TCS ASSIGNMENT 2

Rosemary Tropiano & Luke Ratcliffe

October 2019

1 The Lexer

The lexer takes an input string and then manipulates it into tokens which are then passed to the parser (Syntax Analyser) for syntactical analysis. The job of the lexical analyser is to make sure that everything sent to the parser is a valid token. If there is an invalid token, the lexer will throw a LexicalException.

The input string that the lexer receives is split using a regular expression. It splits based on whitespace and specific symbols (the terminals) and then the resulting individual strings are inserted into an array of strings.

Using if-statements, this array is iterated over, and each element is converted into a token, and sent to the list of tokens. The Token class (given) has an enumerator which lists all of the different token types. These are:

```
 PLUS, MINUS, TIMES, DIVIDE, MOD, ASSIGN, EQUAL, NEQUAL, LT, LE, GT,
GE, LPAREN, RPAREN, LBRACE, RBRACE, AND, OR, SEMICOLON, PUBLIC, CLASS,
STATIC, VOID, MAIN, STRINGARR, ARGS, TYPE, PRINT, WHILE, FOR, IF, ELSE,
DQUOTE, SQUOTE, ID, NUM, CHARLIT, TRUE, FALSE, STRINGLIT
```

Each of these tokens represents a terminal in the Simple Java Grammar (given) and each of the elements in the array (the one we made by splitting the input string) must be turned into one of these tokens. This is done by using the statements:

```
Token.TokenType type = Token.TokenType.TERMINAL;
Token token = new Token(type, "terminal");
tokenList.add(token);
```

where `TERMINAL` is a token from the enum in the Token class, and `"terminal"` is the string representation of that token in double quotes. The standard, single character terminals could use this simple code, represented in the method tokenID in our program, however we also had to account for variable IDs, strings, characters, and numbers.

The method `variableID` handled the cases where the token was `TRUE, FALSE, ID,` or `CHARLIT`. The program is sent to this method if it detects a string and it isn't handled by any code block that comes before it. To test whether the string should be converted into the `TRUE` or `FALSE` tokens, the if statement whether the string equalled the string "true" or "false". To determine whether the string should be converted into the token `CHARLIT` the if statement asked whether the string was of length 1. If it is a string of length greater than 1, the string is an `ID` , else, it returns a LexicalException.

The code is sent to stringID if it detects double quotes, and returns the token STRINGLIT.

The code is sent to numberID if it attempts to parse the string to an integer and is successful, returning the token NUM.

2 The Parser

The role of the parser is to make sure that the tokens are in the correct order and make syntactical sense. Where a lexer can be described using Finite Automata, a parser can be described using the elements of a Push Down Automata, which include the stack and parse tree.

This parser is an LL(1) parser. LL stands for *left-to-right* and *leftmost derivation*. The '1' indicates that the parser can look ahead one token. Left-to-right derivation means that when the rules of the grammar are being parsed, the first non-terminal to be replaced with a terminal is on the left. Left-most derivation determines the order that the steps of the grammar will be taking in. If a language is ambiguous (has more than one result when the left-derivation is taken) or left recursive (contains rules of the form $A \rightarrow A\alpha$) it cannot be in the form LL(1).

This is also a top down parser in that it starts to build the parse tree from the start symbol (our \$) followed by the input.

In order to program the Syntactical Analyser, the grammar must first have all left recursion removed. Because the left recursion had already been removed, the FIRST and FOLLOW sets had to be computed. Then a Parsing Table was constructed.

2.1 The Parse Tree

The parse tree shows the derivation of the grammar in graphical form. The start symbol (the \$) would be the root of our parse tree as it is the first thing the parser sees. Each leaf node on the tree is a terminal and each node inside the the tree that isn't a leaf node is a nonterminal. For a LL(1) language, the parse tree is read left to right. This tree is based on the parsing table discussed in section 3.3 and is used in our program to evaluate the input and determine what the next valid symbol should be.

3 The First and Follow Sets

3.1 The FIRST sets

FIRST sets are computed by taking each variable and determining which terminal(s) could be the first to be read.

FIRST(<<prog >>) = {public}
FIRST(<<los >>) = {while, for, if, ID, int, boolean, char, System.out.print, ;, ϵ}
FIRST(<<stat >>) = {while, for, if, ID, int, boolean, char, System.out.print}
FIRST(<<while >>) = {while}
FIRST(<<for >>) = {for}
FIRST(<<for start >>) = {int, char, boolean, ID, ϵ}
FIRST(<<for arith >>) = {(, ID, NUM, ϵ}
FIRST(<<if >>) = {if}
FIRST(<<else if >>) = {else, ϵ}

