École Polytechnique de Montréal
Département de génie informatique et logiciel

LOG3210 – Éléments de langages et compilateurs

Intra hiver 2013

Solutionnaire

Question 1 – Grammaires (10 points)

Soit la grammaire d'expressions booléennes :

bexpr → *bexpr* **or** *bterm* | *bterm*
bterm → *bterm* **and** *bfactor* | *bfactor*
bfactor → **not** *bfactor* | **(** *bexpr* **)** | **true** | **false**

a) Quels sont les symboles terminaux?
b) Quels sont les symboles non-terminaux?
c) Quel est le symbole de départ (symbole START)?
d) Combien y a-t-il de règles de production?
e) Cette grammaire génère-t-elle des erreurs lorsque compilée avec JavaCC? Si oui, expliquez pourquoi. Si non, expliquez pourquoi.

Solution :

a) **or and not ( ) true false**
b) *bexpr, bterm, bfactor*
c) *bexpr*
d) 8
e) Non, elle n'est pas compatible à cause de la récursivité à gauche aux deux premières règles. À titre indicatif, voici la grammaire modifiée pour éliminer la récursivité à gauche :

bexpr → *bterm bexpr'*
bexpr' → **or** *bterm bexpr'* | ε
bterm → *bfactor bterm'*
bterm' → **and** *bfactor bterm'* | ε
bfactor → **not** *bfactor* | **(** *bexpr* **)** | **true** | **false**

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

La grammaire suivante décrit une partie du langage JSON (JavaScript Object Notation), souvent utilisé dans les applications Web pour l'échange de données :

object	→	**{** *members* **}** | **{ }**
members	→	*pair* **,** *members* | *pair*
pair	→	**string :** *value*
value	→	*object* | **string** | **num**

a) Écrivez le code du parseur descendant récursif prédictif (parseur LL(1)) implantant cette grammaire. Vous pouvez coder dans le langage de votre choix.

NOTE : La grammaire a été reproduite **en annexe si vous désirez la détacher**.

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

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

À titre indicatif, le terminal **string** représente des chaînes de caractères délimitées par des guillemets (ex : "bonjour") et le terminal **num** représente des nombres (ex : 123 ou encore 35.78).

Voici un exemple d'entrée acceptée par cette grammaire :

```
{
  "item" : "Pain",
  "prix" : 4.78,
  "stocks" : { "entrepot" : 10000, "magasin": 50 }
}
```

Solution :

La grammaire comporte des règles qui ont des préfixes communs. Il faut donc procéder à une factorisation à gauche. La grammaire résultante est la suivante :

object	→	**{** *object'*
object'	→	*members* **}** | **}**
members	→	*pair members'*
members'	→	**,** *members* | ε
pair	→	**string :** *value*
value	→	*object* | **string** | **num**

Afin de construire le parseur prédictif, nous devons calculer les *first* de chaque non-terminal :

a) First(*object*) = { **{** }
b) First(*members*) = { **string** }
c) First(*pair*) = { **string** }
d) First(*value*) = { **{**, **string**, **num** }

Le code du parseur descendant récursif prédictif est le suivant :

```
void object() {
      match({); object_prime();
}

void object_prime() {
      switch(lookahead){
            case string:
                  members();
                  match(});
                  break;
            case }:
                  match(});
                  break;
            default:
                  error();
      }
}

void members() {
      pair(); members_prime();
}
```

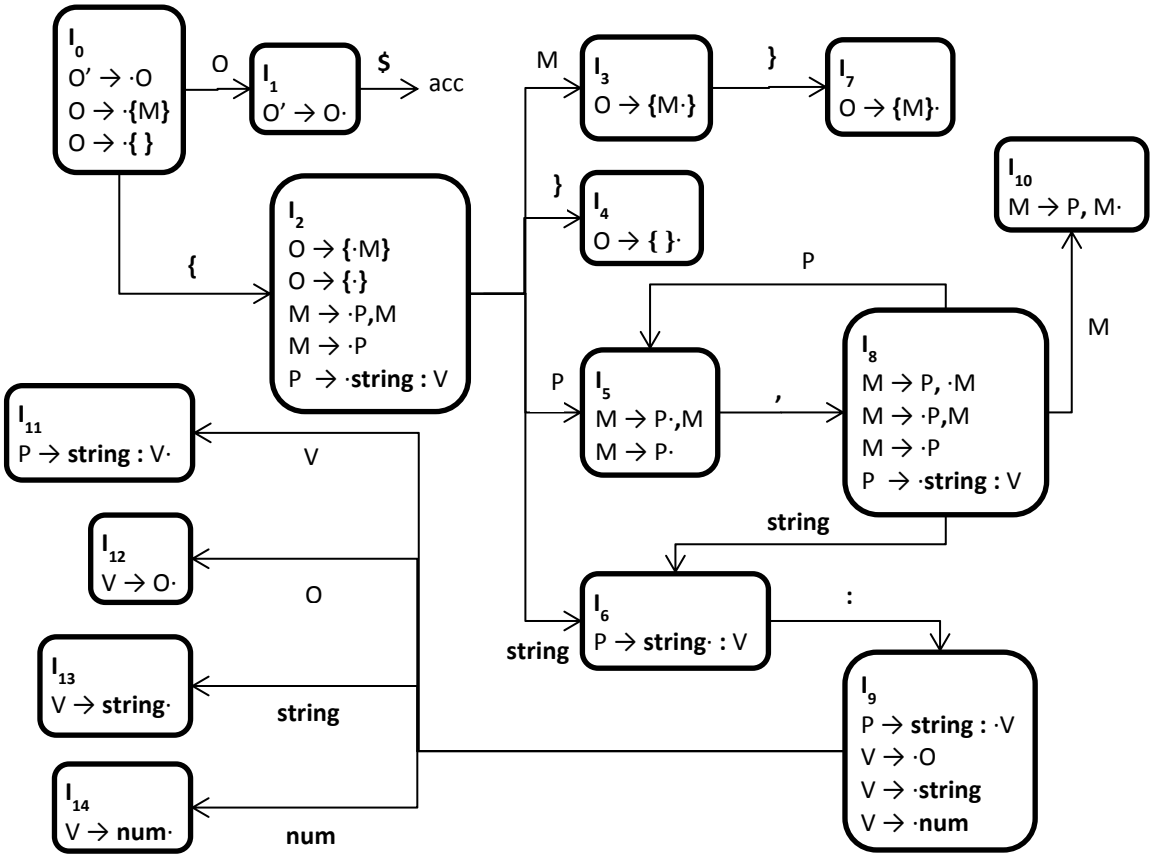
```
void members_prime() {
      if(lookahead == ,)
            match(,); members();
}

void pair() {
      match(string); match(:); value();
}

void value() {
      switch(lookahead){
            case {:
                  object();
                  break;
            case string:
                  match(string);
                  break;
            case num:
                  match(num);
                  break;
            default:
                  error();
      }
}
```

