TD n°3, I45 – Projet
L2 Sciences et Technologies
Université du Sud Toulon-Var
Année universitaire 2005 – 2006

Emmanuel Bruno

Le module d'analyse syntaxique

Objectif

Poser les bases du module d'analyse syntaxique. Cela sera fait au travers de l'étude détaillée de l'analyse des expressions arithmétiques et logiques. Nous verrons comment définir la grammaire d'un langage et comment implanter simplement un analyse (descendant récursif).

Rappel sur les grammaires

Une grammaire permet de décrire l'ensemble des mots d'un langage (celui-ci peut être infini). Une règle de grammaire est composée de deux parties : la partie gauche est un symbole non-terminal qui peut être remplacé (on dira substitué) par la partie droite de la règle composée d'un mélange de symboles terminaux et non-terminaux. On utilisera le symbole S pour désigner la première règle de grammaire. Par exemple :

Grammaire	***Langage***

`S = AB`
`A = 'a'|'b'`
`B = 'c'|'d'`
	L={ac, ad, bc, bd}

`S = 'a'B` `B = B'b'|'b'`	L={ab, abb, abbb, …} récursive à gauche
`S = 'a'B` `B = 'b'B|'b'`	L={ab, abb, abbb, …} récursive à droite

Exercice 1 : des langages simples

Écrire les grammaires associées aux langages suivants :

- L={0p,1i,2p,3i,4p,5i,6p,7i,8p,9i} ; les chiffres pairs sont suivis de *'p'* les impairs de *'i'*.
- L={'ab', 'aab', 'aabbb', …} ; les mots de L sont composés d'une suite de *'a'* suivie d'un suite de *'b'* (*i*'a'* puis *j*'b'*).
- L={'ab', 'aabb', 'aaabbb', …} ; les mots de L sont composés de *n 'a'* suivis de *n 'b'*.

Les expressions arithmétiques

Exercice 2 : expressions arithmétiques très simples

Proposer une grammaire vérifiant les expressions arithmétiques simples dont les opérateurs seraient '+', *'*'* et les valeurs *'a', 'b', 'c', 'd'*. Remarquer qu'une grammaire associe une structure à une expression. Vérifier cette grammaire sur des exemples simples et construire le ou les arbres syntaxiques pour les expressions `a+b+c` et `a*b+c`.

Exercice 3 : un et un seul arbre syntaxique

Dans l'exercice précédent nous avons vu que la grammaire associe une (ou plusieurs) structure(s) aux expressions (un arbre syntaxique). Or, pour qu'une analyse ne soit pas ambiguë il faut que l'arbre associé soit **unique**. De plus, dans le cas des expressions arithmétiques, cette structure doit respecter les priorités relatives des opérateurs. Proposer une grammaire qui permette cela : on distinguera les *termes* et les *facteurs*. Vérifier cette grammaire sur les exemples suivants : `a*b+c, a+b*c, (a+b)*(c+d).`

Grammaire du langage Algo

La grammaire complète du langage ***Algo*** vous sera fournie en annexe. Les règles décrivant les expressions arithmétiques s'appuient sur les principes précédents. Elles ne sont ni ambiguës, ni récursives à gauche. De plus, la lecture d'un symbole (de la gauche vers la droite) permet de savoir quelle règle doit être appliquée. On peut donc s'en servir pour construire un analyseur descendant récursif.

Les bases de l'analyse descendante

L'analyse descendante

Dès lors que l'on a définit une telle grammaire, on peut construire un analyseur descendant. Celui-ci s'appuie sur un analyseur lexical pour lire les symboles (*symbole_suivant)*. Par exemple, pour la grammaire `A = 'a' A 'c' | 'b'`, on a l'analyseur syntaxique suivant :

```
fonction A()
```

```
début
si symbole = 'a' alors
     symbole_suivant()
     A()
     si symbole = 'c' alors symbole_suivant()
     sinon ERREUR
     finsi
sinon
     si symbole = 'b' alors symbole_suivant()
     sinon ERREUR
finsi
fin
```

Un algorithme d'analyse est donc définit pour chaque *non-terminal* de la grammaire. Il est exprimé comme une procédure portant le nom du symbole. Chaque occurrence de ce symbole dans la grammaire sera donc transformée en l'appel de cette procédure. Mais **attention** : pour que cela soit possible, il ne faut pas que deux règles que l'on peut dériver au même moment puissent commencer par le même symbole. Ainsi :

La grammaire… devient…	
`A = B|C`	`A = a(B|C)`
`B = ab`	`B = b`
`C = ac`	`C = c`

Exercice 4 : analyse descendante d'expressions simples

Sur le même principe proposer un algorithme (composé de plusieurs fonctions) pour l'analyse des *expression-simple*s (cf. grammaire en annexe). Définir les fonctions `facteur`, `terme` et `expression_simple`.

