École Polytechnique de Montréal
Département de génie informatique et logiciel

LOG3210 – Éléments de langages et compilateurs

Intra hiver 2014

Solutionnaire

Question 1 – Analyse lexicale (10 points)

Nous désirons analyser des documents contenant des numéros de téléphone et des adresses courriel.

Compléter les instructions JavaCC suivantes afin de permettre à l'analyseur lexical de produire des jetons qui représenteront des numéros de téléphone (ex : (514) 576-0864) et des adresses courriel (ex : bobby.beaulieu@polymtl.ca).

Remplacez les sections dénotées par : « /* À COMPLÉTER */ »

```
TOKEN :
{
   < TELEPHONE :  "(" (["0"-"9"]){3}  ")" (["0"-"9"]){3} "-" (["0"-"9"]){4}>
}

TOKEN :
{
   < COURRIEL :  (["A"-"Z","a"-"z"])+ "." (["A"-"Z","a"-"z"])+ "@" (["A"-"Z","a"-"z"])+ "."
                 (["A"-"Z","a"-"z"])+ >
}
```

Question 2 – Construction d'un parseur descendant récursif (30 points)

Vous désirez transmettre des données d'annuaire entre différentes applications. Pour ce faire, vous avez décidé d'utiliser le format XML. Spécifiquement, vous désirez partager les informations suivantes sur une personne : prénom, nom, un ou plusieurs numéros de téléphone et optionnellement un courriel. Le code ci-dessous illustre comment les informations sont encodées :

```
<Annuaire>
  <Personne>
    <prenom>Bobby</prenom>
    <nom>Beaulieu</nom>
    <telephone type="maison">(514) 456-8764</telephone>
    <telephone type="travail">(514) 565-4365</telephone>
    <courriel>bobby.beaulieu@polymtl.ca</courriel>
  </Personne>
</Annuaire>
```

La grammaire ci-dessous définit le dialecte XML qui est utilisé :

Annuaire	→	**<annuaire>** *Personnes* **</annuaire>**
Personnes	→	*Personnes Personne* | *Personne*
Personne	→	**<personne>** *Prenom Nom Telephones Personne'*
Personne'	→	*Courriel* **</personne> | </personne>**
Prenom	→	**<prenom> string </prenom>**
Nom	→	**<nom> string </nom>**
Telephones	→	*Telephones Telephone* | *Telephone*
Telephone	→	**<telephone** *Type* **> num </telephone>**
Type	→	**type = " string "**
Courriel	→	**<courriel> email </courriel>**

Le code partiel du parseur descendant récursif prédictif (LL(1)) implantant cette grammaire vous est fourni. Complétez les sections manquantes, indiquées par le commentaire « À COMPLÉTER ». Vous pouvez modifier la grammaire et ajoutez de nouvelles méthodes si nécessaire.

À titre indicatif, le terminal **string** représente une chaîne de caractères, le terminal **num** représente un numéro de téléphone (ex : (514) 653-9753) et le terminal **email** représente un courriel (ex : prenom.nom@polymtl.ca).

NOTE : La grammaire et le code ont été reproduits **annexes détachables**.

2.1 : Complétez la méthode Personnes() : (10 points)

La règle de production associée à la méthode Personnes() est la suivante :

Personnes → *Personnes Personne* | *Personne*

Question 2.1.1 : Est-il nécessaire de modifier la règle de production associée à la méthode Personnes()? Si oui, écrivez les nouvelles règles et justifiez votre choix.

Réponse 2.1.1 :

Oui, car la règle est récursive à gauche. Elle devient :

Personnes → *Personne Personnes'*
Personnes' → *Personne Personnes'* | ε

Question 2.1.2 : Écrivez le code associé à la méthode Personnes(). Si vous avez modifié la règle de production de *Personnes* à la question 2.1.1, écrivez le code correspondant aux nouvelles règles de production.

Étant donné la règle modifiée :

Personnes → *Personne Personnes'*
Personnes' → *Personne Personnes'* | ε

Le code est :

```
public void Personnes() {

	if(lookahead.toString().equals("<personne>")) {
		Personne(); Personnes_prime();
	}
	else {
		error();
	}
}

public void Personnes_prime() {

	if(lookahead.toString().equals("<personne>")) {
		Personne(); Personnes_prime();
	}
	else if(lookahead.toString().equals("</annuaire>"){
	//Production epsilon
	}
	else {
		error();
	}
}
```

2.2 : Complétez la méthode Personne_prime() : (10 points)

La règle de production associée à la méthode Personne_prime() est la suivante :

Personne' → *Courriel* **</personne> | </personne>**

Question 2.2.1 : Est-il nécessaire de modifier la règle de production associée à méthode Personne_prime()? Si oui, écrivez les nouvelles règles et justifiez votre choix.

