California State University, Fullerton

Assignment 2

Submitted by:
Janelle Estabillo
Andres Jaramillo
Jaime Lambrecht
Kevin Espinoza
Steven Tran

Submitted to:
Anthony Le
CPSC 323 Instructor

04/15/2021

I. Problem Statement

Implement a Syntax Analyzer using strategies similar to the predictive recursive descent parser or a top-down parser such as RDP. Using the Lexer() created in assignment one, get tokens and rework the parser function so that it prints the tokens, lexemes, and production rules to an output file.

II. Program Execution

A. Extract the zip file
B. To build and execute the program:

Build main.cpp using a C++ toolchain such as GCC or CLang, e.g:

```
$ g++ main.cpp
```

Run the output file (use either specified output filename or default "a.out"):

```
$ ./a.out [TEST FILE FILENAME]
```

If [TEST FILE FILENAME] is left blank, the program will use the default sample_input.txt

C. To view the output of the syntax analyzer, navigate back to the repo of the current project and open the text file named “parser_output.txt” Here you will view the separation of tokens and lexemes, as well as the syntax rules parsed by the program, based on the input test case file used.

III. Program Design

The program’s design is using the recursive descent approach. The Parsing process that outputs the tokens, lexemes, and production rules begins with the void function Parse(). Here, the syntax for outputting each type of production rule is established. This function consists of 9 possible production rules that are based on the type of input received from the test1assignmentOperator.txt file. The implementation of the rules is carried out by a series of functions that call each other according to their respective rule sets. Left-recursion is removed by functions such as “ExpressionPrime” or “TermPrime”, which are specialized in order to ensure that they do not enter an infinite loop. Strings for the output to illustrate the rule sets as they are encountered by the RDP are stored in a vector and printed by their respective functions.

Syntax for production rules

```
<Expression> -> <Expression> + <Term> | <Expression> - <Term> | <Term>
<Term> -> <Term> * <Factor> | <Term> / <Factor> | <Factor>
<Factor> -> ( <Expression> ) | <ID> | <Num>
<ID> -> id
<Statement> -> <Assign>
<Assign> -> <ID> = <Expression>;
<Statement> -> <Declarative>
<Declarative> -> <Type> <ID>
<Type> -> bool | float | int
<Conditional> -> <Expression> <Relop> <Expression> | <Expression>
<Expression> -> + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> |
<Term>
<ExpressionPrime> → <Expression> | ε
<Term> -> * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Factor>
<TermPrime> -> <Term> | ε
```

Major Function Components of the Program:

1. **void lexer();**
   This function will contain all the Lexer functions to properly separate all Tokens/Lexemes
2. **void parse();**
   This function contains the strategy and grammar that is provided in class and we do this by filling the syntax vector with proper output terms.
3. **void StatementList();**
   Checks if were at the end of the lexeme, if not call Statement() and Increment.
4. **void Statement();**
   This function checks if it belongs to Expression or Assignment and then makes the appropriate function calls. If the value is an equal sign “=”, it prints the lexeme and calls

the Assignment() Function. If we find “if”, “else”, “while”, “do”, and “for”, we call the Assignment() function.

5. **void Assignment();**

   This function checks if an assignment begins with an identifier. If it does, we increment and print the lexeme by calling the printLexeme() function. We increment one more time and call the Expression() function. If it doesn’t begin with an identifier. The function will print out an error message and exit.

6. **void Expression();**

   This function then prints the lexeme and calls the term() function. If it is a “+” or “-”, we increment and then call ExpressionPrime() function.

7. **void ExpressionPrime();**

   The function checks if it is a “+” or “-”, then it prints the lexeme, increment, print the lexeme once again and call Term() function and increment once again, and call ExpressionPrime() function.

8. **void Term();**

   This function prints out the term and calls the Factor() function. If “*” and “/” is found, increment and call TermPrime() function.

9. **void TermPrime();**

   The function checks if it is a “*” or “/”, then it prints the lexeme, increment, print the lexeme once again and call Factor() function and increment once again, and call TermPrime() function.

10. **void Factor();**

    It first checks if it has an identifier and is literal. If it is, it then checks if it is an integer or float. If not integer or float it throws in an error message. Then it checks if it is currently pointing to a “(“ if it is, it increments and calls the Expression() function. It increments again and checks if it has a “)” closing parenthesis. If not, it throws in an error message and exits the loop. If the function discovers that it is not an identifier and a literal, it throws the error message “Expected Identifier, number or parenthetical expression.” and exits the loop.

11. **void Num();**

    This function checks if it is literal and is an integer or float. If it is not, it throws in an error message saying “Invalid Number”.

12. **void printLexeme();**

    This function prints the token and lexeme.

IV. Limitations

Compound relational operators are not present in the lexer, instead they are checked as separate operators. Also, they can be checked but are not implemented correctly as it can be read even though there is a space between the characters (ex. ‘==’ and ‘= =’ will be read as the same compound operator.

V. Shortcomings

We did not implement “MoreIDs” to allow statements such as “int a, b, c;”. We also did not implement else, and we used curly braces instead of “then”, “whileend”, etc.