TP3 Caml : D'autres outils...

Sujet à finir pour le dimanche 07/12/2014 à 20h00.
À rendre par mail à aurelie.lagoutte@ens-lyon.fr

1 Monde impératif en Caml

1.1 Tableaux et références

L'exemple suivant illustre l'utilisation de tableaux en oCaml (on passe ici dans le monde impératif, par opposition au monde récursif).

```
let digits = Array.create 10 0;;
for i = 0 to 9 do digits.(i) <- digits.(i) + i done;;
let chriffre_trois = digits.(3);;
for i = 0 to 9 do print_int(digits.(i)) done; print_newline();;
```

En Caml, la valeur d'une variable n'est pas modifiable. On peut utiliser des références pour émuler les pointeurs du langage C.

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

Exercice 1 (Swap) *Écrivez une fonction* `swap : 'a ref -> 'a ref -> unit` *telle que* `swap x y` *échange les contenus des références x et y (en faire une version avec variable auxiliaire et une sans, lorsque x et y contiennent des entiers).*

Exercice 2 (Nombre mystère) *Écrivez une fonction* `mystere` *de type* `unit->unit` *implémentant le jeu du nombre mystère : un nombre est choisi aléatoirement*[1] *entre* 0 *et* 100 *et l'utilisateur dispose de* 7 *essais pour deviner ce nombre. On indiquera à chacun de ses essais si le nombre proposé est plus grand, plus petit ou égal au nombre mystère.*

Exercice 3 (Chance) *Écrivez une fonction* `chance` *de type* `unit->unit` *qui remplit un tableau de taille 10 avec des entiers aléatoires de 0 à 10. L'utilisateur gagne (i.e. reçoit un message de victoire) lorsqu'il existe un ou plusieurs indices* `i` *tels que t.(i)=i. Sinon, il reçoit un message de défaite. La fonction doit également afficher le tableau.*

1.2 Enregistrement

Caml possède une structure nommée *enregistrement*. Testez le code suivant :

```
type client={nom:string; adresse: string; mutable solde:int; mutable anciennete:int};;

let tom={nom="Tom"; adresse= "Ici"; solde=0; anciennete=1};;
```

1. On utilisera la fonction `Random.int n` que l'on initialisera d'abord avec `Random.self_init()`.

```
tom.nom;;

tom.solde<-4;;
tom.solde<-tom.solde+1;;
tom.adresse<-"Paris";;
```

Qu'en déduisez-vous sur le mot clé `mutable` ?

Exercice 4 – *Construisez un type* `etudiant` *contenant un nom, un prénom, et une liste de notes.*
- *Écrivez une fonction* `ajoute: etudiant -> int-> unit` *qui ajoute une note à un étudiant.*
- *Écrivez une fonction* `moyenne: etudiant -> float` *qui calcule la moyenne d'un étudiant.*
- *Écrivez une fonction* `triche: etudiant -> unit` *qui enlève la plus mauvaise note d'un étudiant.*
- *Écrivez une fonction* `moyenne_classe: etudiant list -> float` *qui calcule la moyenne de classe (prendre la première note de chaque liste).*

2 Modules et foncteurs

2.1 Module

La programmation modulaire permet la décomposition d'un programme en *unités logiques* plus petites, ainsi que la *réutilisation* plus aisée d'unités logiques indépendantes.
En OCaml, la déclaration d'un module suit la syntaxe suivante [2] :

```
module Complex = struct
  type t = float * float
  let zero = (0.,0.)
  let cons r i = (r,i)
  let oppose (r,i) = (-.r,-.i)
  let plus (r_1,i_1) (r_2,i_2) = (r_1+.r_2,i_1+.i_2)
  let modu (r,i) = sqrt (r*.r +. i*.i)
end
```

Ici le module `Complex` définit le type `t`, la valeur `zero`, ainsi que les fonctions `cons`, `oppose`, `plus` et `modu`.
On a deux manières de se référer aux types/fonctions déclarées dans le module :
- de manière explicite, par `Complex.zero` ;
- après l'appel à `open Complex`, de manière implicite par `zero`.

