Assignment 3

CPSC 323-04

Iftekharul Islam

1. Problem Statement

The purpose of this assignment is to enhance the syntax analyzer from Assignment 2 by choosing one of many implementation options. I implemented a predictive recursive descent parser that is capable of intermediate code generation. In this implementation, intermediate code includes if-statements, while-loops, begin/end statements, and relational operators.

In a predictive recursive descent implementation, each non-terminal has a unique parsing function that recognizes distinct sequences of tokens that can possibly be generated by that non-terminal. In order to recognize these sequences, each function must call on other non-terminal parsing functions, resulting in a massive tree of computations. To prevent a function from calling itself and causing an endless recursion, the rules have been rewritten to eliminate left-recursion and back-tracking.

2. How to use the program

NOTE: Must have at least Python 3.7 in order to run.

Running in Tuffix:

1. Move files “Compiler.py”, “run.sh”, and all desired text input files to the same directory (three have been provided).
2. Open the Tuffix terminal.
3. Move to the directory holding all files within the terminal.
4. Run the program by typing either “sh run.sh” or “./run.sh” in the terminal.
5. When prompted, input the name of a file for the program to read. After doing so, the file will be analyzed and all tokens, lexemes, and rules used will be outputted to the terminal.

3. Design of the program

This program is written in Python. A LexicalAnalyzer class identifies all tokens from a given source file. A list consisting of a token, lexeme, and line number of that token is appended to another list that holds all the tokens of the entire file. This all-encompassing list is then passed to the SyntaxAnalyzer class. The SyntaxAnalyzer class primarily consists of an index variable that keeps track of which token is currently being analyzed and several methods that implement grammar rules.

The SyntaxAnalyzer class also creates a symbol table, which is a list of each identifier declared in the source file, the memory addresses they are located at, and their datatypes. By default, the initial memory address is set to 5000. This value is

incremented for each new identifier that is declared. The symbol table handling occurs in the functions dedicated to the <Declarative> and <More IDs> productions.
The following is a list of all the grammar rules implemented in this syntax analyzer:

<Statement> → <Assign> | <Expression Semicolon> | <Declarative> | <If Statement> | <While Statement> | <Begin Statement>

<Multiple Statements> → <Statement> <More Statements>

<More Statements> →; <Statement> <More Statements> | <Epsilon>

<If Statement> → if <Conditional> then <Multiple Statements> <Else Statement> endif

<Else Statement> → else <Multiple Statements> | <Epsilon>

<While Statement> → while <Conditional> do <Multiple Statements> whileend

<Begin Statement> → begin <Multiple Statements> end

<Conditional> → <Expression> <Conditional Prime>

<Conditional Prime> → <Relop> <Expression> | <Epsilon>

<Relop> → < | <= | == | <> | >= | >

<Expression> → <Term> <Expression Prime>

<Expression Prime> → + <Term> <Expression Prime> | - <Term> <Expression Prime> | <Epsilon>

<Expression Semicolon> --> <Expression> ;

<Term> → <Factor> <Term Prime>

<Term Prime> → * <Factor> <Term Prime> | / <Factor> <Term Prime> | <Epsilon>

<Factor> → (<Expression>) | <Identifier>

<Assign> → <Identifier> = <Expression Semicolon>

<Declarative> → <Type> <Identifier> <More IDs> ;

<More IDs> → , <Identifier> <More IDs> | <Epsilon>

<Identifier> → id

<Type> → bool | float | int

For each non-terminal value in this list of rules, a different parsing function exists. The rules associated with a parsing function are printed at the start of each function call. Each function returns a boolean value and follows a similar pattern:

1. Check if the token at the current index is equal to a predicted character.
2. If it is not, return False. If it is, check if the index can be incremented.
3. If it cannot, return False. If it can, call the next parsing function.
4. Repeat steps 1-3 until the final terminal/non-terminal value is reached. If the final value is never reached, return False. Otherwise, return True.

4. Any Limitation

All text files being analyzed must have either a white space, a new line (\n), or a tab (\t) at the end of the file. Otherwise, the very last token in the file is skipped and not analyzed by the lexical analyzer, which can cause errors.

5. Any shortcomings for each iterations

None

Testing Symbol Table

Input: ids.txt

```
int num1, num2;
float num3;
bool num4;
float num5, num6, num7;
```

Output:

```
IDENTIFIER          MEMORY LOCATION     TYPE

num1                5000                int
num2                5001                int
num3                5002                float
num4                5003                bool
num5                5004                float
num6                5005                float
num7                5006                float
```