TD sur l'analyse syntaxique février 2008

Prérequis : Grammaire LL(1), fonctionnement d'un analyseur non-récursif (prédictif) LL(1), récurivité à gauche, factorisation à gauche, calcul des ensembles $Vide$, $Premier$, $Suivant$, des symboles directeurs, construction de la table d'analyse.
Durée : 1 h 50

TD 17 – Analyseur descendant non-récursif

Analyseur non-récursif LL(1)

Préambule

Vous avez découvert dans ce module deux types d'analyseurs syntaxiques descendants :

- les analyseurs récursifs
- les analyseurs non-récursifs

Un analyseur non-récursif est un automate à pile dont le but est de reconnaître les mots du langage engendré par la grammaire à partir de laquelle on a construit l'analyseur. Une analyse syntaxique non-récurive LL(1) se décompose en deux parties :

(*i*) **construction de l'analyseur** : à partir d'une grammaire algébrique on construit un automate à pile déterministe qui reconnaît le langage engendré par la grammaire

(*ii*) **reconnaissance d'un mot** : pour un mot donné, on simule le fonctionnement de l'automate à pile précédemment construit pour savoir si le mot est reconnu

Construction de l'analyseur

Construire l'automate à pile consiste à déterminer la table de transition de l'automate dans la mesure où son fonctionnement est toujours le même (l'algorithme de simulation de l'automate est le même quelle que soit la grammaire). Cette table est construite à partir des symboles directeurs des règles de la grammaire. La détermination de ces derniers ne peut cependant se faire qu'à condition que la grammaire vérifie un certain nombre de propriétés : elle ne doit pas être ambiguë, elle doit être réduite, non récursive-gauche et factorisée. En résumé, pour construire un analyseur non-récursif d'une grammaire, il faut :

- vérifier que la grammaire n'est pas ambiguë
- réduire la grammaire si nécessaire
- dérecursiver la grammaire si nécessaire
- factoriser la grammaire si nécessaire
- déterminer les non-terminaux produisant le vide
- calculer les premiers et les suivants des non-terminaux
- en déduire les symboles directeurs des règles
- construire la table d'analyse

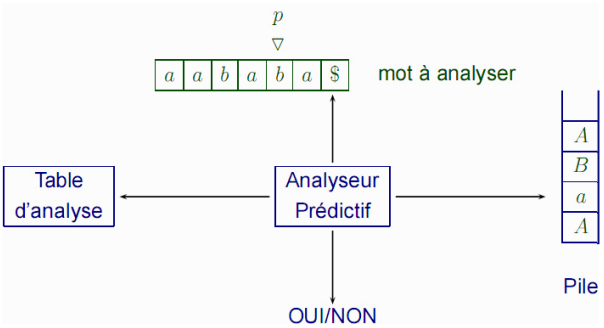
Reconnaissance d'un mot

La reconnaissance d'un mot consiste simplement à simuler le fonctionnement de l'automate à pile construit à partir de la table d'analyse déterminée précédemment.

- si le sommet de la pile est un non-terminal E alors
  - si $p \triangleright a$ et que $table[E, a] = erreur$, alors le mot est refusé
  - sinon, dépiler E et empiler $table[E, a]$
- si le sommet de la pile est un terminal a alors

- si $p \triangleright a$ alors dépiler a et avancer p
- si $p \triangleright b$ avec $b \neq a$ alors le mot est refusé

• Si la pile est vide alors
- si $p \triangleright \$$ alors le mot est accepté
- sinon le mot est refusé



Objectif du TD

Le début d'un constructeur d'analyseur syntaxique non-récursif LL(1) a été implanté. La partie relative à la reconnaissance d'un mot est complète mais il vous est demandé de terminer la partie concernant la génération de la table d'analyse. Pour cela, il est recommandé de reconstruire « à la main » cette table d'analyse pour les exemples étudiés lors des TD précédents.

