.ml"), puis à lancer la commande "ocaml nomfichier" (par exemple "ocaml algo1.ml"). Attention en exécutant vos algorithmes ainsi, vous ne verrez QUE les

affichages réalisés les commandes "print_" (voir aide mémoire, mais on négligera ces fonctions aujourd'hui).

3 Les indispensables (pour aujourd'hui)

Je suppose que vous savez tous ce qu'est un algorithme, mais que tout le monde n'a pas fait de ocaml. Il est possible que les quelques lignes ci-dessous soient triviales pour un sous ensemble du groupe, et nouvelles pour un autre. Ce document est écrit plutôt dans l'esprit d'un rappel que d'un cours auto-suffisant. N'hésitez donc à pas à communiquer entre vous si certaines lignes ne sont pas claires. Notez que vous avez à disposition un aide mémoire (de Michel Quercia) sur la syntaxe de ocaml. Vous pouvez taper tous les exemples (encadrés) données ci-dessous dans l'interpréteur pour voir le résultat. Ceux qui veulent peuvent aller directement en Section 4 ou Section 5.

3.1 Types simples

Voici des types de base disponibles en ocaml :
int (les entiers), float (les réels), bool (les booléens). Vous pouvez vérifier que ocaml connaît bien ces types en tappant les phrases caml suivantes dans l'interpréteur :

```
3;;
5.2;;
true;;
```

Voici des exemples d'opérations disponibles sur ces types de base :

```
3+1;;
5.1 +. 2.0;;
not(true || false) && (4=6);;
```

"||" dénote le "ou" et && dénote le "et". Notez que l'opérateur "=" est prédéfini pour TOUS les types (même ceux que vous définirez vous même, voir Section 3.5)!!!

Enfin, le produit cartésien est prédéfini en Ocaml :

```
(3,true);;
(5,4.3,2);;
```

3.2 Types plus complexes

Les listes sont des objets très importants, car ils permettent de manipuler de grand nombres d'éléments. "[]" dénote la liste vide. L'opérateur " ::" permet d'ajouter un élément en tête de liste, et l'opérateur "@" permet de concatener deux

listes. Remarquons que pour construire rapidement une liste on peut directement tapper "[element1 ;element2 ;... ;elementk] ; ;". Si "l" est une liste, "List.hd (l)" permet d'obtenir l'élément en tête de liste, et "List.tl (l)" permet d'obtenir toute la queue de la liste. Voici un exemple sur la manipulation de listes :

```
1 :: 3 :: 5 :: 3 :: [] ; ;
[3; 2] ; ;
1 :: [3; 2] ; ;
[1; 4]@[1; 5] ; ;
List.hd ([3 ;2]) ; ;
List.tl ([4 ;3 ;4 ;3 ;55]) ; ;
```

Les fonctions sont des objets comme les autres (comme un entier). On construit une fonction comme suit :

```
function x -> x+1 ; ;
function x -> not x ; ;
function (x,y) -> x +. y ; ; function x -> function y -> x +. y ; ;
```

Remarque : Ocaml est très fort, il fait de l'inférence des types ! Autrement il déduit les types des paramètres et du résultat (par exemple quand on écrit "+." ocaml comprend que x et y sont des floats).

3.3 Déclarations

Jusqu'ici tout ce que nous avons tapé est "oublié" immédiatement à chaque fin de phrase (" ; ;"). Pour remédier à cela on peut donc déclarer des variables avec le mot clef "let" :

```
let a = 3 ; ;
a ; ;
b ; ;
let ma_fonction = function x -> x + 1 ; ;
ma_fonction ; ;
ma_fonction (4) ; ;
ma_fonction a ; ;
let liste = [3 ;2 ;5] ; ;
let liste2 = 1 : :liste ; ;
let mon_bool = (List.tl(liste2)=liste) ; ;
```

Commentaires dans l'ordre :
La variable "x" est maintenant connue est associée à la valeur 3, contrairement à "y" qui est inconnu. Remarquez que l'objet en partie droite du "=" peut être de type "complexe". Les parenthèses lors de l'appel de fonction sont facultatives. Enfin, n'utilisez pas de majuscules pour nommez vos variables ! !

