UFR IEEA

Examen de compilation

Licence info, S5
1ère session COMPIL -3h- janvier 2008

FIL

Tous documents autorisés. Sujet : 3 pages

Exercice 1 : Déclarations Java et analyse syntaxique ascendante

On s'intéresse à la grammaire G_J d'une déclaration Java d'un identificateur de type entier ou tableau d'entier. On a $G_J = \{V_T, V_N, decl, P\}$ avec $V_T = \{$ `int`, `[]`, `id` $\}$, $V_N = \{$ *decl*, *cro* $\}$ et P contient les productions :

$$decl \rightarrow \texttt{int id} \mid \texttt{int id}\ cro$$
$$cro \rightarrow \texttt{[]}\ cro \mid \texttt{[]}$$

NB : les crochets ouvrant et fermant sont regroupés dans le seul symbole `[]`.

Q 1.1 : Donner l'automate LR-AFD de G_J. Préciser l'état initial et l'état final. □

Q 1.2 : G_J est-elle LR(0) ? Est-elle SLR(1) ? Justifier. □

Q 1. 3 : Donner pour G_J la suite des piles d'états issue de la reconnaissance du mot `int id []` par un analyseur SLR(1) (indiquer comment les conflits ont été arbitrés le cas échéant). □

Exercice 2 : Itinéraire et analyse syntaxique descendante

On s'intéresse à une grammaire G_I des instructions (très simplifiées) délivrées par un logiciel de calcul d'itinéraire, du style «avancer_100m, au_panneau_Lille tourner_à_gauche, avancer_20m, tourner_à_droite». On utilise pour ce faire les symboles `GO` (avancer_Xm), `TL` (turn left), `TR` (turn right) et `PAN` (panneau).

On a $G_I = \{V_T, V_N, route, P\}$ avec $V_T = \{$ `GO`, `TL`, `TR`, `PAN` $\}$, $V_N = \{$ *route*, *inst*, *suite*, *panneau*, *turn* $\}$ et P contient les productions :

$$route \rightarrow inst\ suite$$
$$suite \rightarrow \epsilon \mid inst\ suite$$
$$inst \rightarrow \texttt{GO} \mid panneau\ turn$$
$$turn \rightarrow \texttt{TL} \mid \texttt{TR}$$
$$panneau \rightarrow \epsilon \mid \texttt{PAN}$$

Q 2.1 : Calculer les ensembles *Premier* et *Suivant* pour G_I. □

Q 2.2 : Donner la table d'analyse LL(1) de G_I. Justifier en utilisant cette table que G_I est une grammaire LL(1). Vous pouvez si vous le souhaitez numéroter sur votre copie les productions de la grammaire et remplir la table avec ces numéros. □

On souhaite coder un analyseur syntaxique récursif descendant pour G_I. On suppose donné un type Java `TypeSymboles` définissant comme des constantes statiques les symboles terminaux de la grammaire `GO`, `TL`, `TR`, `PAN`, plus le marqueur de fin de fichier `EOF`. On s'autorisera à abbrévier `TypeSymboles` en `TS`, on écrira par exemple `TS.GO`. On utilisera la méthode `public void consommer(TypeSymboles t) throws ScannerException` et l'attribut `courant` de type `TypeSymboles` vus en cours.

Q 2. 3 : Donner les méthodes (signature et corps) de l'analyseur récursif descendant qui reconnaissent respectivement le non-terminal *suite* et le non-terminal *turn*, avec levée d'une exception `ParserException` en cas d'erreur syntaxique. □

Q 2. 4 : G_I est-elle une grammaire ambiguë ? Le langage $L(G_I)$ est-il algébrique ? régulier ? Justifier vos réponses. □

Exercice 3 : Circuits électriques

Il est conseillé de bien lire le préambule de l'exercice et les questions avant de répondre aux questions.

On s'intéresse dans cet exercice à la spécification de composants électriques, contenant (pour simplifier) uniquement des résistances.

Le composant de base est la *résistance* (graphiquement représenté sur la figure 1 par un rectangle). Une résistance possède une *valeur de résistance électrique* exprimée en ohm (Ω). Un composant peut aussi être formé de deux sous-composants assemblés *en série*, ou de deux sous-composants assemblés *en parallèle*. La valeur de résistance électrique r d'un composant s'exprime récursivement en fonction de celle de ses sous-composants. Si le circuit est composé :

- d'une résistance de valeur de résistance r_1 : $r = r_1$;
- des deux sous-composants C_1 (de valeur de résistance r_1) et C_2 (de valeur de résistance r_2) :
  - si C_1 et C_2 sont assemblés en série : $r = r_1 + r_2$;
  - si C_1 et C_2 sont assemblés en parallèle : $r = r_1 r_2/(r_1 + r_2)$.

