University of Amsterdam

Assignment 3

Syntactic Analysis

February 24, 2021

Students:
René Kok
13671146

Aram Mutlu
13574116

Lecturer:
Dhr. dr. C.U. Grelck

Course:
Compiler Construction

Course code:
5062COMP6Y

1 Precedence and Associativity

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

Figure 1: Grammar of expressions with proper precedence and associativity

2 Left- and Right-recursive Grammars

```
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

3 Predictive Grammars

Following grammar is a start-seperated and predictive grammar. A predictive grammer 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

4 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;
}
```