POLYTECHNIQUE MONTRÉAL

Questionnaire examen intra

LOG3210

Sigle du cours

Identification de l'étudiant(e)		
Nom : SOLUTIONNAIRE	**Prénom :**	
Signature :	**Matricule :**	**Groupe :**

Réservé

Sigle et titre du cours	*Groupe*	*Trimestre*
LOG3210 – Éléments de langages et compilateurs	**Tous**	**20161**
Professeur	***Local***	***Téléphone***
Ettore Merlo, responsable – Mathieu Mérineau, chargé de cours	**M-4105 M-4211**	**5758 4811**

Jour	*Date*	*Durée*	*Heures*
Mardi	**23 février 2016**	**2 h 00**	**8 h 30 à 10 h 30**

Documentation	*Calculatrice*	
☐ Aucune ☒ Toute ☐ Voir directives particulières	☒ Aucune ☐ Toutes ☐ Non programmable	**Les cellulaires, agendas électroniques ou téléavertisseurs sont interdits.**

Directives particulières

Important

Cet examen contient **4** questions sur un total de **11** pages **(excluant cette page)**

La pondération de cet examen est de **40** **%**

Vous devez répondre sur : ☒ le questionnaire ☐ le cahier ☐ les deux

Vous devez remettre le questionnaire : ☒ oui ☐ non

L'étudiant doit honorer l'engagement pris lors de la signature du code de conduite.

Question 1 : JavaCC (10 points)

Considérez la grammaire suivante :

$Prgm \rightarrow (\ Expr\ \textbf{SEMICOLON}\ )*$
$Expr \rightarrow FunctionCall \mid Identifier \mid \textbf{NUMBER} \mid \textbf{LPAREN}\ Expr\ \textbf{RPAREN}$
$FunctionCall \rightarrow\ Identifier\ \textbf{LPAREN ARG (COMMA ARG)*}\ \ \textbf{RPAREN}$
$Identifier \rightarrow \textbf{ID}$

Cette grammaire est implémentée dans JavaCC comme suit :

```
void Prgm() : { }
{
        ( Expr() <SEMICOLON> ) * <EOF>
}

void Expr() : { }
{
        LOOKAHEAD(k) FunctionCall() |
        Identifier() |
        <NUMBER> |
        <LPAREN> Expr() <RPAREN>
}

void FunctionCall() :
{
        // QUESTION 1d)
}
{
        Identifier()
        <LPAREN> <ARG> ( <COMMA> <ARG> )* <RPAREN>
}

String Identifier() :
{
        Token t;
        String s;
}
{
        t = <ID>
        {
         s = t.image;
         return s;
        }
}
```

où ***k*** est la valeur du lookahead.

a) Calculez FIRST(*Expr*). (1 point)

FIRST(*FunctionCall*) = FIRST(*Identifier*) = { <ID> }
FIRST(*FunctionCall*) ⊂ FIRST(*Expr*)
FIRST(*Identifier*) ⊂ FIRST(*Expr*)

FIRST(*Expr*) = { <ID>, <NUMBER>, <LPAREN> }

b) Cette grammaire est-elle LL(1), LL(2), LL(3) ou LR(1) ? Pourquoi ? Quelle est la valeur de ***k*** ? (2 points)

Cette grammaire est LL(2) et la valeur de k est 2.

Il faut distinguer les règles de production
Expr -> FunctionCall
Expr -> Identifier
qui chacune ont les mêmes FIRST.

c) Pourrait-on rendre cette grammaire LL(1) ? Comment ? (2 points)

Oui, il est nécessaire de factoriser à gauche la grammaire.

$Expr \rightarrow \mathbf{id}\ Expr' \mid \mathbf{NUMBER} \mid \mathbf{LPAREN}\ Expr\ \mathbf{RPAREN}$
$Expr' \rightarrow \mathbf{LPAREN\ ARG\ (COMMA\ ARG)^*\ \ RPAREN} \mid \epsilon$
(La grammaire factorisée n'était pas requise.)

d) Complétez par effets de bord la traduction dirigée par la syntaxe pour imprimer chacun des noms de fonctions appelées par le programme.

