TP 1 : Initiation à OCaml

Personnalisation d'emacs avec le mode tuareg. Pour mieux profiter de l'environnement de développement `emacs/ocaml`, éditer le fichier `~/.emacs` afin d'y insérer le contenu du fichier `~mandel/divers/init-tuareg.el` (avec `ctrl-x i`). Cette modification du fichier de configuration d'emacs permettra – dès le prochain lancement de l'éditeur – de charger automatiquement le mode tuareg d'OCaml chaque fois que vous éditerez un fichier `.ml` (un résumé des raccourcis clavier de ce mode se trouve dans le guide de survie).

1 Manipulation de Listes

Le but de cet exercice est de manipuler la structure de liste chaînée. Cette structure de données élémentaire vous sera très utile pour programmer votre compilateur.

1.1 Tri sur les listes

Tri par insertion. Le premier algorithme de tri que vous devez réaliser est celui du *tri par insertion*. Cet algorihme suit naturellement la structure récursive des listes. Étant donnée une liste `l` :

1. si `l` est vide alors elle est déjà triée
2. sinon, `l` est de la forme `x::s` et,
   - on **trie récursivement** la suite `s` et on obtient une liste triée `s'`
   - on **insert** `x` au bon endroit dans `s'` et on obtient une liste triée

Écrire une fonction `insert` polymorphe de type `'a -> 'a list -> 'a list` qui permet d'insérer au bon endroit un élément `x` dans une liste `l`. Ajouter comme argument supplémentaire à cette fonction une relation d'ordre de type `'a -> 'a -> bool` afin que l'insertion ne dépende pas d'une relation particulière. Utiliser cette fonction pour écrire une fonction `trier : ('a -> 'a -> bool) -> 'a list -> 'a list` qui permet de trier une liste `l` suivant une relation d'ordre `r`. Utiliser votre fonction pour trier suivant l'ordre croissant puis décroissant une liste d'entiers.

Tri rapide. On va maintenant réaliser un tri plus efficace connu sous le nom de *Quicksort*. Le principe de cet algorithme est le suivant :

Soit une liste `l` à trier.

1. si `l` est vide alors elle est triée
2. sinon, choisir un élément `p` de la liste (le premier par exemple) nommé **le pivot**
3. **partager** `l` en deux listes `g` et `d` contenant les autres éléments de `l` qui sont plus petits (resp. plus grands) que la valeur du pivot `p`
4. **trier récursivement** `g` et `d`, on obtient deux listes `g'` et `d'` triées

5. on renvoie la liste `g'@[p]@d'` (qui est bien triée)

Écrire une fonction `partage : 'a -> 'a list -> ('a list * 'a list)` pour partager une liste `l` en deux sous-listes contenant les éléments de `l` plus petits (resp. plus grands) qu'une valeur donnée `p`.

En utilisant cette fonction, écrire une fonction `quicksort: 'a list -> 'a list` qui trie une liste selon l'algorithme présenté ci-dessus.

1.2 Codage de la structure d'ensemble

Le but de cet exercice est de réaliser un module paramétré (*foncteur*) pour représenter la structure d'ensemble et de réaliser une instance à l'aide des listes.

Le foncteur Ensemble. On souhaite réaliser une structure de données d'ensemble ayant la signature suivante :

```
module type Set = sig
  type elt
  type t

  val vide : t
  val cardinal : t -> int
  val appartient : elt -> t -> bool
  val compare : t -> t -> int
  val ajout : elt -> t -> t
  val union : t -> t -> t
  val min : t -> elt
  val fold : (elt -> 'a -> 'a)  -> t -> 'a -> 'a
  val for_all : (elt -> bool) -> t -> bool
end
```

Soit la signature suivante définissant un type de données `t` ordonné par une relation d'ordre `compare`.

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

Écrire un module `Ensemble` paramétré par un argument `X` de type `TypeOrdonne` et utilisant la structure de listes pour implanter ces ensembles. Créer un module `S` d'ensemble de chaînes de caratères à partir de ce foncteur, utiliser `S` pour tester toutes les fonctionnalités de vos ensembles. Créer un module `EE` d'ensemble d'ensembles de chaînes de caratères et tester le.

2 Évaluateur d'expressions arithmétiques

Le but de cette section est de réaliser un évaluateur d'expressions arithmétiques. La définition de type suivante en OCaml permet de décrire la syntaxe abstraite de ces expressions :

```
type expr =  Cst  of int
           | Sum  of expr * expr
           | Diff of expr * expr
           | Prod of expr * expr
           | Div of expr * expr
```

Exercice 1 Écrire une fonction `eval_expr` de type `expr -> int` qui calcule la valeur entière d'une expression.

Exercice 2 On étend maintenant nos expressions arithmétiques avec des variables *locales* pour stocker les résultats de calculs intermédiaires. Le type `expr` est étendu avec deux nouveaux constructeurs :

```
type expr =   ...
            | Var of string
            | Letin of string * expr * expr
```

Modifier votre fonction `eval_expr` pour qu'elle prenne comme argument supplémentaire un environnement `env` qui lie les noms des variables locales à leurs valeurs entières. On pourra utiliser un environnement `env` de type `(string * int) list` et la fonction `List.assoc : 'a -> ('a * 'b) list -> 'b` pour rechercher la valeur liée à une variable locale.

Exercice 3 Dans l'exercice précédent, la fonction `eval_expr` se termine en renvoyant l'exception `Not_found`, sans plus de précision, quand l'expression évaluée utilise une variable qui n'existe pas (ou qui n'est pas dans la portée de sa déclaration). Afin d'être plus précis sur l'origine de l'erreur, on définit une nouvelle exception `VarUndef` :

```
exception VarUndef of string
```

Adapter votre fonction `eval_expr` pour lever cette exception lorsqu'une variable est mal utilisée et indiquer ainsi le nom de la variable posant problème.

Exercice 4 On ajoute maintenant la possibilité d'utiliser des variables *globales* dans nos expressions. Le type `instr` suivant décrit les instructions pour affecter une valeur à une variable globale et afficher la valeur d'une expression :

```
type instr =  Set of string * expr
            | Print of expr
```

Écrire une fonction `eval_instr : instr -> unit` qui prend en entrée une instruction et calcule sa valeur en lisant et écrivant si nécessaire dans un environnement liant les noms des variables globales à leurs valeurs entières. On pourra utiliser un environnement `genv` de type `(string * int) list ref` (c'est à dire une référence sur une liste de paires `string * int`).

Parsing. Afin de vous aider à tester vos fonctions `eval_expr` et `eval_instr`, vous pouvez récupérer le module `Parser` en tapant la commande Unix :

```
cp ~mandel/divers/tp1/parser.cm* .
```

Ce module contient la définition des types `expr` et `instr` ainsi que les fonctions `read_expr : string -> expr` et `read_instr : string -> instr` qui permettent respectivement de transformer une chaîne de caractères de type `string` en une valeur de type `expr` ou `instr`.

Les chaînes de caractères reconnues par la fonction `read_expr` seront par exemple de la forme `10*let x=(3+4) in x/2` ou simplement `45+3*8`. Celles reconnues par la fonction `read_instr` seront de la forme `set x=3+(let y=z*4 in y/2)` ou `print x+9`.