I53: Compilation et théorie des langages
TP 2

Semestre 2019-20

1 Analyseur d'expressions airthmétiques élémentaire

Le but de cette section est d'écrire un premier traducteur en **langage Python** mettant en oeuvre la méthode de la descente récursive et de la traduction dirigée par la syntaxe. Pour cela nous allons commencer par écrire un analyseur syntaxique pour une grammaire élémentaire puis nous l'enrichirons au fur et à mesure.

1. Commençons par considérer la grammaire $G = \{T, N, \mathcal{P}, S\}$ des expressions contenant seulement des + et des -:

   - $T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}$
   - $N = \{Expr\}$
   - $S = Expr$
   - $\mathcal{P}$:

     $$\begin{array}{rcl} Expr & \rightarrow & Expr + Expr \\ & | & Expr - Expr \\ & | & \texttt{0|1|2|3|4|5|6|7|8|9} \end{array}$$

   Réécrire la grammaire précédente pour respecter l'associativité à gauche des opérateurs et **supprimer la récursivité à gauche**.

2. Écrire trois fonctions `expr()`, `reste()` et `terme()` mettant en oeuvre l'algorithme de la descente récursive. Le programme devra prendre comme entrée le premier argument de la ligne de commande et affiche `chaine valide` ou `erreur syntaxique`. En cas d'erreur, le caractère fautif devra être affiché.

3. Ajouter des actions sémantiques afin que le programme crée une liste contenant l'écriture postfixée de l'entrée.

4. Étendre l'analyseur précédent aux opérateurs $\times$ et / ainsi qu'aux parenthèses en respectant l'ordre de priorité.

2 Traducteur complet

Le but de cette partie est de réaliser un traducteur complet gérant tous les opérateurs précédents ainsi que des nombres entiers de taille arbitraire. Pour cela le programme comportera trois parties:

- un analyseur lexical
- un analyseur syntaxique chargé de transformer la chaine d'entrée en expression posfixée
- un producteur de code utilisant une pile pour produire un fichier contenant du code à trois adresses.

2.1 Analyseurs lexical et syntaxique

1. Écrire une fonction `analex(s)` qui prend en paramètre une chaîne de caractère et retourne une liste d'unités lexicales. On considère pour cela le lexique suivant:

   `NOMBRE` $\rightarrow$ `0|1|2|3|4|5|6|7|8|9`
   `OP` $\rightarrow$ `+ | - | * | /`
   `PAR_OUV` $\rightarrow$ `(`
   `PAR_FER` $\rightarrow$ `)`

   Par exemple pour l'entrée $5 - (9 + 2 * 3)$ la fonction retournera `[('NOMBRE',5),('OP','-'), ('PAR_OUV','('), (NOMBRE,9), ('OP', '+'), ('NOMBRE',2), ('OP','*'), ('NOMBRE', 3), ('PAR_FER',')')]`

2. Modifier l'analyseur pour que celui-ci ignore les blancs.

3. Étendre la définition de `NOMBRE` aux entiers de taille quelconque.

4. Reprendre l'analyseur précédent pour que celui-ci gère désormais une liste d'unités lexicales et retourne une liste correspondant à la notation posfixée de la chaînes d'entrée.

2.2 Production de code

1. Écrire une fonction chargée de prendre en entrée la sortie de l'analyseur syntaxique et de produire un fichier Python contenant un code à trois adresses traduisant l'évaluation de l'expression avec une pile. Par exemple la chaîne $9 - (5 + 2)$ produira *in fine* le code:

```
t1 = 9
t2 = 5
t3 = 2
t2 = t2 + t3
t1 = t1 - t2
print(t1)
```

2. Modifier le programme complet pour que celui-ci prenne en entrée un fichier contenant l'expression à traduire et produise en sortie programme Python nommé `a.out` exécutable par `./a.out`.

3 Analyseur d'expressions booléennes

Refaire le même travail avec la grammaire des expressions booléennes:

- $T = \{VRAI, FAUX, OU, ET, NON, (,)\}$
- $N = \{Expr\}$
- $S = Expr$
- $\mathcal{P}$:

$$\begin{array}{rcl} Expr & \rightarrow & Expr\ OU\ Expr \\ & | & Expr\ ET\ Expr \\ & | & NON\ Expr \\ & | & (Expr) \\ & | & VRAI \mid FAUX \end{array}$$