Par exemple, pour l'extrait suivant :

Entrée du programme	**Sortie attendue**

`foo(42);`
`x;`
`bar(x, y, z);`
	`foo`
	`bar`

Modifiez le code de la méthode `FunctionCall()`.

(5 points)

```
void FunctionCall() :
{
        String s;


}
{

        s = Identifier() { System.out.println(s); }
        <LPAREN> <ARG> ( <COMMA> <ARG> )* <RPAREN>




}
```

Question 2 : Analyseur syntaxique prédictif (10 points)

Considérez un extrait de la grammaire pour la logique temporelle linéaire (LTL) [1] :

$$Ltl \rightarrow Opd \mid (\ Ltl\ ) \mid Ltl\ Binop\ Ltl \mid Unop\ Ltl$$
$$Opd \rightarrow \mathbf{true} \mid \mathbf{false} \mid \mathbf{id}$$
$$Binop \rightarrow \mathbf{U} \mid \mathbf{and} \mid \mathbf{or}$$
$$Unop \rightarrow [] \mid <> \mid !$$

a) Cette grammaire est récursive à gauche. Éliminez la récursivité. (2 points)

$$Ltl \rightarrow Opd\ Ltl' \mid (\ Ltl\ )\ Ltl' \mid Unop\ Ltl\ Ltl'$$
$$Ltl' \rightarrow Binop\ Ltl\ Ltl' \mid \epsilon$$
$$Opd \rightarrow \mathbf{true} \mid \mathbf{false} \mid \mathbf{id}$$
$$Binop \rightarrow \mathbf{U} \mid \mathbf{and} \mid \mathbf{or}$$
$$Unop \rightarrow [] \mid <> \mid !$$

[1] http://spinroot.com/spin/Man/ltl.html

$Ltl \rightarrow Opd \mid (\ Ltl\ ) \mid Ltl\ Binop\ Ltl \mid Unop\ Ltl$
$Opd \rightarrow \mathbf{true} \mid \mathbf{false} \mid \mathbf{id}$
$Binop \rightarrow \mathbf{U} \mid \mathbf{and} \mid \mathbf{or}$
$Unop \rightarrow [] \mid <> \mid !$

b) Cette grammaire est ambiguë. Démontrez-le à l'aide de la chaîne :

id U []id and <>id (2 points)

Construire deux arbres de syntaxe, l'un avec une dérivation à gauche, l'autre avec une dérivation à droite.

c) Pour résoudre l'ambigüité de cette grammaire, nous utiliserons la règle d'élimination suivante : Toujours procéder à une dérivation à gauche. Cette règle d'élimination est compatible avec un parseur LL. Quelles sont deux autres solutions à l'ambigüité ?

(1 point)

- Modifier la grammaire.
- Modifier le langage.

Retranscrivez ici votre réponse de la question 2a).

$$Ltl \rightarrow Opd\ Ltl' \mid (\ Ltl\ )\ Ltl' \mid Unop\ Ltl\ Ltl'$$
$$Ltl' \rightarrow Binop\ Ltl\ Ltl' \mid \epsilon$$

$$Opd \rightarrow \mathbf{true} \mid \mathbf{false} \mid \mathbf{id}$$
$$Binop \rightarrow \mathbf{U} \mid \mathbf{and} \mid \mathbf{or}$$
$$Unop \rightarrow [] \mid <> \mid !$$

d) Écrivez le code de l'analyseur syntaxique prédictif avec la détection d'erreurs implémentant votre grammaire modifiée à la question 2a). Vous pouvez le faire dans le langage de programmation de votre choix.

Considérez que la variable *lookahead* contient le terminal courant dans l'entrée. La méthode *match* vous est fournie :

```
void match (terminal t) {
        if (lookahead == t) {
                lookahead = nextTerminal ();
        } else {
                error ();
        }
}
```

Considérez également que les méthodes *Opd*, *Binop* et *Unop* sont fournies.