Réponse 2.2.1 :

Il n'est pas nécessaire de modifier la règle.

Question 2.2.2 : Écrivez le code associé à la méthode Personne_prime(). Si vous avez modifié la règle de production de *Personne'* à la question 2.2.1, écrivez le code correspondant aux nouvelles règles de production.

Réponse 2.2.2

Étant donné la règle :

Personne' → *Courriel* **</personne> | </personne>**

Le code est :

```
public void Personne_prime() {

	if(lookahead.toString().equals("<courriel>")) {
		Courriel(); match("</personne>");
	}
	else if(lookahead.toString().equals("</personne>")) {
		match("</personne>");
	}
	else {
		error();
	}
}
```

2.3 : Complétez la méthode Telephones() : (10 points)

La règle de production associée à la méthode Telephones() est la suivante :

Telephones → *Telephones Telephone* | *Telephone*

Question 2.3.1 : Est-il nécessaire de modifier la règle de production associée à méthode Telephone()? Si oui, écrivez les nouvelles règles et justifiez votre choix.

Réponse 2.3.1 :

Oui, car la règle est récursive à gauche. Elle devient :

Telephones → *Telephone Telephones'*
Telephones' → *Telephone Telephones'* | ε

Question 2.3.2 : Écrivez le code associé à la méthode Telephones(). Si vous avez modifié la règle de production de *Telephones* à la question 2.3.1, écrivez le code correspondant aux nouvelles règles de production.

Réponse 2.3.2

Étant donné la règle modifiée :

Telephones → *Telephone Telephones'*
Telephones' → *Telephone Telephones'* | ε

Le code est :

```
public void Telephones() {

      if(lookahead.toString().equals("<telephone")) {
            Telephone(); Telephones_prime();
      }
      else {
            error();
      }
}

public void Telephones_prime() {

      if(lookahead.toString().equals("<telephone")) {
            Telephone(); Telephones_prime();
      }
      else if(lookahead.toString().equals("<courriel>") ||
              lookahead.toString().equals("</personne>")) {
            //Production epsilon
      }
      else {
            error();
      }
}
```

Question 2.4 (5 points) :

La règle de production originale associée au non terminal *Personne* était :

Personne → **<personne>** *Prenom Nom Telephones* **</personne> |**
<personne> *Prenom Nom Telephones Courriel* **</personne>**

elle est devenue :

Personne → **<personne>** *Prenom Nom Telephones Personne'*
Personne' → *Courriel* **</personne> | </personne>**

Nommez la transformation qui a été effectuée et expliquez son utilité.

Réponse 2.4 :

Il y a eu une factorisation à gauche, ce qui permet au parseur de limiter son *lookahead* à 1 symbole.

Question 3 – Parseur LR (30 points)

Considérez la grammaire augmentée et numérotée suivante :

1. *S'* → S
2. *S* → *A* **a** *A* **b**
3. *S* → *B* **b** *B* **a**
4. *A* → **c**
5. *B* → **c**

a) Remplissez la table de parsage SLR ci-dessous. (25 points)
b) Cette grammaire est-elle SLR? Si oui, pourquoi? Sinon, pourquoi? (5 points)

État	Action				Goto		
	a	**b**	**c**	**$**	S	A	B
0			s4		1	2	3
1				acc			
2	s5						
3		s6					
4	r4 r5	r4 r5					
5			s8			7	
6			s10				9
7		s11					
8	r4	r4					
9	s12						
10	r5	r5					
11				r2			
12				r3			

Réponse question 3

a) Pour remplir la table, nous devons construire les ensembles d'items de la grammaire.

I_0 = S' → ·S
S → ·AaAb
S → ·BbBa
A → ·c
B → ·c

I_1 = S' → S·

I_2 = S → A·aAb

I_3 = S → B·bBa

I_4 = A → c·
B → c·

I_5 = S → Aa·Ab
A → ·c

I_6 = S → Bb·Ba
B → ·c

I_7 = S → AaA·b

I_8 = A → c·

I_9 = S → BbB·a

I_{10} = B → c·

I_{11} = S → AaAb·

I_{12} = S → BbBa·

b) Cette grammaire n'est pas SLR car il y a un conflit de réduction à l'état 4

Question 4 – Traduction dirigée par la syntaxe (20 points)

Vous devez implémenter une traduction dirigée par la syntaxe qui permettra de lire un annuaire en format XML et de créer un objet Annuaire Java. Ci-dessous, vous trouverez le code Java correspondant à un annuaire et à une personne.

```
public class Annuaire {

	private List<Personne> p = new LinkedList<Personne>();

	public void add(Personne personne) {
		p.add(personne);
	}

}

public class Personne {

	private String prenom = null;
	private String nom = null;
	private String courriel = null;
	private HashMap<String,String> telephones =
					new HashMap<String,String>();

	public setPrenom(String prenom) {
		this.prenom = prenom;
	}

	public setNom(String nom) {
		this.nom = nom;
	}

	public addTelephone(String type, String numero) {
		telephones.put(type, numero);
	}

	public setCourriel(String courriel) {
		this.courriel = courriel;
	}

}
```

