Université Joseph Fourier — **Polytech Grenoble**
Département RICM 2 — **Année 2004-05**

Langages et Traducteurs

Quick du jeudi 4 novembre 2004

(indications de correction)

On considère un langage de commandes, noté L_1, construit sur le vocabulaire { `c`, `;`, `*`, `(`, `)`, `$` }. Intuitivement, "`c`" représente une commande de base, "`;`" un opérateur binaire infixé de composition séquentielle, "`*`" un opérateur unaire post-fixé d'itération. Les parenthèses permettent de grouper plusieurs commandes et "`$`" est le caractère de fin de fichier.

La syntaxe complète du langage est donnée par la grammaire G_1 suivante :

$$\begin{array}{lcl} Z & \longrightarrow & C\,\$ \\ C & \longrightarrow & \texttt{c} \mid C\,\texttt{;}\,C \mid C\texttt{*} \mid (C) \end{array}$$

Question 1 (3 pts).
Montrez (sur une phrase très simple du langage) que G_1 est une grammaire ambigüe.

Il suffit de trouver une phrase de G_1 qui admette deux arbres de dérivation distincts.
Par exemple "`c ; c ; c`".

Question 2 (3 pts).
On complète maintenant la définition du langage L_1 en considérant :
– que l'opérateur "`*`" est plus prioritaire que l'opérateur "`;`".
– que l'opérateur "`;`" est associatif à gauche ("`c ; c ; c`" doit être interprété comme "`(c ; c) ; c`").
On note L_2 le langage obtenu en prenant en compte ces nouvelles contraintes.

1. *Proposez une grammaire G_2, non ambigüe, qui décrive le langage L_2.*
2. *Donnez l'arbre de dérivation produit par G_2 pour la séquence "`c ; c ; c* $`".*

Grammaire G_2 :

$$\begin{array}{lcl} Z & \longrightarrow & C\,\$ \\ C & \longrightarrow & C\,\texttt{;}\,T \mid T \\ T & \longrightarrow & T\texttt{*} \mid (C) \mid \texttt{c} \end{array}$$

On vérifie en construisant l'arbre de dérivation dans G_2 de "`c ; c ; c* $`" que "`;`" est associatif à gauche et que "`*`" est plus prioritaire que "`;`".

Question 3 (5 pts).
La grammaire G_2 que vous avez proposé à la question 2 est-elle LL(1) ? Si non transformez-là en une grammaire G_2' qui soit LL(1) et qui décrive L_2. Justifiez vos réponses.

G_2 n'est pas LL(1). En effet :

$$\begin{array}{lcl} \texttt{c} & \in & Dir(C \longrightarrow C\,\texttt{;}\,T) \\ \texttt{c} & \in & Dir(C \longrightarrow T) \end{array}$$

Ces deux ensembles de directeurs ne sont donc pas disjoints.

On contsruit G'_2 en éliminant la récursivité à gauche dans G_2 :

$$\begin{array}{rcl} Z & \longrightarrow & C\,\$ \\ C & \longrightarrow & T\,T' \\ T' & \longrightarrow & \varepsilon \mid ;T \\ T & \longrightarrow & \texttt{c}T'' \mid (C)\,T'' \\ T'' & \longrightarrow & \varepsilon \mid *T'' \end{array}$$

On vérifie alors que G'_2 est bien LL(1) :

$$\begin{array}{rcl} Dir(T' \longrightarrow \varepsilon) & = & Suivant(T') \\ & = & Suivant(C) \\ & = & \{), \$\} \\ Dir(T' \longrightarrow ;T) & = & \{;\} \\ Dir(T'' \longrightarrow \varepsilon) & = & Suivant(T'') \\ & = & Suivant(T) \\ & = & Premier(T') \cup Suivant(C) \\ & = & \{), \$, ;\} \\ Dir(T'' \longrightarrow *T'') & = & \{*\} \end{array}$$

Les autres règles ne posent pas de problème.

Question 4 (5 pts).
La grammaire G_2 que vous avez proposé à la question 2 est-elle SLR(1), LALR(1), et/ou LR(1). Justifiez votre réponse.

On vérifie facilement en construisant l'automate que cette grammaire est SLR(1).

Question 5 (4 pts).
On cherche maintenant à obtenir pour ce langage de commandes une grammaire qui soit LR(0).

1. *Montrez que ce n'est pas possible pour le langage L_2 (on ne demande pas une démonstration complète, mais un argument pertinent expliqué en quelques lignes).*
2. *Proposez alors un langage L3, qui soit une extension du langage L_2 (qui conserve les mêmes opérateurs et les mêmes règles de priorité), et qui puisse être décrit par une grammaire LR(0) que vous préciserez. Vous pouvez étendre le vocabulaire si vous le jugez utile.*

$L2$ doit contenir des phrases du type "`c ; c ; ... ; c`" et "`c ; c ; ... ; c *`", sachant que `*` est plus prioritaire que `;`. Par conséquent, aprés lecture de la séquence de "`; c`", un analyseur ne peut pas savoir (sans regarder un symbole en avant dans la chaîne d'entrée) s'il doit :
– appliquer une réduction (pour reconnaitre "`c ; c`")
– ou lire un lexème de plus (pour reconnaitre "`c *`").
Une manière simple d'obtenir une grammaire LR(0) consiste à ajouter des *délimiteurs* (par exemple des crochets `[` et `]`, ou des `begin` et `end`) pour fixer la portée de l'opérateur `*`.

$$\begin{array}{rcl} Z & \longrightarrow & C\,\$ \\ C & \longrightarrow & C\,;T \mid T \\ T & \longrightarrow & [T]* \mid (C) \mid \texttt{c} \end{array}$$