Complétez également les FIRST et les FOLLOW nécessaires.

(Répondre aux pages suivantes.)

Étapes intermédiaires (1 point)

FIRST(*Ltl*) = { **true, false, id, (, [], <>, !** }

FOLLOW(*Ltl*) = **{ $,), U, and, or }**

FIRST(*Ltl'*) = **{ U, and, or, ϵ }**

FOLLOW(*Ltl'*) = **{ $,), U, and, or }**

(Implémentation à la page suivante.)

Implémentation (4 points)

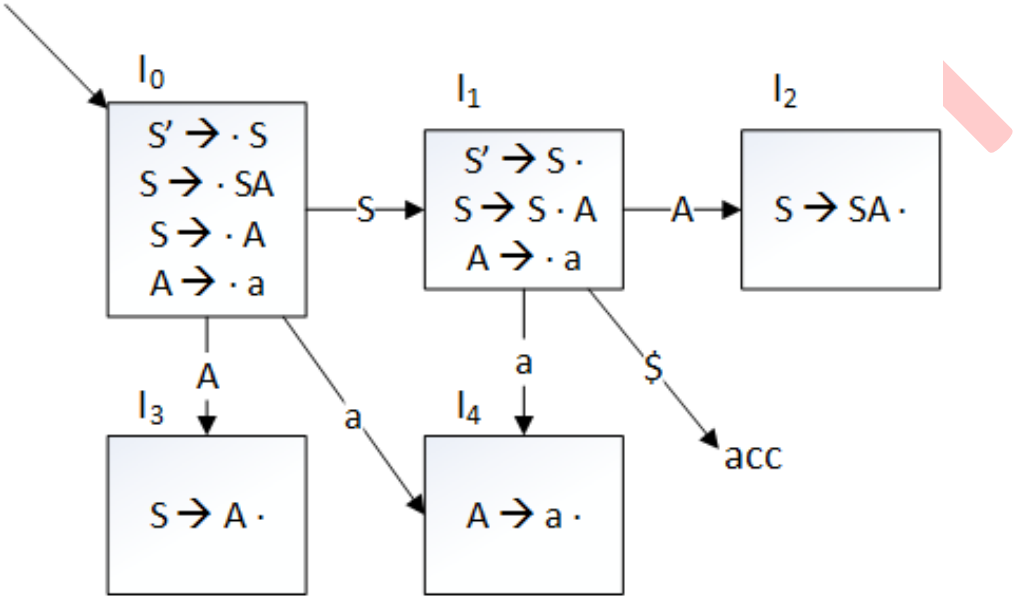
```
void Ltl () {

if ( lookahead == <true>
  || lookahead == <false>
  || lookahead == <id> ) {
          Opd(); Ltl_Prime();
}
else if ( lookahead == <(> ) {
          match( <(> );
          Ltl();
          match( <)> );
          Ltl_Prime();
}
else if ( lookahead == « [] »
       || lookahead == « <> »
       || lookahead == « ! » ) {
          Unop(); Ltl(); Ltl_Prime();
}
else
          error('Unexpected ' . lookahead);

}


void Ltl_Prime() {

if ( lookahead == <U>
  || lookahead == <and>
  || lookahead == <or> ) {
          Binop(); Ltl(); Ltl_Prime();
}
else if ( lookahead == <EOF>
       || lookahead == <)> ) {
          // Do nothing. Epsilon-production.
}
else
          error('Unexpected ' . lookahead);



// Implémentation avec un switch-case aussi possible.



}
```

Question 3 : Analyseur syntaxique ascendant (10 points)

Considérez la grammaire augmentée suivante :

1. $S' \rightarrow S$
2. $S \rightarrow SA$
3. $S \rightarrow A$
4. $A \rightarrow \mathbf{a}$

Pour laquelle nous avons construit l'automate LR(0) suivant :



FOLLOW(S) = { **a, $** } FOLLOW($A$) ={ **a, $** }

a) Complétez la table de parsage Simple LR (SLR). (5 points)
Respectez les conventions vues en classe (s*X*, r*Y*, acc, err).

