Chapitre 5

Les modules et types abstraits de données en caml

En caml [1] comme en C, des programmes conséquents doivent toujours être decomposés en modules. Associer à une approche descendante de la programmation (décomposition recursive d'un problème initial en sous problème plus simple) cela permet de mieux organiser votre code. Mais cela permet surtout la réutilation de code (par vous même) et le dévellopement de bibliothèques réutilisables (par d'autre programmeurs). Les modules doivent en outre pouvoir être compilés séparements (cela pemet de ne pas recompiler tout le code à chaque modification ou permet de travailler à plusieur en parallel). Enfin on doit veiller à restreindre la porté des modifications. C'est-à-dire que si nous fournissons un module à d'autres programmeurs, leur code ne devra pas être sensible à d'evemtuelles modifications ult'erieurs des structures internes de notre module, pouvu que les fonctionalités qu'il propose reste les mêmes.

Il y donc en quelque sorte un contrat entre le fournisseur et l'utilisateur d'un module.

Illustrons cela par un exemple en écrivant un module pour la représentation et la manipulation des nombres complexes.

5.1 Exemple d'un module de manipulation des nombres complexe

5.1.1 Premieres definition

On écrit le fichier `complex.ml` suivant:

```
type complex == float*float;;
(* on defini un synonime pour les paires d'entier *)

let pi=(acos (-1.));;

let module (r,i)=sqrt ((r**2.)+.i**2.);;

let argument (r,i)=
 if (r=0.) then (pi /. 2.)
         else (
            let q= (i /. r) in
            let teta = (atan q) in
```

[1] On décrit ici le système de module de camllight. En Ocaml les modules devient des objets de premiere classe paramètrable et manipulable au travers des foncteurs

```
                 if (teta < 0.) then (if i < 0. then teta
                                else (teta+.pi))
                    else (if (i > 0.) then teta
                                 else pi +. teta));;

let eq (r1,i1) (r2,i2)= (r1 = r2) & (i1 = i2)
(* il faudrait gerer les pb d'arrondi crees par  atan cos ...*);;

let plus (r1,i1) (r2,i2) = r1+.r2,i1+.i2;;
let moins (r1,i1) (r2,i2) = r1-.r1,i1-.i2;;
let mult (r1,i1)(r2,i2) = (r1*.r2-.i1*.i2),(r1*.i2+.r2*.i1);;
let div (r1,i1) (r2,i2)  = let m= (module (r2,i2)) in
                         let diviseur = m**2. in
               ((r1*.r2+.i1*.i1)/.diviseur),((r2*.i2-.i1*.r2)/.diviseur);;

let racine c =
let m=(sqrt (module c))
 and arg = (argument c)/.2. in
               (m*.(cos arg)),(m*.(sin arg));;
(* affichage *)
let print_complex (r,i) = print_float r;
                          print_string" + ";
                          print_float i;
                          print_string"i";;
```

Compilons ce module sous unix par la commande `camlc` avec l'option `-c`.

```
COURS> camlc -c complex1.ml
COURS> ls complex1.*
complex1.ml  complex.zi  complex.zo
```

Cela produit deux nouveaux fichiers `complex.zi` `complex.zo`.

Notre ami "Equation Man" ecrit maintenant un module utilisant notre module `complex`, par exemple un module de résolution d'équation à coéficients dans C contenant en autre la fonction solve qui résoud une équation du second degrès.
Soit le fichier `equation.ml` suivant:

```
#open "complex";; (* on utilise le module complex *)
(* attention les operations sont celle definies sur les complexes *)
let solve a b c = let discriminant=(moins (mult b b) (mult (4.,0.) (mult a c))) in
                  let deuxa=(mult (2.,0.) a) in
                  let racDisc =(racine discriminant) in
                  (div (moins racDisc b) deuxa), (div (plus racDisc b) deuxa);;
```


```
#open "complex";; (* on utilise le module complex *)
(* attention les operations sont celle definies sur les complexes *)
let solve a b c = let discriminant=(moins (mult b b) (mult (4.,0.) (mult a c))) in
```

```
                  let deuxa=((2.*.(fst a)),(2.*.(snd a))) in
                  let racDisc =(racine discriminant) in
                  (div (moins racDisc b) deuxa), (div (plus racDisc b) deuxa);;
```

