Licence ST Informatique – S5 – 2005/2006

Compilation – Examen 2ème session

vendredi 24 février 2006

documents autorisés, calculettes interdites — durée 3 heures

Exercice 1 : On considère la grammaire G_0 suivante, d'axiome P', d'alphabet terminal $\{ + , - , \textbf{,} , \textbf{i} \}$ et dont les règles sont :

$$
\begin{array}{lcl}
P' & \longrightarrow & P \\
P & \longrightarrow & S\ N \\
N & \longrightarrow & E\ \textbf{,}\ I \mid I \\
E & \longrightarrow & I \mid \varepsilon \\
S & \longrightarrow & + \mid - \mid \varepsilon \\
I & \longrightarrow & \textbf{i}
\end{array}
$$

Question 1 : Construire les ensembles *premiers* et *suivants* associés aux variables de la grammaire G_0.

Question 2 : La grammaire G_0 est-elle $LL(1)$, $SLR(1)$, ambiguë ? Justifier.

Question 3 : Trouver une grammaire régulière engendrant le même langage que la grammaire G_0.

Exercice 2 : Soit la grammaire G_1 suivante engendrant des listes *à la Scheme*, d'axiome L, d'alphabet terminal $\{a,\ (\ ,\ )\ \}$ et dont les règles sont :

$$
\begin{array}{lcl}
L & \longrightarrow & (\ S\ ) \mid a \\
S & \longrightarrow & L\ S \mid \varepsilon
\end{array}
$$

Question 1 : Construire l'automate LR(0) de la grammaire G_1.

Question 2 : La grammaire G_1 est-elle LR(0), SLR(1) ? Justifier.

Question 3 : Détailler le déroulement de l'analyseur SLR(1) pour la grammaire G_1 sur l'entrée `"( a ( ) ) $"`.

Question 4 : Décorer la grammaire G_1 d'actions sémantiques permettant de déterminer si une liste contient au moins une occurrence de la liste vide. Par exemple, la réponse doit être positive pour les listes `( )` ou `( a ( a ( ) ) a ( a ) )` ; elle est négative pour `( a ( a ( a a ) ) a ( a ) )`.

Exercice 3 : Dans le cadre du développement d'applications réparties à base de composants logiciels réutilisables, on souhaite décrire l'architecture de l'application à l'aide d'un langage permettant de construire de nouveaux composants par assemblage d'autres composants.

Un composant logiciel est caractérisé par un nombre de connexions d'entrée et un nombre de connexions de sortie. Une représentation graphique d'un composant élémentaire est donnée figure 1. Textuellement, on représentera un composant élémentaire par un couple <*nombre de connexions d'entrée*;*nombre de connexions de sortie*>. Par exemple, le composant de la figure 1 s'écrit `< 3 ; 2 >`.

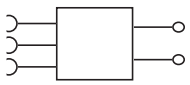


FIG. 1 – Un composant logiciel à 3 entrées et 2 sorties

Pour assembler les composants, on dispose de 2 opérations élémentaires :

La mise en série : La mise en série d'un composant c_1 et d'un composant c_2 correspond à la connexion des sorties de c_1 aux entrées de c_2. Cette mise en série n'est possible que si le nombre de sorties de c_1 est égal au nombre d'entrées de c_2. On obtient alors un composant dont le nombre d'entrées est le nombre d'entrées de c_1 et le nombre de sorties est le nombre de sorties de c_2. Par exemple `< 3 ; 2 > < 2 ; 2 >` est une représentation textuelle de la mise en série de la figure 2. En revanche, `< 3 ; 2 > < 3 ; 2 >` n'est pas une représentation de mise en série valide.

La mise en parallèle : La mise en parallèle d'un composant c_1 et d'un

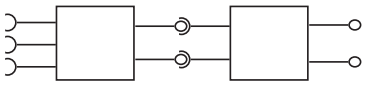


FIG. 2 – Une mise en série de 2 composants

composant c_2 correspond à la construction d'un nouveau composant comportant un nombre d'entrées égal à la somme du nombre d'entrées de c_1 et de c_2 et un nombre de sorties égal à la somme du nombre de sorties de c_1 et de c_2. On représentera textuellement la mise en parallèle de deux composants sous la forme d'un opérateur binaire associatif noté `&`. Par exemple `< 3 ; 2 > & < 2 ; 2 >` est une représentation textuelle de la mise en parallèle de la figure 3.

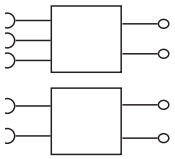
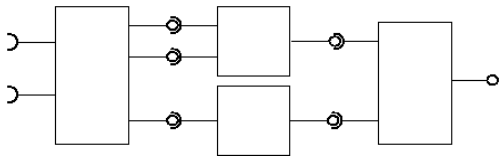


FIG. 3 – La mise en parallèle de 2 composants

La syntaxe du langage d'assemblage de composants est donnée par la grammaire $G = (X, V, C, P)$ où $X = \{ <, >, ;, \&, (,), \texttt{int} \}$, $V = \{ C \}$, et $P = \{C \longrightarrow CC \mid C \,\&\, C \mid (\, C\,) \mid <\texttt{int} \;;\; \texttt{int}>\}$

Question 1 : On précise que la mise en série est prioritaire sur la mise en parallèle. Donner une représentation textuelle pour l'assemblage de composants suivant :



Question 2 : Dire pourquoi la grammaire G n'est pas LL(1).

Question 3 : En respectant la priorité des deux opérateurs d'assemblage de composants, construire une grammaire G' qui soit LL(1) et équivalente à la grammaire G.

Question 4 : Construire les ensembles *premiers* et *suivants* associés aux variables de G' et construire sa table d'analyse LL(1).

Une représentation textuelle d'un assemblage peut être syntaxiquement correcte (c'est à dire engendrable par la grammaire G), mais sémantiquement incorrecte par rapport aux conditions à respecter pour la mise en série. C'est le cas par exemple pour l'expression suivante :

```
< 2 ; 3 > ( < 1 ; 2 > < 2 ; 1 > & < 1 ; 3 > )
```

Question 5 : Décorer votre grammaire G' d'actions sémantiques vérifiant la validité d'une représentation textuelle d'assemblage de composants, avec déclenchement d'une exception en cas d'expression invalide. Pour chaque variable, indiquer les attributs qui leur sont associés en donnant leur type et en précisant s'il s'agit d'attributs synthétisés ou hérités.

On suppose disposer d'une classe *Java* d'analyse lexicale disposant des fonctionnalités suivantes

```
package analyse ;
public class Lexeur {
  public static final Token ET_COMMERCIAL ...
  public static final Token OUVRANTE ...
  public static final Token FERMANTE ...
  public static final Token INFERIEUR ...
  public static final Token SUPERIEUR ...
  public static final Token POINT_VIRGULE ...
  public static final Token ENTIER ...
  public static final Token FIN ... // token $
  // avance au token suivant
  public void lire()...
  //retourne le token courant
  public Token  tokenCourant() ...
  // retourne le texte en entrée correspondant
  public String tokenImage()...
}
```

Question 6 : Construire un analyseur récursif descendant correspondant à la grammaire attribuée de la question 3.5.