Maîtrise d'Informatique

COMPILATION 2

Durée 3 heures, documents manuscrits autorisés

Session de janvier 1999

Problème 1 (5 Points)

On utilise la grammaire du langage `Java` donnéee en Annexe, dans laquelle le symbole : signifie $\rightarrow$, et où les non-terminaux sont représentés par des mots. Un non-terminal suivi par opt signifie qu'il est optionnel, ce qui dans la notation du cours se traduit par deux règles : l'une où le non terminal apparaît et l'autre où il n'apparaît pas.

Question 1. Donner l'arbre de dérivation de la partie de programme suivante :

```
Table() {toto[] x;}
```

On utilisera comme axiome le non-terminal `ConstructorDeclaration`

Question 2. Donner l'arbre de dérivation de la partie de programme suivante

```
Table() {toto[a] = x}
```

On utilisera le même axiome que précédemment

Question 3. On suppose que dans la grammaire précédente les règles concernant `ArrayType`, `ArrayAccess` et `Type` sont remplacées par :

```
ArrayType :
   Type []

ArrayAccess :
   Name [ Expression ]

Type :
   Name
   NumericType
```

En utilisant les deux exemples ci-dessus, montrer de manière très informelle que la grammaire n'est pas alors LR(1).

Problème 2 (8 points)

Dans le contexte de l'analyse du génome, on travaille sur des chaînes de caractères formées paR des symboles A, C, G, T, et on recherche des propriétés satisfaites par les sous-séquences de ces chaînes . Pour suivre les notations du cours on considère l'alphabet $A = \{a, b, c, d\}$; d'autre part étant donnés deux mots f et g de A^*, on dit que g est sous-mot de f si on peut obtenir g en supprimant certaines des lettres de f, soit formellement

$$g = g_1 g_2 \dots g_n \qquad f = f_1 g_1 f_2 g_2 \dots f_n g_n f_{n+1}$$

où les f_i et les g_i sont des mots. Ainsi aca est un sous-mot de $badcbda$

Question 1. Donner un automate déterministe qui reconnaît l'ensemble de tous les mots mots de A^* qui ont pour sous-mot $acab$

Question 2. Soit g et h deux mots de A^*, montrer que l'ensemble des mots f de A^* qui ont pour sous-mot g et qui n'ont pas h pour sous-mot forment un langage rationnel.

Question 3. Donner un automate qui reconnaît le langage défini à la question précédente en prenant $g = acab$ et $h = cabd$.

On dit que g est sous-mot complémentaire de h dans le mot f si g et h sont sous-mots de f et si en supprimant g de f on retrouve h. Soit de manière plus formelle :

$$g = g_1 g_2 \dots g_n \quad f = h_1 g_1 h_2 g_2 \dots h_n g_n h_{n+1} \quad h = h_1 h_2 \dots h_n h_{n+1}$$

On considère l'ensemble L des mots qui contiennent $g = dcab$ comme sous-mot.

Question 4. Construire un automate (avec sortie) qui à partir d'un mot f d'entrée, que l'on suppose appartenant à L donne comme sortie un sous-mot h qui soit complémentaire de $dcab$ dans f.

Question 5. Donner un programme `flex` qui réalise la même fonction que l'automate demandé à la question 4.

Problème 3 (7 points)

On expose iun algorithme qui permet de faire l'analyse syntaxique dans une grammaire quelconque.

On se donne une grammaire $\mathcal{G} = (A, \Sigma, \mathcal{R}, S)$ où les notations sont celles du cours, S est l'axiome. On appelle E-item un élément de la forme :

$$[X \to \alpha.\beta, i]$$

Où $X \to \alpha\beta$ est une règle de $\mathcal{R}$, où α et β sont des mots qui peuvent être vides et où i est un entier.

