Université Bordeaux
Licence, module Compilation, 2017/2018

TD4 – Analyse syntaxique

Exercice 4.1

Grammaire non LARL. Soit la grammaire suivante écrite au format *bison*,

```
statement_list:
        statement_list statement
        | statement
        ;

statement:
        IDENTIFIER AFF expression SEMICOLON
        IF expression THEN statement
        | IF expression THEN statement ELSE statement
        | WHILE expression DO statement
        ;

expression:
        IDENTIFIER
        | INTEGER
        | expression PLUS expression
        | expression MINUS expression
        | expression TIMES expression
        | expression DIV expression
        | expression OR expression
        | expression AND expression
        | MINUS expression
        | NOT expression
        | expression LT expression
        | expression LE expression
        | expression GT expression
        | expression GE expression
        | expression EQ expression
        | expression DIFF expression
        | LPAR expression RPAR
        ;
```

Où les terminaux correspondent à ce tableau :

AFF `:=`	SEMICOLON `;`	IF `if`	THEN `then`	ELSE `else`	WHILE `while`	DO `do`	
PLUS `+`	MINUS `-`	TIMES `*`	DIV `/`	OR `||`	AND `&&`	NOT `!`	
LT `<`	LE `<=`	GT `>`	GE `>=`	EQ `=`	DIFF `!=`	LPAR `(`	RPAR `)`

1. Écrire un analyseur syntaxique qui respecte ces productions et le compiler avec l'option de débogage `-v`.
2. Utiliser les règles de précédence des opérateurs pour corriger les erreurs affichées et pour respecter la sémantique usuelle des expressions et instructions.

Exercice 4.2

Définition dirigée par la syntaxe pour une expression de la logique propositionnelle. Manipulation algébrique des expressions.

1. Écrire en langage C une structure de données $Tree$ permettant de représenter un arbre binaire $t(tag, value, lhs, rhs)$. Où
   — tag est une valeur d'un énuméré `enum Tag {AND, OR, NOT, IMPL, EQ , CONSTANT , VARIABLE}` désignant un opérateur logique ($\wedge, \vee, \neg, \rightarrow, \leftrightarrow$), une constante ou une variable.
   — value est une valeur dans $\{0, 1\}$ désignant une constante, ou une valeur entière i désignant une variable p_i.
   — lhs et rhs (*left/right hand-side*) désignent les branches gauche (resp. droite) de l'arbre binaire.
2. Écrire en langage C une routine `struct Tree *createTree()` qui alloue une structure `Tree` pour représenter une expression logique.
3. Écrire en langage C une routine `void printTree(struct Tree *tree)` qui affiche un arbre sous la forme d'une formule de la logique propositionnelle.
4. Écrire une définition dirigée par la syntaxe en `lex/yacc` (`flex/bison`) pour obtenir une expression de la logique propositionnelle dans le format précédent. Pour faciliter l'écriture de la grammaire en *Yacc*, on utilisera les propriétés de précédence des opérateurs. Nous rappelons que les opérateurs $\vee, \rightarrow, \leftrightarrow$ sont prioritaires sur $\wedge$ et que $\neg$ est prioritaire sur tous les autres. Par ailleurs, tous les opérateurs binaires sont associatifs à gauche.
5. Mettre sous forme normale négative une formule propositionnelle donnée et l'afficher. Pour cela, créer deux fonctions $fnn()$ et $neg()$ définies ainsi :

   $fnn(0) = 0$
   $fnn(1) = 1$
   $fnn(p_i) = p_i$
   $fnn(\neg P) = neg(P)$
   $fnn(P \vee Q) = fnn(P) \vee fnn(Q)$
   $fnn(P \wedge Q) = fnn(P) \wedge fnn(Q)$
   $fnn(P \rightarrow Q) = neg(P) \vee fnn(Q)$
   $fnn(P \leftrightarrow Q) = (fnn(P) \wedge fnn(Q)) \vee (neg(P) \wedge neg(Q))$
   $neg(0) = 1$
   $neg(1) = 0$
   $neg(p_i) = \neg p_i$
   $neg(\neg P) = fnn(P)$
   $neg(P \wedge Q) = neg(P) \vee neg(Q)$
   $neg(P \vee Q) = neg(P) \wedge neg(Q)$

$neg(P \to Q) = fnn(P) \wedge neg(Q)$
$neg(P \leftrightarrow Q) = (fnn(P) \wedge neg(Q)) \vee (neg(P) \wedge fnn(Q)$

Exemple :

$$((p \to q) \to p) \to p$$

devra afficher

$$((\neg p \vee q) \wedge \neg p)) \vee p$$

Exercice 4.3

Une grammaire attribuée est une extension d'une grammaire algébrique $G = (\Sigma, V_N, S, R)$ où

- Dans un arbre de dérivation quelconque, on attache à chaque noeud étiqueté par X un ensemble fini d'attributs : $A(X) = \{a_1(X), \ldots, a_n(X)\}$.
  $A(X)$ est partitionné en deux ensembles disjoints : $A_s(X)$ et $A_h(X)$, respectivement les attributs synthétisés, et les attributs hérités. Les attributs synthétisés $A_s(X)$ sont ceux dont le calcul dépend des attributs attachés aux noeuds en dessous de X dans l'arbre de dérivation. Les attributs hérités $A_h(X)$ sont ceux qui sont calculés à partir des noeuds frères ou père de X.
- On associe à chaque production $p = X_0 \to X_1 \ldots X_n$ un ensemble de règles $R(p) = \{a_n(X_i) = f(a_m(X_j), \ldots)\}$

Prenons un exemple simple inspiré de [Knuth, 1968], dans le domaine des grammaires attribuées et associant à chaque nombre binaire, sa valeur décimale.

- $\Sigma = \{0, 1\}$
- S, N, B sont les symboles de la grammaire (B pour les chiffres binaires et N pour les nombres exprimés avec ces chiffres). On distinguera N_1 et N_2, deux occurrences différentes du même symbole N dans une même règle.
- S est le symbol initial
- Pour tout noeud N ou B de l'arbre de dérivation, v est un attribut synthétisé et s un attribut hérité. Le premier contient la valeur du nombre binaire, le second correspond au décalage du chiffre à partir de la droite.

$N \to B$	$v(N) = v(B), s(B) = s(N)$
$N_1 \to N_2 B$	$v(N_1) = v(N_2) + v(B), s(N_2) = s(N_1) + 1, s(B) = s(N_1)$
$B \to 0$	$v(B) = 0$
$B \to 1$	$v(B) = 2^{s(B)}$
$S \to N$	$v(S) = v(N), s(N) = 0$

1. Décrire dans le détail le calcul de la valeur décimale du nombre 1001. On dessinera un arbre de dérivation décoré d'un graphe des attributs hérités et d'un graphe des attributs synthétisés.
2. Pour quelle(s) raison(s) n'est-il pas possible d'écrire une grammaire *YACC* qui implémente immédiatement la grammaire précédente ?
3. Tenter d'écrire une grammaire *YACC* qui implémente cette grammaire en utilisant les différents outils de la programmation impérative C.

Références

[Knuth, 1968] Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, 2(2) :127–145.