Polytech'Paris-Sacaly Informatique Compilation 2020-2021

Grammaires et Analyse SLR

Exercice 1 : Soit la grammaire ambiguë suivante dont le langage associé est celui des expressions rationnelles sur un alphabet donné. Un élément de l'alphabet est désignée par le token `L`. Le token ∅ est la notation pour l'expression rationnelle vide (**ci-dessous, il s'agit donc d'un token, comme `'+'` ou `L`**). On note l'union par `'+'`, la concaténation par juxtaposition, l'itération un nombre quelconque de fois par `'*'`, l'itération un nombre fixé de fois en faisant suivre l'expression par une constante entière.

```
E::= E + E | E E | E * | E Cste | L | ( E ) | ∅
```

L'union est moins prioritaire que la concaténation qui est elle-même moins prioritaire que les deux itérations qui ont même priorité. L'union et la concaténation sont associatives à gauche.

1. Combien existe-t-il d'arbres syntaxiques possibles pour le mot : `L + L L Cste` ? Dessinez celui qui correspond aux priorités énoncées.
2. Donnez une grammaire **équivalente non ambiguë** avec les priorités et associativités demandées.

Calculez *First* et *Follow* pour la **grammaire de départ** augmentée de la règle S `::= E $`.

Exercice 2:. Calculez les ensembles *First* et *Follow*, l'automate LR(0) et montrez que le grammaire ci-dessous est SLR(1)

```
G1:  S' ::= S $
     S ::= ( S ) S |ε
```

Donnez les étapes de l'analyse syntaxique SLR(1) des mots ci-dessous (seul le premier mot est correct !)

```
()()$
())$
```

Exercice 3. La grammaire suivante augmentée de `S' ::= S $` est-elle SLR ?

```
G2 :  S ::= a | ( T )
      T ::= S T |ε
```

Exercice 4 :Soit la grammaire suivante qui définit des expressions arithmétiques et logiques (la grammaire a été simplifiée pour les besoins de l'exercice) :

```
E ::= E + E
E ::= E =E
E ::= E and E
E ::= E or E
E ::= not E
E ::= Id
```

Les identificateurs sont supposés représenter des valeurs de type `integer` ou `boolean`.

1. On suppose que **tous les opérateurs binaires** sont **associatifs à gauche**. Les opérateurs `and` et `or` ont la plus faible précédence (identique), suivis de `=`, de `+` et enfin `not`, qui a donc la plus forte précédence. Donnez une grammaire non ambiguë qui engendre le même langage que la grammaire initiale en respectant ces indications.

2. Donnez l'arbre syntaxique pour `not a + b = c and d`, en supposant que `a, b, c` et `d` sont des instances d'identificateurs.

La grammaire donnée en 3 est (probablement) non-ambiguë mais inutilement détaillée. Le concepteur considère alors la grammaire suivante :

```
E ::= E + E
E ::= E = E
E ::= E RelOp E
E ::= not E
E ::= Id
```

`RelOp` est maintenant une unité reconnue lexicalement dont la « valeur » vaut soit `and` soit `or`.

3. Expliquez la démarche du concepteur de la grammaire. Pourquoi ne fait-il pas pareil avec les opérateurs `+` et `=` ?
4. Donnez les ensembles *First* et *Follow* pour la grammaire augmentée de `E' ::= E$.`
5. A l'aide de l'automate LR(0) fourni en annexe, déterminez **l'ensemble** des conflits *shift-reduce* et *reduce-reduce*.
6. En vous servant des indications de précédence et d'associativité, montrez comment résoudre les conflits précédents (**la table d'actions n'est pas demandée**).
7. Donnez la séquence d'actions effectuée par l'analyseur pour le mot suivant :

   `not a = b + c = d and e`
8. L'opérateur `=` est donc associatif à gauche, ce qui autorise des expressions telles que ci-dessus qui n'ont pas beaucoup de sens. Peut-on résoudre facilement le problème en modifiant la table de l'analyseur syntaxique de façon à interdire de telles expressions non parenthésées (on qualifie un tel opérateur de **non associatif**) ? Le nouvel analyseur doit terminer en échec sans pouvoir dépasser le second caractère `=` .