L'algorithme consiste, étant donné un mot f de longueur n engendré par la grammaire, à construire $n+1$ ensembles d'items notés $I_0, I_1, \dots, I_n$. On commence à construire I_0 puis itérativement I_j à partir de I_{j-1}, pour $j = 1$ à n. **Construction de I_0 :** On commence par le premier des trois pas décrits ci-dessous et on poursuit en appliquant les deux suivants jusqu'à ce que l'on ne puisse plus rien ajouter

- Pour toute règle $S \to \alpha$ ayant l'axiome S comme membre gauche membre gauche ajouter $[S \to .\alpha, 0]$ à I_0
- Si $[X \to \alpha.Y\beta, 0]$ appartient à I_0 alors ajouter $[Y \to .\gamma, 0]$ à I_0 pour toutes les règles de la forme $Y \to \gamma$.
- Si $[X \to \alpha., 0]$ et $[Y \to \alpha.X\beta, 0]$ appartiennent à I_0, ajouter $[Y \to \alpha X.\beta, 0]$ à I_0

Construction de I_j à partir des I_k, $k < j$: On commence par le premier des trois pas et on poursuit en appliquant les deux suivants jusqu'à ce que l'on ne puis plus rien ajouter. On suppose que le mot à analyser s'écrit $f = a_1 a_2 \dots a_i \dots a_n$

- Pour chaque $[X \to \alpha.a\beta, i]$ dans I_{j-1} tel que $a = a_j$ ajouter $[X \to \alpha a.\beta, i]$ dans I_j
- Si $[X \to \alpha., i]$ appartient à I_j examiner s'il existe des items de la forme $[Y \to \alpha.X\beta, k]$ dans I_i. Pour chacun de ces items trouvés ajouter $[Y \to \alpha X.\beta, k]$ dans I_j

- Pour chaque item de la forme $[X \to \alpha.Y\beta, k]$ appartenant à I_j ajouter $[Y \to .\gamma, j]$ à I_j pour toutes les règles de la forme $Y \to \gamma$.

Question 1. On considère la grammaire

$$\begin{array}{ll} E \to T + E & E \to T \\ T \to F * T & T \to F \\ F \to (E) & F \to a \end{array}$$

Attention ce n'est pas tout à fait la grammaire E, T, F habituelle! On donne le mot $(a + a) * a$, construire les 8 ensembles d'E-items $I_0, I_1, \ldots, I_7$ à l'aide de l'algorithme précédent.

Question 2. Donner un algorithme d'analyse syntaxique d'un mot engendré par la grammaire précédente basé sur la construction des E-items.

Question 3. Généraliser à une grammaire quelconque. Donner une évaluation intuitive de la complexité de l'algorithme correspondant.

Annexe: Grammaire Java utile:

Déclarations de constructeurs

```
ConstructorDeclaration:
 ConstructorDeclarator ConstructorBody

ConstructorDeclarator:
 SimpleName ( FormalParameterList_opt )

ConstructorBody:
 {BlockStatements_opt }
```

Types, valeurs, et variables

```
VariableDeclarators:
 VariableDeclarator
 VariableDeclarators , VariableDeclarator

VariableDeclarator:
 VariableDeclaratorId
 VariableDeclaratorId = VariableInitializer

VariableDeclaratorId:
 Identifier
 Variable

Type:
 PrimitiveType
 ReferenceType

PrimitiveType:
 NumericType
 boolean


ReferenceType:
 ClassOrInterfaceType
 ArrayType

ArrayType:
 PrimitiveType [ ]
 Name [ ]
 ArrayType [ ]
```

Noms

```
Name:
 SimpleName
 QualifiedName

SimpleName:
 Identifier

QualifiedName:
 Name . Identifier
```

Blocs et instructions

```
Block:
 {BlockStatements_opt }


BlockStatements:
 BlockStatement
 BlockStatements BlockStatement

BlockStatement:
 LocalVariableDeclarationStatement
 Statement

LocalVariableDeclarationStatement:
 LocalVariableDeclaration ;

LocalVariableDeclaration:
 Type VariableDeclarators

Statement:
 StatementWithoutTrailingSubstatement
 LabeledStatement
 BlockStatementsBlockStatementsIfThenStatement
 IfThenElseStatement
 WhileStatement
 ForStatement


StatementWithoutTrailingSubstatement:
 Block
 EmptyStatement
 ExpressionStatement
 SwitchStatement
 DoStatement
 BreakStatement
 ContinueStatement
 ReturnStatement
 SynchronizedStatement
 ThrowStatement
 TryStatement

EmptyStatement:
 ;

ExpressionStatement:
 StatementExpression ;

StatementExpression:
 Assignment

Assignment:
 LeftHandSide AssignOperator AssignExpression

LeftHandSide:
 Name
 FieldAccess
 ArrayAccess

AssignExpression:
 Name

AssignOperator: one of
 = *= /=
```