Licence ST Informatique – S5 – 2005/2006

Compilation – Examen 1ère session

mardi 10 janvier 2006

documents autorisés — durée 3 heures

Exercice 1 : On considère la grammaire $G_0 = (X_0, V_0, S_0, P_0)$ où $X_0 = \{ a , b \}$, $V_0 = \{ S_0 , S_1 \}$, et $P_0 = \{S_0 \longrightarrow S_1\ S_0\ S_1 \mid b,\ S_1 \longrightarrow a\ S_1\ a \mid \varepsilon\}$.

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

Question 2 : Trouver une grammaire sous forme normale de Chomsky engendrant le même langage que la grammaire G_0.

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 :
$L \longrightarrow (\ S\ ) \mid a$
$S \longrightarrow L\ S \mid \varepsilon$

Question 1 : Construire les ensembles *premiers* et *suivants* pour G_1.

Question 2 : Construire la table d'analyse LL(1) de la grammaire.

Question 3 : Décorer la grammaire G_1 d'actions sémantiques permettant d'imprimer chaque occurrence d'atome d'une liste avec son numéro de Dewey comme donné dans l'exemple suivant :

Exemple 1 *Pour la liste* `( ( x y ) x ( y ) ( y ( x y ) ) )`*, l'impression doit donner :*

```
1.1.1 : x
```

```
1.1.2 : y
1.2 : x
1.3.1 : y
1.4.1 : y
1.4.2.1 : x
1.4.2.2 : y
```

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

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

Question 4 : Construire un analyseur récursif descendant correspondant à la grammaire attribuée de la question 2.3.

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 >`.

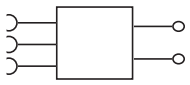


FIG. 1 – Un composant logiciel à 3 entrées et 2 sorties

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.

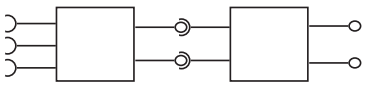


FIG. 2 – Une mise en série de 2 composants

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

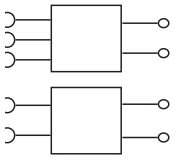


FIG. 3 – La mise en parallèle de 2 composants

Pour décrire la syntaxe du langage d'assemblage de composants, on se donne la grammaire $G = (X, V, C, P)$ où $X = \{ <, >, ;, \&, (,), \texttt{int} \}$,

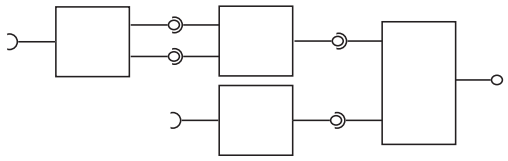
$V = \{ C , P , S , E , I , O\}$, et $P = \{$

$C \longrightarrow P$
$P \longrightarrow \mathrm{P}$ `&` $S \mid S$
$S \longrightarrow S\ E \mid E$
$E \longrightarrow$ `<` I `;` O `>` $\mid$ `(` P `)`
$I \longrightarrow$ `int`
$O \longrightarrow$ `int`

$\}$.

Question 1 : D'après la grammaire G, quelle opération est la plus prioritaire : la mise en série ou la mise en parallèle ?

Question 2 : Donner une représentation textuelle pour l'assemblage de composants suivant :



Question 3 : Trouver les ensembles *premiers* et *suivants* associés aux variables de la grammaire G.

Question 4 : Construire l'automate LR(0) pour la grammaire G. La grammaire G est-elle LR(0), SLR(1), ambiguë ? Justifer.

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 la grammaire G d'actions sémantiques vérifiant la validité d'une représentation textuelle d'assemblage de composants, avec declenchement 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.