On peut de plus « cacher » la définition de `t` en forçant le *type module* de `Complex` :
À un module on associe une ou plusieurs signatures qui vont rendre visible un sous-ensemble ou l'ensemble des fonctions/types déclarés dans le module (on parle d'encapsulation). Un module est donc indiscociable d'une signature.

```
module type TComplex1 = sig
  type t = float * float
  val zero : t
```

2. Notez le point qui suit chaque opérateur arithmétique. Ces opérateurs prennent des flotants en opérande.

```
  val cons : float -> float -> t
  val oppose : t -> t
  val plus : t -> t -> t
  val modu : t -> float
end

module Complex : TComplex1 = struct
  ...
end
```

Remarquez la notation exprimant que le module `Complex` est vu comme `TComplex1`. Les signatures peuvent être déclarées dans le fichier d'interface `.mli`.
Notez que l'on peut donner une signature plus générale, par exemple :

```
module type TComplex2 = sig
  type t
  val zero : t
  val cons : float -> float -> t
  val plus : t -> t -> t
end

module Complex : TComplex2 = struct
  ...
end
```

Ici non seulement des fonctions ont été rendues invisibles à un utilisateur de `Complex` mais aussi le type `t` a été abstrait : il est à la discrétion de la personne qui implante les complexes, qui peut choisir d'utiliser un tableau plutôt qu'un couple de deux flottants. La signature vous permet de cacher les détails d'implémentation.

Exercice 5 (Les complexes) *Écrivez les instructions ci-dessous dans un fichier .ml et compiler (ou récupérer le fichier* `essaiModules.ml` *sur ma page web). Tester le programme dans un terminal. Que s'affiche-t-il ?*

```
open Format

module type TComplex = sig
  type t
  val zero : t
  val cons : float -> float -> t
  val oppose : t -> t
  val plus : t -> t -> t
  val modu : t -> float
  val print : t -> unit
end

module Complex : TComplex = struct
  type t = float * float
  let zero = (0.,0.)
  let cons r i = (r,i)
```

```
  let oppose (r,i) = (-.r,-.i)
  let plus (r_1,i_1) (r_2,i_2) = (r_1+.r_2,i_1+.i_2)
  let modu (r,i) = sqrt (r*.r +. i*.i)
  let print (r,i) = printf "%f + %f I" r i
end

let jeu () =
  (
    let c1 = Complex.cons 3.14 2.0 and
c2 = Complex.cons 6.17 (-2.7) in
    let c3 = Complex.plus c1 c2 and
modulc2 = Complex.modu c2 in
      printf "%f" modulc2;
      print_newline();
      Complex.print c3;
      print_newline();
  )

let main()=
 printf "debut\n";
 jeu();
 printf "fin\n";;

main();;
```

Exercice 6 (Écriture d'un module de gestion d'une pile) – *Définir le type module* `TPile` *correspondant à l'interface d'un module fournissant un type polymorphe « pile », ainsi que les fonctions* `pile_vide`, `est_vide`, `push` *et* `pop` *et une exception* `Pile_Vide`.
– *Écrire un module* `Pile` *implantant cette interface.*
– *Utiliser ce module afin de réaliser un reconnaisseur du langage* $a^n b^n$. *L'appel* `dans_langage liste` *retournera* `true` *si le "mot" (liste de caractères) passé en paramètre est dans le langage,* `false` *sinon. On pourra par exemple empiler les 'a' jusqu'à obtenir un premier 'b' et ensuite dépiler un 'a' à chaque fois que l'on rencontre un 'b' dans la liste. On fera attention à bien détecter les cas d'arrêt.*

2.2 Foncteur

Les *foncteurs* sont des fonctions du domaine des modules vers le domaine des modules. Ils permettent la définition de modules *paramétrés par un ou plusieurs autres modules.*
Un foncteur est défini par le mot-clé `functor` , suivi de la signature du module paramètre (ici `P`), puis de la définition du foncteur en fonction de ce paramètre.

```
module type Groupe = sig
  type t
  val zero : t
  val oppose : t -> t
  val plus : t -> t -> t
end

module Matrices = functor (P : Groupe) ->
```

```
struct
  type t = P.t list list
  let plus = List.map2 (List.map2 P.plus)
end
```

Question Que fait la fonction `List.map2` ? [3]

Le résultat de l'instanciation d'un foncteur avec un module respectant la signature donnée (c'est-à-dire comportant *au moins* les types et valeurs de cette signature — ce module peut être plus complet !) est un module, que l'on peut lui-même nommer, utiliser, ouvrir avec le mot-clé `open` comme n'importe quel module.

```
module ComplexMat = Matrices(Complex)
```

Exercice 7 (Écriture d'un foncteur de tri)

1. *Donner la définition, sous forme de type module, d'un* type ordonné, *c'est à dire d'un type sur lequel on peut effectuer des comparaisons.*
2. *Soit le type module :*

   ```
   module type TTri = functor (E : TypeOrdonne) -> sig
     val trier : E.t list -> E.t list
   end
   ```