Notez enfin que pour la déclaration de fonction, il est possible d'utiliser la forme "plus pratique" suivante :

```
let ma_fonction x = x + 1 ; ;
```

Les déclarations de fonctions récursives se font avec le mot clef "rec" :

```
let rec factorielle x =
if x = 0 then 1 else
x * factorielle (x-1) ; ;
```

Enfin, les déclarations locales (qui permettent de définir une variable (pouvant être de type simple ou complexe) qui ne sera visible que dans la phrase caml courante), se vont avec le mot clef "in" :

```
let f x =
let pi = 3.14 in
let f_aux l =
match l with
[] -> true
|_ -> false in

if List.hd (x) > pi then true
else f_aux x ; ;
```

3.4 "if" et "match"

Le "if" et le "match" permettent d'écrire des fonctions plus intéressantes ! (95% des fonctions demandées en TP s'écrivent avec un match !) La syntaxe de ces deux instructions est donnée dans l'aide mémoire. Voici quelques exemples :

```
let est_pos a = if a > 0 then true else false ; ;
let f x = match x with
3 -> 1
|4 -> 2
|_ -> 0 ; ;
```

Notez que les types de retour doivent être les mêmes dans tous les cas (on ne peut pas remplacer "2" par "true") et que le "_" (appelé "motif universel") est utilisé lorsque aucun motif en partie gauche n'a pu être reconnu. Une force du Ocaml est que l'on peut utiliser des variables dans les "parties gauches" des règles :

```
let f c = match c with
(0,y) -> y+1
|(y,0)->2*y
|(a,b) -> a+b ; ;
```

Un autre exemple classique où l'on filtre une liste :

```
let rec nb_elem c = match c with
[] -> 0
|a : :reste-> 1 + nb_elem reste;;
```

3.5 Déclaration de types

Une des phrases possibles en caml est la déclaration d'un type. Cela permet de manipuler d'autres objets que les int, float, bool, liste etc .. Le cas le plus simple est le "renommage", comme par exemple :

```
type liste_de_liste_dentiers = int list list;;
type couple_bizarre = int * bool;;
```

Plus intéressant, la définition de nouveaux types. Deux familles de types existent en caml : les types produits et les types sommes. Nous ne parlerons ici que des types sommes. Nous nous contenterons d'exemples :

```
type couleur = Rouge | Noir;;
let x = Rouge;;
let y = Noir;;
x=y;;
```

"Rouge" et "Noir" sont appelés des constructeurs (sans paramètres), et la première lettre des constructeurs DOIT être une majuscule. On peut également utiliser des constructeurs avec paramètres :

```
type de_colore =
De of couleur * int;;
let de1 = De(Rouge,4);;
```

Enfin (le plus important), on peut définir des types récursifs :

```
type liste_int_ou_bool =
Vide
|Booleen of bool * liste_int_ou_bool
|Entier of int * liste_int_ou_bool;;

Entier(4,Booleen(true,Entier(5,Vide)));;
```

4 Exercices pour ceux qui ont jamais fait de caml

Quelques exercices uniquement basés sur les listes.

4.1 Somme

Ecrire une fonction **somme l** qui prend en argument une liste d'entiers et qui retourne la somme des éléments de cette liste

4.2 Concat

Ecrire une fonction **concat l1 l2** qui concatène les listes l1 et l2 (sans utiliser l'opérateur @ bien sûr !).

4.3 Dernier

Ecrire une fonction **dernier l** qui prend en argument une liste **non vide** et qui retourne le dernier élément de celle ci.

4.4 Max

Ecrire une fonction **recherche_max l** qui prend en argument une liste d'entiers et qui retourne l'élément maximal de cet liste (on pourra tout d'abord ecrire une fonction **rech_aux l best** qui, pour $l = [a1, ..., ak]$, renvoi le maximum entre *best* et $max_{1 \leq i \leq k} a_i$).

5 Exercices pour tous : les ABR

Pas de grands projets aujourd'hui, juste une mise en jambe. On considère le type arbre (binaire) suivant :

```
type arbre =
Vide
| Noeud of arbre*int*arbre;;
```

Un exemple arbre :

```
let arbre1 = Noeud(Noeud(Vide,4,Vide), 5, Vide);;
let arbre2 =
Noeud(Noeud(Vide,4,Noeud(Noeud(Vide,3,Vide),6,Vide)), 8, Noeud(Vide,10,Vide));;
```