Méthodes à compléter

La liste des méthodes à compléter est la suivante :

1. *grammaire*.Regle boolean **produitVide()**
2. *grammaire*.Grammaire void **calculerVide()**
3. *grammaire*.Regle ArrayList **premiers()**
4. *grammaire*.Regle HashSet **suivants(NonTerminal N)**
5. *grammaire*.Grammaire ArrayList **symboleDirecteur(Regle r)**
6. *grammaire*.Grammaire void **remplirTablePredictive()**

1. boolean produitVide()

Soit $r = A \rightarrow \alpha_1 \dots \alpha_n \in$ *Class Regle*.
Postcondition : `r.produitVide() == true` ssi

$$\forall i \in [1, n],\ \alpha_i = \epsilon \text{ ou } \alpha_i \in N \text{ et } \alpha_i \overset{*}{\rightarrowtail} \epsilon$$

Remarques :
- `r.element(i)` permet d'obtenir $\alpha_i \in$ *Class Lettre*
- `r.taille()` retourne n
- $\forall l \in$ *Class Lettre*, `l.produitVide()` retourne *true* ssi $l = \epsilon$ ou $l \in N$ et $l \overset{*}{\rightarrowtail} \epsilon$

2. void calculerVide()

Soit $G = (N, T, \rightarrow, X) \in$ *Class Grammaire*.
Précondition : `G.ensembleVide` $= \varnothing$
Postcondition : `G.ensembleVide` $= \{\ A \in N \mid A \overset{*}{\rightarrowtail} \epsilon\ \}$
Remarques :

- `G.ensembleVide.add(A)` avec $A \in$ *Class NonTerminal* permet ajouter le non terminal A à `G.ensembleVide`
- `G.nombreRegles()` retourne le nombre de règles de G
- `G.regleNumero(i)` retourne la i^e règle de G

3. ArrayList premiers()

Soit $r = A \rightarrow \alpha_1 \dots \alpha_n \in$ *Class Regle*.

Postcondition : `r.premier()` $= \{a \in T \mid \alpha_1 \dots \alpha_n \overset{*}{\rightarrowtail} a\beta \}$

Remarques :

- `r.element(i)` permet d'obtenir $\alpha_i \in$ *Class Lettre*
- `r.taille()` retourne n
- $\forall A, B \in$ *Class Lettre*, `A.equals(B)` retourne *true* ssi A et B représentent la même lettre (non terminale ou terminale)
- $\forall l \in$ *Class Lettre*, `l.premiers()` retourne $\{l\}$ si l est un terminal et $Premier(l)$ si l est un non terminal (si cet ensemble n'est pas encore calculé, le calcul sera lancé)
- $\forall E, F \in$ *Class Collection*[1], `E.addAll(F)` réalise l'opération : $E \leftarrow E \cup F$
- $\forall E \in$ *Class Collection*, `E.contains(element)` retourne *true* ssi *element* $\in E$

4. HashSet suivants(NonTerminal N)

Soient $r = A \rightarrow \alpha_1 \dots \alpha_n \in$ *Class Regle* et N un non terminal.

Postconditions :

- $Premier(\alpha_{i+1} \dots \alpha_n) \subset$ `r.suivants(N)` si $\alpha_i = N$
- $Suivant(A) \subset$ `r.suivants(N)` si $\alpha_n = N$
- $Suivant(A) \subset$ `r.suivants(N)` si $\alpha_i = N$ et $\forall j \in [i+1, n],\ \alpha_j \overset{*}{\rightarrowtail} \epsilon$

5. ArrayList symboleDirecteur(Regle r)

Soient $G = (N, T, \rightarrow, X) \in$ *Class Grammaire* et $r = A \rightarrow \alpha_1 \dots \alpha_n \in$ *Class Regle*.

Postcondition : `G.symboleDirecteur(r)` $= SD(A \rightarrow \alpha_1 \dots \alpha_n)$

6. void remplirTablePredictive()

Soit $G = (N, T, \rightarrow, X) \in$ *Class Grammaire*.

Postcondition : $\forall A \in N,\ s \in T,\ r \in \rightarrow$

$$tablePredictive[A, s] = r \quad \Leftrightarrow \quad s = SD(r) \text{ et } \exists \beta \in (N \cup T)^*,\ r = A \rightarrow \beta$$

Remarques :

- `G.tablePredictive` est l'attribut représentant la table prédictive
- $\forall A \in$ *Class NonTerminal*, $s \in$ *Class Terminal* et $r \in$ *Class Regle*, `tablePredictive.set(A, s, r)` ajoute la règle r dans la cellule dont la ligne est celle du non terminal A et la colonne celle du terminal s

[1] *ArrayList*, *HashSet*, ...