	Action		Goto	
État	**a**	**$**	*S*	*A*
0	**s4**		**1**	**3**
1	**s4**	**acc**		**2**
2	**r2**	**r2**		
3	**r3**	**r3**		
4	**r4**	**r4**		

1. $S' \rightarrow S$
2. $S \rightarrow SA$
3. $S \rightarrow A$
4. $A \rightarrow \mathbf{a}$

b) Complétez la table des actions pour la chaîne :
aa (3 points)

Étape	Pile	Entrée	Action
(0)	\$ S_0	**aa**\$	s4
(1)	\$ S_0 S_4	**a**\$	r4
(2)	\$ S_0 S_3	**a**\$	r3
(3)	**\$ S_0 S_1**	**a\$**	**s4**
(4)	**\$ S_0 S_1 S_4**	**\$**	**r4**
(5)	**\$ S_0 S_1 S_2**	**\$**	**r2**
(6)	\$ S_0 S_1	\$	acc

c) Cette grammaire est-elle LR(1) ? Pourquoi ? (1 point)

Oui, cette grammaire est SLR puisqu'il est possible de construire une table de parsage SLR qui ne contient pas de conflits d'actions et SLR ⊂ LR.

d) Cette grammaire est-elle LL(1) ? Pourquoi ? (1 point)

Non, elle est récursive à gauche.

Question 4 : Traduction dirigée par la syntaxe (10 points)

Considérez la grammaire suivante :

$$Pgrm' \rightarrow Pgrm$$
$$Pgrm \rightarrow Ltl\ Pgrm \mid Ltl$$
$$Ltl \rightarrow Opd \mid (\ Ltl\ ) \mid Ltl\ \textbf{binop}\ Ltl \mid \textbf{unop}\ Ltl$$
$$Opd \rightarrow \textbf{true} \mid \textbf{false} \mid \textbf{id}$$

a) Complétez la définition dirigée par la syntaxe (SDD) qui calcule le nombre de caractères alphanumériques dans le programme (ignorez les caractères d'espacement) et affiche ce nombre une seule fois. Utilisez la fonction *size* pour obtenir la longueur d'un **lexval**. Aucun autre effet de bord (telles des variables globales) n'est permis. Utilisez le tableau suivant.

(9 points)

Règles de production	Actions sémantiques
$Pgrm' \rightarrow Pgrm$	*print*($Pgrm.val$)
$Pgrm \rightarrow Ltl\ Pgrm_1$	**Pgrm.val = Ltl.val + Pgrm$_1$.val**
$Pgrm \rightarrow Ltl$	**Pgrm.val = Ltl.val**
$Ltl \rightarrow Opd$	**Ltl.val = Opd.val**
$Ltl \rightarrow (\ Ltl_1\ )$	**Ltl.val = Ltl$_1$.val + 2** **(Ne pas additionnez 2 est acceptable comme les parenthèses ne sont pas des caractères alphanumériques.)**
$Ltl \rightarrow Ltl_1\ \textbf{binop}\ Ltl_2$	**Ltl.val = Ltl$_1$.val + size(binop.lexval) + Ltl$_2$.val**
$Ltl \rightarrow \textbf{unop}\ Ltl_1$	**Ltl.val = size(unop.lexval) + Ltl$_1$.val**
$Opd \rightarrow \textbf{true}$	**Opd.val = 4**
$Opd \rightarrow \textbf{false}$	**Opd.val = 5**
$Opd \rightarrow \textbf{id}$	**Opd.val = size(id.lexval)**

b) Quelle est la classe de votre SDD ? Pourquoi ? (1 point)

S-Attributed, car tous les attributs sont synthétisés.