De plus, on s'intéresse uniquement à ce que l'on appelle les arbres binaires de recherche (notés ABR). Un ABR est un arbre tel que la valeur de tout noeud x est supérieur ou égale à tous les noeuds du sous arbre gauche de x, et strictement inférieure à tous les noeuds du sous arbre droit de x. Par exemple, arbre1 ci-dessus est un ABR, et arbre2 n'est pas un ABR puisque 3 est dans le sous arbre droit de 4.
Voici par exemple un affichage en profondeur d'un ABR (j'utilise "print_int (x)" qui affiche un entier x, ainsi que le séparateur ";" pour séquencer 2 instructions) :

```
let rec affic_prof arb =
match arb with
Vide -> ()
| Noeud(a,x,b) ->
affic_prof a;
print_int (x);
affic_prof b;;
```

5.1 Recherche dans un ABR

L'intérêt d'un ABR est évidemment la recherche, qui s'effectue en $O(h_A)$, où h_A est la hauteur de l'ABR A. Ecrire une fonction **rech arb x** qui renvoi *true* ssi x est présent dans l'ABR arb.

5.2 Insertion dans un ABR

Ecrire une fonction **insert arb x** qui renvoi l'arbre obtenu en ajoutant l'entier x à l'ARB arb.

5.3 Suppression dans un ABR

Ecrire une fonction **suppr arb x** qui supprime l'élément x de l'ABR arb (ou ne fait rien si x n'est pas présent).

5.4 Verification

Ecrire une fonction **verif arb** qui renvoi *true* ssi arb est un ABR.

6 Exercices pour tous : un peu de graphe

Un graphe est un ensemble V de sommets (identifiés par des entiers distincts), et un ensemble $G \subset V \times V$ d'arêtes ($(i,j) \in G$ signifie que les sommets i et j sont reliés). On choisit de représenter un graphe par "liste de successeurs" : pour chaque sommet i, on associe la liste de tous les successeurs j tels que $(i,j) \in G$. Un graphe est donc une liste de liste d'entiers. On considère le type graphe suivant :

```
type graphe = int list list;;
```

Voici un exemple de graphe (voir figure 1) :

```
let g1 = [[1;2;3;6];[2;1];[3;1];[6;1;5;4];[4;5;6];[5;4;6]];;
```

Le sommet 1 (par exemple) a donc comme voisins les sommets 2, 3 et 6. Le sommet 6 (par exemple) a donc comme voisins les sommets 1, 5 et 4.

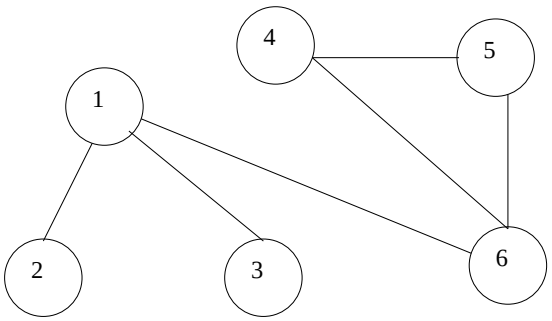

FIG. 1 – Un dessin pour le graphe g1

6.1 Voisins

Ecrire une fonction **voisins gr x y** qui renvoi *true* ssi x et y sont voisins dans gr. Par exemple, voisins g1 3 6 doit renvoyer false, et voisins g1 4 5 doit renvoyer true.

6.2 Atteignables

Ecrire une fonction **atteignable gr x y** qui renvoi *true* ssi y est atteignable depuis x dans gr. On suppose qu'il n'y a pas de boucles dans gr.

6.3 Ajout

Ecrire une fonction **ajout gr x liste** qui renvoi le graphe obtenu en ajoutant le sommet x au graphe gr. "liste" contient l'ensemble des voisins que x doit avoir dans le nouveau graphe (on suppose donc que tous les sommets de liste sont initialement présents dans gr).

7 Pour s'amuser

Ecrire une fonction **boulanger x l** qui étant donné une somme d'argent x (la monnaie à rendre, un entier positif), et une liste de pièces disponibles l (chaque pièce de l est disponible en quantité infinie) renvoi la liste de toute les façons possibles de rendre la monnaie. On renvoi donc une liste de liste. Par exemple, boulanger 6 [3 ;2 ;1] doit renvoyer [[3 ; 3] ; [1 ; 2 ; 3] ; [1 ; 1 ; 1 ; 3] ; [2 ; 2 ; 2] ; [1 ; 1 ; 2 ; 2] ; [1 ; 1 ; 1 ; 1 ; 2] ; [1 ; 1 ; 1 ; 1 ; 1 ; 1]]