Partiel de compilation

M1 Informatique Fondamentale, ENS Lyon
Vendredi 22 novembre 2013

Durée: 2 heures. Seules les notes de cours sont autorisées.
Ce partiel est constitué de 3 exercices indépendants et d'une annexe.
L'exercice 2 doit être obligatoirement rédigé sur une feuille séparée.

Exercice 1. *Analyse lexicale*

Sur l'alphabet $\Sigma = \{+, ., 0, 1, e\}$, on considère la **description lexicale** avec les lexèmes suivants:

- **Lexème -1-:** $+$
- **Lexème -2-:** $(0|1)^+$, *exemple: 101*
- **Lexème -3-:** $1.(0|1)^+e(0|1)+$, *exemple: 1.0e10*

L'analyseur lexical vu en TP produit l'automate donné en annexe.

Q1) En quelques phrases, expliquez comment cet automate est produit.

Q2) Détaillez l'exécution de l'analyseur lexical sur les entrées:

- 1.+$
- 10+1.01e10$

$ dénote le caractère de fin d'entrée (EOF).

Exercice 2. *Analyse syntaxique*

Pour cet exercice, rédigez vos réponses sur une feuille séparée.

On considère la grammaire suivante:

$$\begin{array}{rcl} S & \rightarrow & T \mid I \\ T & \rightarrow & \texttt{ID} \mid \texttt{-}\ T \\ I & \rightarrow & \texttt{INT} \mid \texttt{-}\ I \end{array}$$

Où `ID` désigne une chaîne de caractères, `INT` un entier et "`-`" le symbole *moins*.

Q3) Cette grammaire est-elle $LL(k)$? Si oui, pour quel k?

Q4) Donnez une grammaire $LL(1)$ équivalente.

Il est souvent fastidieux de transformer la grammaire. On veut pouvoir effectuer l'analyse syntaxique en utilisant la grammaire originale.

La figure ci-dessous donne l'automate généré par ANTLR[1] pour le non-terminal S (T et I se comportent comme dans l'analyseur $LL(1)$ original).

Q5) Quel est le sens des transitions quittant q_2? Tester sur "`-----1`".

Q6) Si l'entrée ne contient jamais "`--`", cet automate se comporte comme un analyseur $LL(k)$. Pour quel k?

[1] Un générateur d'analyseurs syntaxiques répandu

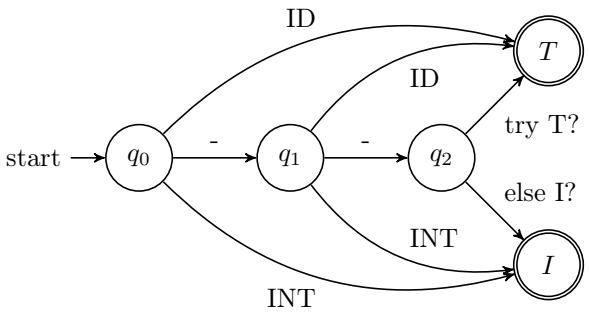

Figure 1: Automate avec *backtracking* pour le non-terminal S

Exercice 3. *Traduction dirigée par la syntaxe*

En C, les variables locales sont obligatoirement déclarées dans le bloc principal du corps de fonction (voir programme de gauche ci-dessous). On veut déclarer des variables locales dans n'importe quel sous-bloc (voir programme de droite) avec les règles usuelles de portée:

- La *portée* d'une déclaration de variable (partie de code où la variable existe) dans un bloc se limite à ce bloc et ses sous-blocs.
- Si une instruction est dans la portée de plusieurs déclarations utilisant le même identificateur (programme de droite, variable `x`), on considère la déclaration la plus profonde (programme de droite, variable `x` du sous-bloc).

```
int foo(int n)
{ //Bloc principal
  int[3] t;
  int i,x;
  ...
}
```

```
int foo(int n)
{ //Bloc principal
  int[3] t;
  int i,x;
  for(i=0; i<N; i++)
  { //Sous-bloc
    int[2] x;
    ...
  }
}
```

Q7) Dessinez l'enregistrement d'activation de `foo`, programme de gauche.

Q8) Donnez la règle de traduction pour un sous-bloc $[\![$ { $\text{type}_1 \text{id}_1$; ...; $\text{type}_n \text{id}_n$; BODY } $]\!]_\rho$. On utilisera récursivement $[\![$ BODY $]\!]_{\rho'}$ sur un environnement ρ' à définir.

Q9) Dessinez l'enregistrement d'activation après être entré dans le bloc du `for`, programme de droite.

Annexe.