   Écrire un foncteur `QuickSort` *implémentant l'interface* `TTri` *avec l'algorithme du QuickSort.*
3. *Écrire un foncteur* `TriFusion` *implémentant l'interface* `TTri` *par un tri par arbre binaire de recherche.*
4. *Définir, à l'aide d'un de ces foncteurs et en utilisant le module* `Complex` *vu ci-dessus, un module permettant de trier des nombres complexes par module croissant (on pourra créer un module* `ComplexOrdonnes` *qui fera appel à certaines propriétés de* `Complex`*).*
5. *Définir un module* `ListesOrd` *de type* `TypeOrdonne` *permettant de comparer des listes d'entiers selon la somme de leur contenu. Utiliser les questions ci-dessus pour définir un module* `TriListes` *permettant de trier de tels objets.*
6. *Essayer le code suivant. Que se passe-t-il et pourquoi ?*

   ```
   let l1=[3;-5;6];;
   let l2=[9;0;2];;
   ListesOrd.plus_grand l1 l2;;
   ```

7. *Si l'on veut que cela soit possible, il faut alors "désencapsuler"* `ListesOrd` *et* `TypeOrdonne`*. Faites-le, et re-tester. L'appel au foncteur pour la création de* `TriListes` *marche-t-il encore ? Pourquoi ?*

3. C'est une bonne occasion d'aller voir la documentation en ligne `http://caml.inria.fr/pub/docs/old-311/libref/List.html`.

3 Graphisme en Caml

La bibliothèque `graphics` permet de dessiner des images, de détecter des mouvements de souris et des frappes de touches. Elle permet même de produire des sons !
Une description des fonctions disponibles se trouve à l'URL suivante :

`http://caml.inria.fr/pub/docs/manual-caml-light/node16.html`

Éssayez d'analyser et tester ce que fait le code suivant :

```
#load "graphics.cma";;
Graphics.open_graph "";;
let rec draw l = match l with
  |[] -> ()
  |(x0,y0)::l1 -> Graphics.lineto x0 y0;
                  Graphics.moveto x0 y0;
                  draw l1;;
```

Exercice 8 (Damier)
Écrivez une fonction prenant en entrée deux entiers m et n et qui dessine un échiquier de taille m par n. On utilisera `fill_rect`*, et on construira des carrés de taille 100 par 100. Bonus : affiner ceci en adaptant la taille des carrés (on utilisera les fonctions* `size_x, size_y`*).*

Exercice 9 (Flocon de Koch) *Le flocon de Koch est un exemple très classique de fractale. On l'obtient de la manière suivante :*
- *Partir d'un triangle équilatéral (plutôt grand) ;*
- *Pour chaque côté :*
  - *Le subdiviser en trois segments de longueurs égales ;*
  - *Poser un nouveau triangle équilatéral sur le segment du milieu, à l'extérieur du grand triangle ;*
  - *Effacer ce "segment du milieu".*
- *Répéter récursivement sur les 12 segments restants.*

Écrivez une fonction qui effectue un nombre d'itérations donné de la construction du flocon de Koch, et testez cette fonction sur un nombre significatif d'itérations.

4 Complément sur les modules

4.1 La librairie standard

Le compilateur OCaml est fourni avec une librairie standard comprenant un ensemble de modules, dont par exemple :
– le module *List*, regroupant les opérations standard sur les listes (*map*, *mem*, *sort*...) ;
– le module *Array*, regroupant les opérations standard sur les tableaux ;
– le module *Arg* permettant l'accès aux arguments de la ligne de commande ;
– les modules *Stack* (piles), *Queue* (files), les foncteurs *Set* et *Map* permettant de gérer respectivement des ensembles et des associations d'éléments ordonnés ;
– etc.
La documentation complète de cette librairie standard est accessible par l'url `http://caml.inria.fr/pub/docs/manual-ocaml/libref/`.

4.2 La notion de modules paramétrés dans les langages de programmation

La notion de module paramétré permet une plus grande réutilisabilité du code écrit. Cette notion est présente dans la plupart des langages destinés au développement de gros projets logiciels :

– en Ada, la notion de *paquetage générique* permet de paramétrer un paquetage par des types, des fonctions ou encore d'autres paquetages ;
– en C++, les *templates* permettent la définition de classes paramétrées par des types, des fonctions, ou d'autres classes.
– la notion de *classes génériques* est présente en Java depuis la version 1.5 du langage.

Bien que la notion de classe soit différente de celle de module, l'utilisation pratique de ces deux notions poursuit généralement le même objectif : définir un type abstrait, ainsi que les opérations qui lui sont associées. Les avantages et inconvénients respectifs de ces deux approches sont l'objet d'un débat très ouvert.