Exercice 5 : arbre syntaxique d'un facteur

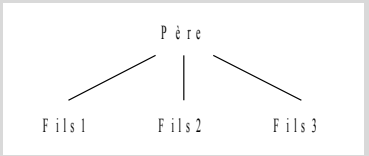
Fonctions disponibles

L'arbre syntaxique est un arbre n-aire. Pour le construire, nous disposons de deux fonctions :

- `creer_racine(noeud *racine, char *nom)` qui crée un nœud dont le label est `nom`
- `ajouter_fils(noeud racine, noeud fils)` qui ajoute un fils à un nœud déjà existant

Exemple :

```
creer_racine(&racine, 'Père');
creer_racine(&fils, 'Fils1');
ajouter_fils(racine, fils);
creer_racine(&fils, 'Fils2');
ajouter_fils(racine, fils);
creer_racine(&fils, 'Fils3');
ajouter_fils(racine, fils);
```



Proposer une modification de l'algorithme de lecture d'un facteur pour qu'il construise l'arbre syntaxique correspondant en utilisant les fonctions `creer_racine` et `ajouter_fils`.

De même, proposer une modification de l'algorithme de lecture d'un terme pour qu'il construise l'arbre correspondant. On supposera que l'algorithme `facteur(arbre_facteur)` construit l'arbre du facteur dans la variable `arbre_facteur`. Faire fonctionner l'algorithme sur l'expression `12 * 100 mod 3 div 5`.

TD n°3, I45 – Projet
L2 Sciences et Technologies
Université du Sud Toulon-Var
Année universitaire 2005 – 2006

Emmanuel Bruno

Le module d'analyse syntaxique (1) : correction

Exercice 1 : des langages simples

```
S = P'p'|I'i'
P = '0'|'2'|'4'|'6'|'8'
I = '1'|'3'|'5'|'7'|'9'
```

```
S = AB
A = 'a'A|'a'
B = 'b'B|'b'
```

```
S=P
P='a'P'b'
```

Exercice 2 : expressions arithmétiques très simples

```
expression = 'a' | 'b' | 'c' | 'd'
           | expression '+' expression
           | expression '*' expression
```

Exercice 3 : un et un seul arbre syntaxique

```
expression = terme {'+' terme}
terme = facteur {'*' facteur}
facteur = valeur | '(' expression ')'
valeur = 'a' | 'b' | 'c' | 'd'
```

Exercice 4 : analyse descendante d'expressions simples

```
fonction facteur()
début
au cas où lex_sym vaut
     s_ident alors lire_symbole()
     s_réel alors lire_symbole()
     s_entier alors lire_symbole()
     s_vrai alors lire_symbole()
     s_faux alors lire_symbole()
     s_parent_g alors
          expression()
          si lex_sym différent de s_parent_d
                alors ERREUR
          sinon lire_symbole()
          finsi
     s_non alors
          lire_symbole()
          facteur()
     défaut alors ERREUR
fin cas
fin
```

```
fonction terme()
début
facteur()
tant que ((lex_sym=s_etoile) ou (lex_sym=s_slash) ou
(lex_sym=s_div) ou (lex_sym=s_mod) ou (lex_sym=s_et))
faire
     lire_symbole()
     facteur()
fin tant que
fin
```

```
fonction expression-simple()
début
au cas où lex_sym vaut
     s_plus alors lire_symbole()
     s_moins alors lire_symbole()
fin cas
terme()
tant que ((lex_sym=s_plus) ou (lex_sym=s_moins) ou
(lex_sym=s_ou)) faire
     lire_symbole()
     terme()
fin tant que
fin
```

Exercice 5
arbre syntaxique d'un facteur

```
procédure facteur (abre_facteur : noeud)
arbre_valeur : noeud
début
     selon que lex_sym vaut
           s_ident alors
                 creer_racine(arbre_facteur,
'identificateur')
                 creer_racine(arbre_valeur, lex_val)
                 ajouter_fils(arbre_facteur,
arbre_valeur)
                 lire_symbole
           s_réel alors
                 creer_racine(arbre_facteur, 'réel')
                 creer_racine(arbre_valeur, lex_val)
                 ajouter_fils(arbre_facteur,
arbre valeur)
                 lire_symbole
           s_entier alors
                 creer_racine(arbre_facteur, 'entier')
                 creer_racine(arbre_valeur, lex_val)
                 ajouter_fils(arbre_facteur,
arbre_valeur)
                 lire_symbole
           s_parent_g alors
                 lire_symbole
                 arbre_valeur=expression()
                 si lex_sym != s_parent_d
                       alors ERREUR
                       sinon

     creer_racine(arbre_facteur, 'parenthèse')

     ajouter_fils(arbre_facteur, arbre_valeur)
                           lire_symbole
                 finsi
           s non alors
                       creer_racine(arbre_facteur,
'négation')
                       lire_symbole
                       facteur(arbre_valeur)
                       ajouter_fils(arbre_facteur,
arbre_valeur)
     finsq
fin
```

arbre syntaxique d'un terme

```
procédure expression (arbre_terme : noeud)
arbre_avant : noeud
arbre_facteur : noeud
début
     arbre_avant=facteur()
     tant que lex_sym = s_etoile ou s_slash ou s_div
ou s_mod ou s_et faire
           selon que lex_sym vaut
                 s_etoile alors
creer_racine(arbre_terme, 'multiplier')
                 s_slash alors
creer_racine(arbre_terme, 'diviser')
                 s_div alors creer_racine(arbre_terme,
'div')
                 s_mod alors creer_racine(arbre_terme,
'modulo')
                 s_et alors creer_racine(arbre_terme,
'et')
           finsq
           lire_symbole
           arbre_facteur=facteur()
           ajouter_fils(arbre_terme, arbre_avant)
           ajouter_fils(arbre_terme, arbre_facteur)
     fintq
fin
```