FIRST(<<else?if >>) = {else}
FIRST(<<poss if >>) = {if, ϵ}
FIRST(<<assign >>) = {ID}
FIRST(<<decl >>) = {int, boolean, char}
FIRST(<<poss assign >>) = {=, ϵ} FIRST(<<print >>) {System.out.print}
FIRST(<<type >>) = {int, boolean, char}
FIRST(<<expr >>) = {(, ID, NUM, CHAREXPR, TRUE, FALSE}
FIRST(<<bool expr' >>) = {==, ! =, &&, ||, ϵ}
FIRST(<<bool op >>) = {==, ! =, &&, ||}
FIRST(<<bool eq >>) = {==, ! =}
FIRST(<<bool log >>) = {&&, ||}
FIRST(<<rel expr >>) = {(, ID, NUM, true, false}
FIRST(<<rel expr' >>) = {<, <=, >, >=, ϵ}
FIRST(<<rel op >>) = {<, <=, >, >=}
FIRST(<<arith expr >>) = {(, ID, NUM}
FIRST(<<arith expr' >>) = {+, −, ϵ}
FIRST(<<term >>) = {(, ID, NUM}
FIRST(<<term' >>) = {*, /, %, ϵ}
FIRST(<<factor >>) = {(, ID, NUM}
FIRST(<<print expr >>) = {",(, ID, NUM}

3.2 The FOLLOW sets

Follow sets are computed by determining which terminal(s), for each variable could come after that variable is read.

FOLLOW(<<prog >>) = {$}
FOLLOW(<<los >>) = {}}
FOLLOW(<<stat >>) = {while, for, if, ID, int, boolean, char, System.out.print, ;}
FOLLOW(<<while >>) = {while, for, if, ID, int, boolean, char, System.out.print, ;}
FOLLOW(<<for >>) = {while, for, if, ID, int, boolean, char, System.out.print, ;}
FOLLOW(<<for start >>) = {;}
FOLLOW(<<for arith >>) = {)}
FOLLOW(<<if >>) = {while, for, if, ID, int, boolean, char, System.out.print, ;}
FOLLOW(<<else if >>) = {while, for, if, ID, int, boolean, char, System.out.print, ;}
FOLLOW(<<else?if >>) = {{}
FOLLOW(<<poss if >>) = {{}
FOLLOW(<<assign >>) = {;}
FOLLOW(<<decl >>) = {;}
FOLLOW(<<poss assign >>) = {;}
FOLLOW(<<print >>) = {while, for, if, ID, int, boolean, char, System.out.print, ;}
FOLLOW(<<type >>) = {ID}
FOLLOW(<<expr >>) = {;}
FOLLOW(<<bool expr' >>) = {), ;} FOLLOW(<<bool op >>) = {(, ID, NUM, true, false}
FOLLOW(<<bool eq >>) = {(, ID, NUM, true, false}
FOLLOW(<<bool log >>) = {ID, NUM, true, false}

FOLLOW(<<rel expr >>) = {==, ! =, &&, ||}
FOLLOW(<<rel expr' >>) = {==, ! =, &&, ||}
FOLLOW(<<rel op >>) = {(, ID, NUM}
FOLLOW(<<arith expr >>) = {), ==, ! =, &&, ||, <, <=, >, >=}
FOLLOW(<<arith expr' >>) = {), ==, ! =, &&, ||, <, <=, >, >=}
FOLLOW(<<term >>) = {+, -}
FOLLOW(<<term' >>) = {+, -}
FOLLOW(<<factor >>) = {*, /, %}
FOLLOW(<<print expr >>) = {)}

3.3 The Parsing Table

The parsing table is a basis for the parse tree that allows you to implement the rules in code.

The rules for the parsing table are as follows:

1. For each $A \to \alpha \in R$ apply steps 2 and 3.
2. For each $a \in \Sigma \cap$ FIRST(α), add the production $A \to \alpha$ to M[A, a].
3. If $\epsilon \in$ FIRST(α), add $A \to \alpha$ to M[A, b] for every $b \in \Sigma \cap$ FOLLOW(A).
   If \$ $\in$ FOLLOW(A), add $A \to \alpha$ to M[A, \$].

The table is thus created from the FIRST and FOLLOW sets. While creating the table, if there are any two entries that fit in the same cell, the grammar cannot be parsed using LL(1). There are no double entries in our parsing table. Hitting any blank cells in the table when parsing means an error has occurred, and the program should throw an error.

The table is too wide to be included in this document, but it is attached in the appendix.

4 Appendix

Find the Parsing Table in .xlsx format here –>https://tinyurl.com/tcsRoseLuke

5 References

Tutorials Point 2019, *Compiler Design - Top Down Parser*, n.a., viewed 19 October 2019, <https://www.tutorialspoint.com/compiler_design/compiler_design_top_down_parser.htm>.

Mathieson, L. (2019),*41080*, Slideshow, UTS, Sydney, 17 October 2019, <https://online.uts.edu.au/bbcswebdav/pid-3670108-dt-content-rid-53322090_1/courses/41080-2019-SPRING-CITY/Week6LectureHandout.pdf >