Il le compile également avec l'option `-c` et peut maintenant l'utiliser soit dasn la boucle d'interaction Caml soit en créant un programe de test qu'il faudra compiler :

1. Dans un programme complet soit le fichier test.ml

```
#open  "equation";;
#open   "complex";;

let main()=
let a=(2.1,3.5) and b= (0.1,3.5) and c=(2.1,6.5) in
print_string "une solution de l'equation ;\n(";
print_complex a;
print_string ")X^2 +(";
print_complex b;
print_string ")X +(";
print_complex c;
print_string ") est :";
print_newline();
print_complex (fst (solve a b c));
print_newline();
exit 0;;


main();;
```

   Appres compilation l'execution produira :

```
COURS> montest
une solution de l'equation ;
(2.1 + 3.5i)X^2 +(0.1 + 3.5i)X +(2.1 + 6.5i) est :
0.130915848728 + 0.255020097472i
```

2. En mode interactif

```
#open "equation";;
load_object"complex.zo";;
load_object"equation.zo";;
....
```

   Apres ces directives la fonction `solve` définie dans le module `equation` est visible directement:

```
#solve;;
- :
 float * float ->
 float * float -> float * float -> (float * float) * (float * float) = <fun>
```

```
mais la fonction {\tt mult} definie dans {\tt complex} ne l'est pas car on n'a pas applique {\tt open} a ce module. On peut acceder a la fonction {\tt mult} en la prefixant du nom du module et de deux {\tt _}.
#mult;;
Toplevel input:
>mult;;
>^^^^
The value identifier mult is unbound.
#complex__mult;;
- : float * float -> float * float -> float * float = <fun>
```

La notation prefixée est utile si l'on veut utiliser plusieurs modules qui contiennent des fonctions de même nom.

Quelques jours plus tard ayant approfondi notre cours sur les complexes on decide de changer notre représentation interne et d'utiliser non plus une paire d'entiers représentants la partie réelle et la partie imaginaire mais le module et l'argument. En consequence on réécrit une nouvelle version du module `complex`. On optient alors le fichier suivant:

Probleme, le module `equation` d'"Equation man" est il toujours utilisable sans modification ? Non bien sur ! par exemple `((2.*.(fst a)),(2.*(snd a)))` ne represente plus 2 a mais un complex dont le le module et l'argument sont le double de ceux de `a`. Les chose sont encore pire dans le fichier principal `test` ou croyant créer `5+i` je crée `2.70151152934+ 4.20735492404i`[2]

5.1.2 Notion d'interface

```
(*  interface d'un module  travaillant sur les nombres complexes
(* la representation interne du type complex est masquee *)

          type complex;;
          value module:complex->float;;
          value argument:complex->float;;
          value plus :complex->complex;;
          value moins :complex->complex;;
          value div :complex->complex;;
          value mult :complex->complex;;
          value racine:complex->complex;;
          value print_complex :complex->unit
```

Remarque 1. sur le synonyme

[2] soit 5...

Structure générale Un module se compose generalement de 2 fichiers :

1. Le fichier nom.mli qui contient la description de l'interface. Elle donne la liste de ce qui sera visible de l'exterieur du module c'est-à-dire ce que le module met à disposition de l'utilisateur. On parle de la specification du module.
2. Le fichier nom.ml donne l'implementation des types et fonctions de l'interface plus eventuelement d'autre type et fonction qui ne sont visibles que de lui.