Ex : sur le composant de la figure 1, les deux résistances de gauche sont assemblées en série mais les deux résistances de droite sont assemblées en parallèle. La résistance électrique globale du composant est donc $10 + 22 + 10 * 33/(10 + 33)$.

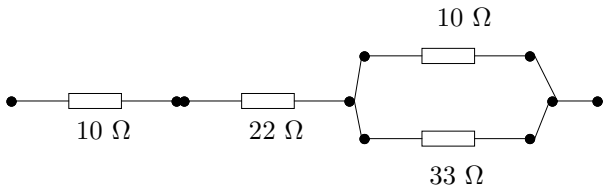

FIG. 1 – Un assemblage de résistances

Une spécification décrit un composant en désignant chaque résistance par un *identificateur* (suite de lettre et de chiffre commençant par une lettre). Elle est organisée en deux sections séparées par le mot-clé `circuit`. La première section déclare des identificateurs de résistance (qui doivent être *tous différents*) en leur associant une valeur de résistance électrique en ohm (un entier strictement positif et non signé). La seconde section décrit les connexions entre composants grâce aux mot-clés `ser` (assemblage en série) et `par` (assemblage en parallèle). Toute résistance *utilisée* dans cette seconde section doit avoir été *déclarée* dans la première section. La *mise en série* de deux composants est *prioritaire* sur leur *mise en parallèle*. Le calcul de résistance se fait de la gauche vers la droite : on considère que les opérateurs de mise en série et mise en parallèle sont *associatifs à gauche*.

Le composant de la figure 1 sera par exemple décrit par la spécification suivante :

```
R1 10
R2 22
R3 33
circuit
R1 ser R2 ser (R1 par R3)
```

La syntaxe du langage de description de circuits est donnée par la grammaire $G = (V_T, V_N, spec, P)$ où V_T = { `id`, `ser`, `par`, `po`, `pf`, `sep`, `entier` }, $V_N = \{spec, descr, C\}$ et P contient les productions suivantes :

$$spec \rightarrow descr \texttt{ sep } C$$
$$descr \rightarrow \texttt{id entier} \mid descr \texttt{ id entier}$$
$$C \rightarrow \texttt{id} \mid C \texttt{ par } C \mid C \texttt{ ser } C \mid \texttt{po } C \texttt{ pf}$$

Q 3.1 : Donner pour chaque terminal de G une description régulière qui le décrit en tant qu'unité lexicale (vous pouvez si vous le souhaitez utiliser la syntaxe de JFLEX). □

Q 3.2 : Montrer que G est ambiguë (justifier rigoureusement). □

Q 3.3 : En tenant compte de l'associativité et de la priorité des opérateurs données plus haut, donner une grammaire G' non ambiguë équivalente à G. □

Q 3.4 : Quelles vérifications sémantiques doivent être faites pour assurer que la spécification est correcte ? □

On souhaite attribuer G' pour :

1. vérifier que la spécification est sémantiquement correcte (levée d'une exception de type `SemException` sinon) ;
2. calculer la valeur de résistance globale du composant.

Pour ce faire, on **suppose donnée** une classe `Table` (analogue au type Java `Map`) permettant de mémoriser les couples ⟨identificateur,valeur⟩ qui décrivent les résistances. Son code **n'est pas à compléter** :

```
public class Table {
  public Table() {...}
  public void put(String id, int val) {...};
  public boolean contains(String id);
  public int get(String id);
}
```

Q 3.5 : Attribuer G' comme demandé. Préciser pour chaque attribut utilisé à quel terminal ou non-terminal il est associé, son type, s'il est fixé par l'analyseur lexical, et s'il est synthétisé ou hérité. Si vous n'avez pas donné de grammaire G', vous pouvez répondre à cette question en utilisant G. □

Q 3.6 : L'outil CUP permet d'analyser des grammaires ambiguës grâce à des précisions dans la spécification de l'analyseur. Que faut-il ajouter à la spécification CUP de la grammaire G pour obtenir une grammaire équivalente à G' ? □