Écrivez une traduction dirigée par la syntaxe (SDT) qui permettra de construire un annuaire JAVA à partir d'un annuaire XML. Vous devez instancier un seul objet de type annuaire et y ajouter autant d'objets de type personne qu'il y a de personnes dans l'annuaire XML. L'ajout de personnes dans l'annuaire s'effectue à l'aide de la fonction *add* de la classe Annuaire. Finalement, pour chaque objet de type personne, utilisez les méthodes *setPrenom* et *setNom* pour fixer les prénoms et noms, ajoutez les numéros de téléphone(s) avec la méthode *addTelephone* et le courriel avec la méthode *setCourriel*.

La SDT produite doit être **attribuée à gauche** (*L-attributed*).

En termes d'effets de bord, vous pouvez instancier des objets de type Annuaire et de type Personne et faire appel à leur méthodes publiques.

Vous pouvez considérer que **string**.lexval, **num**.lexval et **email**.lexval retournent le contenu des terminaux **string, num** et **email** respectivement.

Répondez directement **dans les cases du tableau**. Les règles de la grammaire XML de l'annuaire y ont été reproduites.

Réponse question 4 :

Règle de production	Actions sémantiques
A → **<Annuaire>** *Ps* **</annuaire>**	A.annuaire = new Annuaire() Ps.annuaire = A.annuaire
Ps → Ps_1 *P*	Ps_1.annuaire = Ps.annuaire P.annuaire = Ps.annuaire
Ps → P	P.annuaire = Ps.annuaire
P → **<personne>** *Pr N Ts P'*	P.personne = new Personne() P.annuaire.add(P.personne) P.personne.setPrenom(Pr.val) P.personne.setNom(N.val) Ts.personne = P.personne P'.personne = P.personne
P' → *C* **</personne>**	P'.personne.setCourriel(C.val)

P' → **</personne>**	
Pr → **<prenom> string </prenom>**	Pr.val = **string**.lexval
N → **<nom> string </nom>**	N.val = **string**.lexval
Ts → *Ts_1 T*	Ts_1.personne = Ts.personne T.personne = Ts.personne
Ts → *T*	T.personne = Ts.personne
T → **<telephone** *Type***> num </telephone>**	T.personne.addTelephone(Type.val, **num**.lexval)
Type → **type = " string "**	Type.val = **string**.lexval
C → **<courriel> email </courriel>**	C.val = **email**.lexval

Annexe 1 : Grammaire du bottin XML à détacher

Annuaire	→	**<annuaire>** *Personnes* **</annuaire>**
Personnes	→	*Personnes Personne* | *Personne*
Personne	→	**<personne>** *Prenom Nom Telephones Personne'*
Personne'	→	*Courriel* **</personne> | </personne>**
Prenom	→	**<prenom> string </prenom>**
Nom	→	**<nom> string </nom>**
Telephones	→	*Telephones Telephone* | *Telephone*
Telephone	→	**<telephone** *Type* **> num </telephone>**
Type	→	**type = " string "**
Courriel	→	**<courriel> email </courriel>**

Annexe 1 : Code du parseur XML à détacher

```
public class Parser {

	private TokenStream ts;
	private Token lookahead;

	public Parser(TokenStream ts) {
		this.ts = ts;
		lookahead = ts.first();
	}

	public void match (String t) {
		if(lookahead.toString() == t)
			lookahead = ts.next();
		else
			error();
	}

	public void Annuaire() {
		match(“<annuaire>”); Personnes(); match(“</annuaire>”);
	}

	public void Personnes() {
		//À COMPLÉTER (Question 2.1)
	}

	public void Personne() {
		match(“<personne>”);
		Prenom(); Nom(); Telephones(); Personne_prime();
	}

	public void Personne_prime() {
		// À COMPLÉTER (Question 2.2)
	}

	public void Prenom() {
		match(“<prenom>”);
		match(“string”);
		match(“</prenom>”);
	}

	public void Nom() {
		match(“<nom>”);
		match(“string”);
		match(“</nom>”);
	}
```

```
	public void Telephones() {
		// À COMPLÉTER (Question 2.3)
	}

	public void Telephone() {
		match(“<telephone”); Type(); match(“>”);
		match(“num”);
		match(“</telephone>”);
	}

	public void Type() {
		match(“type”); match(“=”);
		match(“"”); match(“string”); match(“"”);
	}

	public void Courriel() {
		match(“<courriel>”);
		match(“email”);
		match(“</courriel>”);
	}
}
```