Utilisation dans d'autre modules

```
#open monModule;;
```

Utilisation en mode interactif

```
#open monModule;;
load_object ‘‘monModule.zo’’;;
```

Chapitre 6

Quelques structures de donnée en ML

6.1 Pile

(predefinies dans stack.ml)

6.2 les Queues

(predefinies dans queue.ml) FIFO

```
(* Stacks *)

(* This module implements stacks (LIFOs), with in-place modification. *)

type 'a t;;
        (* Type des pile contenant des elements de type  ['a]. *)

exception Empty;;
        (* Levee quand  [pop] est applique a la pile vide *)

value new: unit -> 'a t
        (* cree un nouvelle pile initialement vide  *)
  and push: 'a -> 'a t -> unit
        (* [push x s] adds the element [x] at the top of stack [s]. *)
  and pop: 'a t -> 'a
        (* [pop s] efface et renvoie l'element au somet de la pile[s],
           leve  [Empty] si la pile est vide  *)and pop: 'a t -> 'a
  and top: 'a t -> 'a
        (* [top s]  renvoie l'element au sommet de la pile[s],
           leve  [Empty] si la pile est vide  *)
  and clear : 'a t -> unit
        (* Efface tout les element de la pile *)
  and length: 'a t -> int
        (* Donne le nombre d'element de la pile  *)

;;
```

Une implémentation répondant à cette spécification est:

```
type 'a t = { mutable c : 'a list };;

let new () = { c = [] };;

let clear s = s.c <- [];;

let push x s = s.c <- x :: s.c;;

let pop s =
  match s.c with
  | hd::tl -> s.c <- tl; hd
  | []     -> raise Empty;;

let length s = list_length s.c;;

let iter f s = do_list f s.c;;
```

Pour qu'on puisse accéder aux identificateurs du module stack, il faut que ce module soit accessible, c'est-à-dire résidant dans le répertoire de travail actuel ou bien dans la bibliothèque standard de Caml Light, ou bien encore dans l'un des répertoires indiqués sur la ligne de commande du compilateur avec l'option -I. Lors de la création de l'exécutable, il faut également demander l'édition des liens de tous les modules utilisés dans l'application (autres que les modules de la bibliothèque standard).

Les modules permettent évidemment la compilation séparée, ce qui sous le système Unix s'accompagne de l'utilisation de l'utilitaire make. Nous donnons donc un makefile minimum pour gérer un programme découpé en modules. On s'inspirera de ce fichier pour créer ses propres makefiles. Dans ce squelette de fichier make, la variable OBJS est la liste des modules, et EXEC contient le nom de l'exécutable à fabriquer. Pour notre example il vient :

```
CAMLC=camlc -W -g -I .

OBJS= complex.zo equation.zo test.zo
EXEC= montest

all: $(OBJS)
$(CAMLC) -o $(EXEC) $(OBJS)

clean:
rm -f *.z[io] *.zix *~ #*#

depend:
mv Makefile Makefile.bak
(sed -n -e '1,/^### DO NOT DELETE THIS LINE/p' Makefile.bak; \
         camldep *.mli *.ml) > Makefile
rm Makefile.bak
```

```
.SUFFIXES:
.SUFFIXES: .ml .mli .zo .zi

.mli.zi:
$(CAMLC) -c $<
.ml.zo:
$(CAMLC) -c $<

### EVERYTHING THAT GOES BEYOND THIS COMMENT IS GENERATED
### DO NOT DELETE THIS LINE
```

La commande `make all` ou simplement make, refabrique l'exécutable `montest` (ne l'appelez jamais test c'est une commande unix) et `make clean` efface les fichiers compilés. Les dépendances entre modules sont automatiquement recalculées en lançant `make depend`. La commande `camldep` est un script en perl qui se trouve dans `p̃ons/bin` (attention dans le `Makefile` chaque ligne apres les `:` doit commencer par une tabulation et non par des blancs, sinon vous aurait un message du style : `Make: Must be a separator on line x. Stop` ou x est la ligne incriminee.)