TD 4 : Programmation modulaire

Pierre Talbot (pierre.talbot@univ-nantes.fr)

16 avril 2019

Objectif(s)

- ✯ Programmer des fonctions et types polymorphiques.
- ✯ Maitriser les modules et la programmation modulaire.
- ✯ Notion de modules paramétrés (foncteurs).

Exercice 1 – Type polymorphique

On considère le type arbre suivant :

```
type 'a tree =
| Empty
| Node of ('a * 'a tree * 'a tree)
```

1. Créer une fonction `map` et `fold` sur cette arbre.
2. Tester vos fonctions et utiliser `fold` pour compter le nombre de nœuds.

Exercice 2 – Programmation modulaire avec Dune

Lorsqu'un projet depasse la taille d'un unique fichier, il devient compliquer de gérer la compilation à la main. Même les Makefile peuvent être compliqués, et il devient plus pratique d'utiliser un système de *build* permettant de compiler automatiquement le projet, gérer les dépendances, faire du pre-processing sur les fichiers (e.g. `ocamlyacc`, `menhir`), etc. On va donc utiliser l'outil `dune` qui est le dernier système de *build* pour OCaml. Une fois le fichier de configuration créer (un unique fichier `dune` par répertoire), nous nous attaquerons à la programmation modulaire en OCaml.

1. Créer le fichier `dune` dans le dossier `src` de votre projet.
2. Créer les fichiers suivants : `hello.ml`, `hello.mli` et `main.ml`.
3. Créer une fonction `hello: unit -> unit` qui affiche "Hello World" et appeler cette fonction du fichier `main.ml`. Compiler et tester.

Exercice 3 – Module Map

Nous allons, dans cet exercice, réimplanter une partie du module `Map` de la librairie standard OCaml. Le module `Map` [1] représente une structure de données arborescente qui permet d'associer à une clé k une valeur v ; et contient les fonctions de manipulation d'une telle structure (création, ajout, recherche, ...).

La structure contenue dans une `Map` est un arbre binaire de recherche dans lequel tout nœud contient une paire (clé,valeur) et tel que tout sous-fils droit d'un nœud (k, v) contient des clés inférieures à k et tout fils droit contient des clés supérieures à k.

1. à ne pas confondre avec la fonction `List.map` !

Notre module qui représente une map possédera une fonction `create` permettant de générer une nouvelle map, ainsi qu'une fonction `add` qui ajoutera une nouvelle association (clé,valeur), et enfin une fonction `find` qui retournera la valeur associée à une clé.

1. Donnez le type de l'arbre binaire de recherche contenu dans une map. Attention, notre arbre doit pouvoir contenir n'importe quelle clé et valeur.
2. Donnez le code de l'implantation du module `Map`.
3. On va améliorer la signature de ce module en paramétrisant le module avec un autre module de comparaison de type

```
module type OrderedType =
  sig
    type t
    val compare : t -> t -> int
  end
```

   Donnez la signature du foncteur Map.Make qui génère un module Map, sachant que la clé d'une Map peut être de n'importe quel type, dès le moment que ce dernier dispose d'une relation d'ordre (et donc que le module qui le représente fournisse une fonction de comparaison entre ses éléments).