UJF **POLYTECH**
RICM1 **Année 2006-07**

Langages et Programmation 2

TP3 - Modules et Foncteurs

Objectifs

Ce TP doit vous permettre de vous familiariser avec la notion de module.

Dans une première partie nous illustrerons leur utilisation par un encodage des nombres complexes, puis par l'implantation d'une pile que l'on utilisera pour reconnaître des langages formels simples.

Dans une deuxième partie, on s'intéressera aux modules paramétrés, appelés également *foncteurs*. Nous implanterons un foncteur de tri.

Il est **fortement conseillé** de tester vos fonctions avec une ou plusieurs entrées quelconques. Toutes les constructions syntaxiques qui vous seront nécessaires sont dans le cours ou dans la documentation en ligne d'OCaml.

1 Les modules

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[1] :

```
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.

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

```
module type TComplex1 = sig
  type t = float × float
  val zero : t
  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`. [2]
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 rendu invisible 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 flottant. La signature vous permet de cacher les détails d'implantation.

Exercice 1 : Les complexes et compilation

1.1. Nous n'allons plus interpréter nos programmes mais les compiler et les exécuter. Commencez par copier le fichier *~gonnordl/tp3/tp3.ml* sur votre compte, puis reportez vous au précis de compilation ocaml (page de Mathias Peron) pour voir comment invoquer *simplement* le compilateur `ocamlc`. Modifiez le type et l'implantation du fichier fourni afin de pouvoir imprimer les complexes. Tester le programme dans un terminal.

Exercice 2 : Écriture d'un module de gestion d'une pile

2.1. 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*.

2.2. Écrire un module *Pile* implantant cette interface.

[2] Penser par exemple, à la manière dont sont définis les paquetages en ADA : un fichier `.ads` contenant l'*interface*, et un fichier `.adb` contenant l'implémentation.

2.3. 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 Foncteurs

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
```

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 3 : Écriture d'un foncteur de tri

3.1. 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.

3.2. 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. Si vous n'avez jamais vu cet algorithme de tri, implémentez un autre algorithme !

3.3. Écrire un foncteur *ABRSort* implémentant l'interface *TTri* par un tri par arbre binaire de recherche. Normalement cet algorithme a été vu en TD.

3.4. Définir, à l'aide d'un de ces foncteur et en utilisant le module *Complex* vu en 1, un module permettant de trier des nombres complexes par module croissant.

3 Pour aller plus loin ...

3.1 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/`.

3.2 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.

Évaluation

Le ou les fichiers correctement numérotés, contenant le numéro du binôme AINSI QUE LEURS NOMS, sont à envoyer à `laure.gonnord@imag.fr` avant le ... mars minuit. L'email devra être EN TEXTE BRUT (pas de html svp) et le sujet sera de la forme `[LP2] tp3 : groupe XX` avec `XX` le numéro de votre groupe.