Question 3 – Construction d'un parseur ascendant (30 points)

La grammaire JSON de la question précédente a été reprise ici. Les non-terminaux ont été abrégés, la grammaire a été augmentée et les règles ont été numérotées :

0. O' → O
1. O → **{** M **}**
2. O → **{ }**
3. M → P **,** M
4. M → P
5. P → **string :** V
6. V → O
7. V → **string**
8. V → **num**

La figure qui suit, reproduite **en annexe si vous désirez la détacher**, représente l'automate *Simple LR* (SLR) de cette grammaire, construit à partir de l'ensemble d'items LR(0) :



a) Étant donnés la grammaire augmentée et numérotée et son automate SLR, remplissez la table de parsage SLR suivante :

État	Action							Goto			
	{	**}**	**,**	**string**	**:**	**num**	$	O	M	P	V
0	s2							1			
1							acc				
2		s4		s6					3	5	
3		s7									
4		r2	r2				r2				
5		r4	s8								
6					s9						
7		r1	r1				r1				
8				s6					10	5	
9				s13		s14		12			11
10		r3									
11		r5	r5								
12		r6	r6								
13		r7	r7								
14		r8	r8								

Respectez les conventions du livre du dragon :

1. sX (*shift X*) signifie décaler l'entrée et empiler l'état X.
2. rY (*reduce Y*) signifie réduire par la règle de production Y.
3. acc (*accept*) signifie que la chaîne d'entrée est valide; le parseur arrête.
4. Une case vide signale une erreur; le parseur arrête.

Solution :

Pour calculer les *reduce*, il faut déterminer les *follow* de chaque non-terminal :

a) Follow(O) = { **$ } , }**
b) Follow(M) = { **}** }
c) Follow(P) = { **} ,** }
d) Follow(V) = { **} ,** }

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

En vous basant sur la grammaire JSON simplifiée de la question 3, écrivez une traduction dirigée par la syntaxe (SDT) qui permettra de reconstruire en mémoire l'objet représenté par une chaîne de caractères de type JSON. Les *object*, dénotés par le non-terminal O, seront représentés par des tables de hachage et les *pair*, dénotées par le non-terminal P, formeront les paires de clé et valeur à ajouter dans la table de hachage. Finalement, la SDT produite doit être **attribuée à gauche** (*L-attributed*). Répondez directement dans les cases du tableau.

En termes d'effets de bord, vous pouvez seulement faire appel aux fonctions suivantes :

1. new HashMap() : Crée et retourne une référence vers une table de hachage.
2. put(*table, clé, valeur*) : Ajoute une paire (*clé, valeur*) dans la table *table*.

Règle de production	Actions sémantiques
O → **{** M **}**	O.map = new HashMap(); M.map = O.map;
O → **{ }**	O.map = new HashMap();
M → P **,** M_1	P.map = M.map; M_1.map = M.map;
M → P	P.map = M.map;
P → **string :** V	put(P.map, **string**.lexval, V.val)
V → O	V.val = O.map
V → **string**	V.val = **string**.lexval
V → **num**	V.val = **num**.lexval

Question 5 – Création d'une grammaire (10 points)

Écrivez une grammaire qui permette de reconnaître l'ensemble de **toutes** les chaînes de caractères, de **longueur** ≥ **1**, composées uniquement de 0 et de 1, et qui sont des *palindromes*; c'est-à-dire que la chaîne de caractères se lit de la même manière de gauche à droite ou de droite à gauche.

Solution :

S → **0** S **0** | **1** S **1** | **00** | **11** | **0** | **1**

Annexe 1 : Grammaire JSON détachable

object	→	**{** *members* **}** | **{ }**
members	→	*pair* **,** *members* | *pair*
pair	→	**string :** *value*
value	→	*object* | **string** | **num**

Annexe 2 : Grammaire JSON augmentée et automate SLR détachables

0. O' → O
1. O → **{** M **}**
2. O → **{ }**
3. M → P **,** M
4. M → P
5. P → **string :** V
6. V → O
7. V → **string**
8. V → **num**