TP Compilateur

L'objectif du TP est de créer un compilateur pour un langage de programmation simple : c'est-à-dire, dans ce cas, de passer de fonctions simples calculant des expressions mathématiques vers un assembleur simplifié. Les expressions du langage suivent la grammaire formelle :

- $G = (\Sigma, V, S, R)$;
- $L = \{a, b, c, ..., z\}$;
- $W = L^+$;
- $\Sigma = \{+, -, *, /, (,), [,]\} \bigcup \mathbb{Z} \bigcup L \bigcup \{,\}$;
- $V = \{S, E, F\}$;
- où les règles R sont les suivantes :
  - $S \rightarrow [J]E$
  - $J \rightarrow \varepsilon | F, W$
  - $E \rightarrow T | E + E | E - E$
  - $T \rightarrow F | E / E | E * E$
  - $F \rightarrow \mathbb{Z} | W | (E)$

Ainsi, S dénote les fonctions, F dénote les arguments de fonctions, E et T dénotent les opérations mathématiques du langage et ont été construits de manière à avoir, comme on le voudrait intuitivement, une hiérarchie des opérations.

1 Analyse Lexicale, le Tokenizeur

La tokenisation lexicale est la conversion d'un texte en une suite de **tokens lexicaux** sémantiquement ou syntaxiquement significatifs, appartenant à des catégories définies par un programme appelé un "lexeur". Ici, l'ensemble des catégories sera défini par un type somme en OCaml qui sera le suivant :

```
type token =
  | ADD | SUB | MUL | DIV | LPAR | RPAR | RBRA | LBRA
  | VAR of string
  | CONST of int
  | EOF
  | COMMA;;
```

Ce qui signifie que notre langage pourra performer des additions, des soustractions, des multiplications, des divisions. De plus, des parenthesages (dénotés par LPAR et RPAR). L'objectif est de pouvoir implémenter des fonctions, c'est pour cela qu'on a rajouté des crochets qui dénotent la déclaration des arguments.
On a de plus ajouté un **token** dénotant les constantes qui sont des variables "immédiates" (des nombres), et un **token** dénotant les arguments de la fonction qui seront d'ailleurs délimités par des virgules (COMMA). Pour simplifier les tâches suivantes, on rajoute un **token** "end of file" (EOF).

1. Écrire une fonction :

   ```
   let rec explode(code: string) : char list;;
   ```

   qui prend en entrée une chaine de caractères et renvoie le tableau de caractères associé à cette dernière.

2. En remarquant que les arguments **doivent** être dénotés par des chaines de lettres, écrire des fonctions :

   ```
   let rec tokenize(code: char list) (liste_tokens: token list) : token list
   and lexW(code: char list) (token: string) (liste_tokens: token list)  : token list
   and lexZ(code: char list) (token: string) (liste_tokens: token list)  : token list;;
   ```

   qui implémentent respectivement la mise sous forme de tokens de la liste de caractères **code**, des arguments (W pour words), et des valeurs immédiates (Z pour dénoter $\mathbb{Z}$).

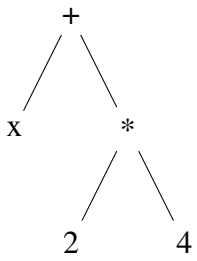
3. En déduire une fonction :

   ```
   let lex(code: string) : token list;;
   ```

   Qui effectue la mise sous forme de tokens d'une fonction du langage, prise en argument en une chaine de caractères.

2 Analyse syntaxique, le Parser

L'analyse syntaxique ou **parsing**, est le procédé d'analyse d'une chaine de caractères, selon les règles d'une grammaire formelle. Les règles de la grammaire formelle du langage ont été définies plus haut. Pour cela, nous utiliserons la structure de donnée d'Arbres de Syntaxe Abstraite, **AST** (Abstract Syntax Tree en anglais) qui sont des structures utilisées massivement dans les compilateurs pour représenter la structure d'un code. Ces structures servent à une représentation intermédiaire du programme à travers plusieurs étapes de compilations requises.
Un exemple d'**AST** (dénotant $3+2*4$) :



L'objectif de cette partie est donc de passer d'une liste de **tokens** construite dans la première partie à un **AST**.
On définit ainsi le type :

```
type ast =
  | Imm of int (* valeur immediate *)
  | Arg of int (* reference au nieme argument *)
  | Add of ast * ast (* addition des deux sous-arbres *)
  | Sub of ast * ast (* soustraction des deux sous-arbres *)
  | Mul of ast * ast (* multiplication des deux sous-arbres *)
  | Div of ast * ast (* division des deux sous-arbres *);;
```

1. Écrire des fonctions :

```
let rec parseS(tl: Token.token list) : Ast.ast * Token.token list;;
let rec parseJ(tl: Token.token list) : string list * Token.token list;;
let rec parseE(tl: Token.token list) (args: (string * int) list) : Ast.ast * Token.token list;;
let rec parseT(tl: Token.token list) (args: (string * int) list) : Ast.ast * Token.token list;;
let rec parseF(tl: Token.token list) (args: (string * int) list) : Ast.ast * Token.token list;;
```

   qui réalisent l'analyse syntaxique pour chacun des symboles de la grammaire.

2. En déduire une fonction :

```
let parse(tl: Token.token list) : Ast.ast
```

   qui réalise le **parsing** d'une liste de **tokens** tl.

3 Optimisation : Simplifications d'ASTs

Remarquons que l'**AST** produit dans les questions précédentes est simplifiable pour des cas de calculs avec des valeurs constantes comme cela était le cas pour le sous-arbre droit de l'exemple partie 2.

1. Écrire une fonction :

```
let rec opti_constants(ast: Ast.ast) : Ast.ast;;
```

   qui simplifie l'**AST** passé en argument dans le cas d'opérations entre valeurs constantes.

4 Simulation d'assemblage

L'**assemblage** est une phase de la compilation qui consiste à transformer du code assembleur en fichier binaire. L'objectif ici est de simuler un pseudo-assembleur muni de deux registres R0 et R1 et des instructions suivantes :

```
IM n : charge la valeur constante n dans R0
AR n : charge le n-ieme argument dans R0
SW : echange les valeurs contenues dans les 2 registres
PU : met la valeur contenue dans R0 sur la pile
PO : retire un element de la pile et le stocke dans R0
AD : ajoute R1 a R0 et met le resultat dans R0
SU : soustrait R1 de R0 et met le resultat dans R0
MU : multiplie R0 par R1 et met le resultat dans R0
DI : divise R0 par R1 et met le resultat dans R0
```

Pour cela, on va traduire les données de notre **AST** en ce pseudo-assembleur puis interpréter chacune des lignes de la liste d'instructions produite précédemment.

1. Écrire une fonction :

```
let assemble(ast: Ast.ast) : Instr.t list;;
```

   qui transforme l'**AST** en une liste d'instructions assembleur performant les mêmes opérations et renvoyant le résultat dans R0.

2. Écrire une fonction :

```
let simulate(prgm: Instr.t list) (args: int array) : int;;
```

où args représente les valeurs que l'on veut assigner à chaque argument de la fonction lors de l'exécution.

—— Fin du TP! ——