UNIVERSITY OF AMSTERDAM

ASSIGNMENT 3

Syntactic Analysis

February 27, 2021

Students:
René Kok
13671146

Aram Mutlu
13574116

Lecturer:
Dhr. dr. C.U. Grelck

Course:
Compiler Construction

Course code:
5062COMP6Y

1 Introduction

This report provides information regarding the third assignment "Syntactic Analysis" of the course Compiler Construction lectured by Dhr. dr. C.U. Grelck. The goal of this assignment is to derive pseudo code for a top-down recursive-descent parse. To accomplish this we will start to rewrite the given grammar according to the C standard. From there we can convert the left-recursive grammar into a right-recursive grammar and convert it into a start-seperated, predictive grammar. With this solution we can create pseudo code for the goal of this assignment. The given grammar for this assignment:

Expr	$\Rightarrow$	Id
	|	Expr + Expr
	|	– Expr
	|	Expr ++
	|	(Expr)

2 Precedence and Associativity

The first part of the assignment is to rewrite the given grammar such that properly expresses precedence and associativity according to the C standard. In C there are 17 precedence levels to check on. So we have to device our grammar into multiple levels. For this grammar we need 4 levels (+, -, ++ and (...) / id). If we separate these we will get the following grammar:

$$\begin{array}{lcl}
\text{Expr4} & \Rightarrow & \text{Expr4} + \text{Expr3} \\
 & | & \text{Expr3} \\
\\
\text{Expr3} & \Rightarrow & - \text{Expr3} \\
 & | & \text{Expr2} \\
\\
\text{Expr2} & \Rightarrow & \text{Expr2} ++ \\
 & | & \text{Expr1} \\
\\
\text{Expr1} & \Rightarrow & (\text{ Expr4 }) \\
 & | & \text{Id}
\end{array}$$

Figure 1: Grammar of expressions with proper precedence and associativity

3 Left- and Right-recursive Grammars

Based on the grammar we created in the first part of the assignment, we can now convert it into a right-recursive grammar. A right-recursive grammar is said to be right recursive if it has right recursion. This means the tree of the grammar will grow to the right. If we change our grammar into a right-recursive grammar, we will get:

```
Expr4    ⇒  Expr3 Expr4'
Expr4'   ⇒  + Expr4 Expr3
         |  ϵ

Expr3    ⇒  Expr2 Expr3'
Expr3'   ⇒  − Expr3
         |  ϵ

Expr2    ⇒  Expr1 Expr2'
Expr2'   ⇒  ++ Expr2
         |  ϵ

Expr1    ⇒  Id Expr1'
Expr1'   ⇒  ( Expr4 )
         |  ϵ
```

Figure 2: Right-recursive grammar of expressions with proper precedence and associativity

In this grammar we can see that each level in the grammar gets a new inner level named with prime and ends with an ϵ (epsilon) which means an end of line.

4 Predictive Grammars

Following grammar is a start-seperated and predictive grammar. A predictive grammar is one where it's possible decide the right rule by looking at the first token or first N tokens.

```
Start    ⇒  Expr4

Expr4    ⇒  Expr3 Expr4'
Expr4'   ⇒  + Expr4 Expr3
         |  ϵ

Expr3    ⇒  Expr2 Expr3'
Expr3'   ⇒  − Expr3
         |  ϵ

Expr2    ⇒  Expr1 Expr2'
Expr2'   ⇒  ++ Expr2
         |  ϵ

Expr1    ⇒  Id Expr1'
Expr1'   ⇒  ( Expr4 )
         |  ϵ
```

Figure 3: Right-recursive grammar of expressions with proper precedence and associativity

5 Recursive-descent Parsing

```
/**
 * Authors: Rene Kok & Aram Mutlu
 * Pseudo code for a top-down recursive-descent parser
 * from the start-separated, predictive grammar of Assignment 3.3.
 */
Start() {
    return Expr4() && (nextToken () == eof);
}

Expr4() {
    return Expr3() && Expr4P();
}

Expr4P() {
    switch (token = nextToken()) {
        case Addition: return Expr4() && Expr3();
        default: unget(token);
                 return true;
    }
}

Expr3() {
    return Expr2() && Expr3P();
}

Expr3P() {
    switch (token = nextToken()) {
        case UnaryMinus: return Expr3();
        default: unget(token);
                 eturn true;
    }
}

Expr2() {
    return Expr1() && Expr2P();
}

Expr2P() {
    switch (token = nextToken()) {
        case PrefixIncrement: return Expr2();
        default: unget(token);
                 return true;
    }
}

Expr1() {
    return (nextToken() == Id) && Expr1P();
}

Expr1P() {
    switch (token = nextToken()) {
        case (: return Expr4();
        default: unget(token);
                 return true;
    }
}
```

```
Id() {
    return nextToken() == Id;
}
```