Analyse syntaxique

Dans ce TP nous allons programmer pour la première fois un *parseur*, également appelé *analyseur syntaxique*. Le principe est le suivant : on considère une grammaire non-contextuelle et à partir d'une suite de symboles non terminaux, on souhaite reconstruire un arbre de dérivation qui correspond à cette séquence. A l'issu de l'analyse syntaxique, les données sont sous forme d'arbres qui sont bien plus faciles à manipuler que la donnée brute initiale.

On commencera par l'analyse syntaxique d'une grammaire très simple représentant des expressions arithmétiques :

$$S \rightarrow \quad < entier > \quad | \quad (S + S) \quad | \quad (S * S)$$

1 Analyse lexicale

En général les non-terminaux utilisés ne sont pas des simples caractères mais plutôt des *jetons* aussi appelés *lexèmes*. Ce sont des unités lexicales qui peuvent être constituées de plusieurs caractères. Dans l'exemple de la grammaire ci-dessus, les lexèmes sont les opérateurs, les parenthèses mais aussi un <*entier*> qui est formé à partir d'une suite de chiffres.

Un *analyseur lexical* a pour but de lire une suite de caractères et de la traduire en suite de lexèmes. Cette phase est en général réalisée par des automates : les types de lexèmes sont décrits par un expression régulière et convertie en automate capable de les reconnaitre. Comme il est fastidieux de concevoir ces automates à la main, nous allons utiliser l'outil `ocamllex` qui construira automatiquement les automates à notre place.

L'outil `ocamllex` fonctionne à partir d'un fichier `lexer.mll` (par exemple) qui contient les descriptions des différents types de lexème. Ce fichier ressemble à du code OCaml mais n'est pas un programme OCaml. Il a pour format

```
{
    code OCaml preliminaire
}
rule point_entree = parser
| expr1 { sortie1 }
| expr2 { sortie2 }
| ...
{
    code OCaml de fin de fichier
}
```

Les `expr` sont des expressions régulières qui sont attendues et la `sortie` sont les éléments de code `OCaml` à produire lorsque cette expression régulière est reconnue.

Question 1

1. Ouvrir le fichier `lexer.mll` fourni et comprendre son contenu.
2. Compiler ce fichier avec la commande `ocamllex lexer.mll`.
3. L'outil `ocamllex` a construit un fichier source `lexer.ml`, observer son contenu.
4. Compiler le fichier `lexer.ml` avec la commande `ocamlc -c lexer.ml`. Un fichier `lexer.cmo` a été produit.

Pour chaque règle présente dans le fichier `.mll`, le code `.ml` fournira une fonction de type :
`val point_entree : Lexing.lexbuf -> type sortie`

Le `Lexing.lexbuf` attendu est un objet OCaml qui représente l'état du flux d'entrée de l'analyseur. Il y a deux manières principales pour produire un `Lexing.lexbuf` :
`Lexing.from_string : string -> Lexing.lexbuf`
`Lexing.from_channel : Pervasives.in_channel -> Lexing.lexbuf`

la première méthode lit les caractères d'une chaîne de caractères, la seconde méthode lit les caractères d'un canal d'entrée (par exemple `stdin`).

Question 2

Dans l'interprète OCaml, exécuter et comprendre le code suivant :

```
#load "lexer.cmo";;
(* la ligne ci-dessus charge le module Lexer
 a partir du fichier lexer.cmo *)

open Lexer;;
(* permet d'ecrire XXX a la place de Lexer.XXX *)
expr_ar;;
(* Observer le type *)

expr_ar (Lexing.from_string "(12 + 17)*14");;

let liste_jetons s =
    expr_ar (Lexing.from_string s)
;;
```

Il est bien sûr aussi possible de ne pas utiliser la directive `#load` et d'utiliser simplement la compilation séparée. Cependant la méthode présentée est plus agréable car elle permet de facilement observer les résultats obtenus.

2 Une calculatrice simple

À partir de maintenant, on considère que l'on peut utiliser la fonction `liste_jetons` pour produire une liste de `Lexer.jeton` à partir d'une chaîne de caractères.

Question 3

Écrire une fonction :
`verifie_par : jeton list -> bool`
vérifiant si la liste de jetons est correctement parenthésée.

Question 4

Écrire une fonction :
`decoupe : jeton list -> jeton list * jeton * jeton list`
qui prend en entrée une suite de jeton de la forme `(S1 op S2)` et qui retourne le triplet `(S1, op, S2)`. Une exception `Parseur_erreur` pourra être déclenchée en cas d'échec.

On introduit le type

```
type arbre_ar =
    | Const of int
    | Somme of arbre_ar * arbre_ar
    | Produit of arbre_ar * arbre_ar
;;
```

Question 5

Écrire une fonction
`parseur : jeton list -> arbre_ar`
prenant en entrée une liste de lexèmes et construisant un arbre de dérivation associé.

Question 6

Combiner tout ce qui précède pour obtenir une fonction calculatrice
`calc : string -> int`
capable de calculer la valeur d'une expression arithmétique.

Question 7

Améliorer cette calculatrice. On pourra par exemple :
1. ajouter l'operateur % (modulo)
2. permettre qu'elle prenne en compte les entiers négatifs
3. travailler sur les flottants plutot que les entiers

3 Logique propositionnelle

Maintenant que l'on a mieux compris le fonctionnement de l'outil `ocamllex` et que l'on sait écrire un petit parseur, on peut s'attaquer à une grammaire légèrement plus complexe :

$$S \rightarrow \quad true \quad | \quad false \quad | \quad !S \quad | \quad (S \,\&\, S) \quad | \quad (S \mid S) \quad | \quad (S -> S)$$

Question 8

1. Concevoir le fichier `lexer.mll` adequat.
2. Dans un fichier `logique.ml` introduire un type `formule` pour représenter les formules de la logique propositionnelle sans variable.
3. Dans un fichier `logique.ml` écrire le code permettant d'aboutir à une fonction

```
parseur : jeton list -> formule
```

4. En déuire une fonction

```
eval : string -> bool
```

prenant en entrée une formule sans variable sous forme de chaîne de caractères et retournant la valeur de vérité de cette formule.

Question 9

En utilisant la compilation séparée, obtenir un executable `eval_logique` évaluant une formule logique propositionnelle sans variable fournie en entrée standard et retournant sa valeur de vérité. On pourra l'exécuter par exemple avec la commande `./eval_logique < (echo '!(true & false)')`