Assignment 2 – Syntax Analysis

CPSC 323-04

Iftekharul Islam

1. Problem Statement

The purpose of this assignment is to design a syntax analyzer, the step that comes after lexical analysis when building a compiler. When performing an analysis of syntax, grammar rules need to be defined. Each rule is composed of a series of rules (non-terminal values) and terminal values. These grammar rules dictate what code is accepted or rejected by the compiler. After defining these rules, different methods of implementing them exist. I implemented a predictive recursive descent parser.

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.
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 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> ;

<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.

5. Any shortcomings for each iterations

Although error messages are implemented and printed to the console in the event of a syntax error, the existence of one error causes a chain reaction that triggers several error messages instead of just the one directly related to the syntax error. This is due to the recursive implementation and the fact that the functions are heavily dependent on one another. As a result, one mistake multiplies and becomes many.

Testing All Rules

Output of test_all.txt:

```
Token: KEYWORD  Lexeme: int
        <Statement> --> <Assign> | <Expression Semicolon> | <Declarative> |
<If Statement> | <While Statement> | <Begin Statement>
        <Expression Semicolon> --> <Expression> ;
        <Expression> --> <Term> <Expression Prime>
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Declarative> --> <Type> <Identifier> <More IDs> ;
        <Type> --> int

Token: IDENTIFIER       Lexeme: a
        <Identifier> --> id

Token: SEPARATOR        Lexeme: ,
        <More IDs> --> , <Identifier> <More IDs> | <Epsilon>

Token: IDENTIFIER       Lexeme: b
        <Identifier> --> id
```

```
Token: SEPARATOR        Lexeme: ,
        <More IDs> --> , <Identifier> <More IDs> | <Epsilon>

Token: IDENTIFIER       Lexeme: c
        <Identifier> --> id

Token: SEPARATOR        Lexeme: ;
        <More IDs> --> , <Identifier> <More IDs> | <Epsilon>

Token: IDENTIFIER       Lexeme: c
        <Statement> --> <Assign> | <Expression Semicolon> | <Declarative> |
<If Statement> | <While Statement> | <Begin Statement>
        <Identifier> --> id
        <Assign> --> <Identifier> = <Expression Semicolon>
        <Identifier> --> id

Token: OPERATOR         Lexeme: =

Token: IDENTIFIER       Lexeme: a
        <Expression> --> <Term> <Expression Prime>
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: OPERATOR         Lexeme: +
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>
        <Expression Prime> --> + <Term> <Expression Prime> | - <Term>
<Expression Prime> | <Epsilon>

Token: IDENTIFIER       Lexeme: b
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: SEPARATOR        Lexeme: ;
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>
        <Expression Prime> --> + <Term> <Expression Prime> | - <Term>
<Expression Prime> | <Epsilon>

Token: KEYWORD  Lexeme: while
        <Statement> --> <Assign> | <Expression Semicolon> | <Declarative> |
<If Statement> | <While Statement> | <Begin Statement>
        <Expression Semicolon> --> <Expression> ;
        <Expression> --> <Term> <Expression Prime>
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Declarative> --> <Type> <Identifier> <More IDs> ;
        <While Statement> --> while <Conditional> do <Multiple Statements>
whileend

Token: IDENTIFIER       Lexeme: a
        <Conditional> --> <Expression> <Conditional Prime>
        <Expression> --> <Term> <Expression Prime>
```

```
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: OPERATOR         Lexeme: >
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>
        <Expression Prime> --> + <Term> <Expression Prime> | - <Term>
<Expression Prime> | <Epsilon>
        <Conditional Prime> --> <Relop> <Expression> | <Epsilon>
        <Relop> --> < | <= | == | <> | >= | >

Token: OPERATOR         Lexeme: =

Token: IDENTIFIER       Lexeme: b
        <Expression> --> <Term> <Expression Prime>
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: KEYWORD  Lexeme: do
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>
        <Expression Prime> --> + <Term> <Expression Prime> | - <Term>
<Expression Prime> | <Epsilon>

Token: IDENTIFIER       Lexeme: b
        <Multiple Statements> --> <Statement> <More Statements>
        <Statement> --> <Assign> | <Expression Semicolon> | <Declarative> |
<If Statement> | <While Statement> | <Begin Statement>
        <Identifier> --> id
        <Assign> --> <Identifier> = <Expression Semicolon>
        <Identifier> --> id

Token: OPERATOR         Lexeme: =

Token: IDENTIFIER       Lexeme: b
        <Expression> --> <Term> <Expression Prime>
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: OPERATOR         Lexeme: /
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>

Token: IDENTIFIER       Lexeme: c
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: SEPARATOR        Lexeme: ;
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>
        <Expression Prime> --> + <Term> <Expression Prime> | - <Term>
<Expression Prime> | <Epsilon>
```

```
        <More Statements> --> ; <Statement> <More Statements> | <Epsilon>

Token: IDENTIFIER       Lexeme: c
        <Statement> --> <Assign> | <Expression Semicolon> | <Declarative> |
<If Statement> | <While Statement> | <Begin Statement>
        <Identifier> --> id
        <Assign> --> <Identifier> = <Expression Semicolon>
        <Identifier> --> id

Token: OPERATOR         Lexeme: =

Token: IDENTIFIER       Lexeme: a
        <Expression> --> <Term> <Expression Prime>
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: OPERATOR         Lexeme: *
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>

Token: IDENTIFIER       Lexeme: b
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: SEPARATOR        Lexeme: ;
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>
        <Expression Prime> --> + <Term> <Expression Prime> | - <Term>
<Expression Prime> | <Epsilon>

Token: KEYWORD  Lexeme: whileend
        <More Statements> --> ; <Statement> <More Statements> | <Epsilon>

Token: KEYWORD  Lexeme: if
        <Statement> --> <Assign> | <Expression Semicolon> | <Declarative> |
<If Statement> | <While Statement> | <Begin Statement>
        <Expression Semicolon> --> <Expression> ;
        <Expression> --> <Term> <Expression Prime>
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Declarative> --> <Type> <Identifier> <More IDs> ;
        <If Statement> --> if <Conditional> then <Multiple Statements>
<Else Statement> endif

Token: IDENTIFIER       Lexeme: a
        <Conditional> --> <Expression> <Conditional Prime>
        <Expression> --> <Term> <Expression Prime>
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: OPERATOR         Lexeme: <
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>
```

```
        <Expression Prime> --> + <Term> <Expression Prime> | - <Term>
<Expression Prime> | <Epsilon>
        <Conditional Prime> --> <Relop> <Expression> | <Epsilon>
        <Relop> --> < | <= | == | <> | >= | >

Token: IDENTIFIER       Lexeme: c
        <Expression> --> <Term> <Expression Prime>
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: KEYWORD  Lexeme: then
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>
        <Expression Prime> --> + <Term> <Expression Prime> | - <Term>
<Expression Prime> | <Epsilon>

Token: IDENTIFIER       Lexeme: a
        <Multiple Statements> --> <Statement> <More Statements>
        <Statement> --> <Assign> | <Expression Semicolon> | <Declarative> |
<If Statement> | <While Statement> | <Begin Statement>
        <Identifier> --> id
        <Assign> --> <Identifier> = <Expression Semicolon>
        <Identifier> --> id

Token: OPERATOR         Lexeme: =

Token: IDENTIFIER       Lexeme: a
        <Expression> --> <Term> <Expression Prime>
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: OPERATOR         Lexeme: +
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>
        <Expression Prime> --> + <Term> <Expression Prime> | - <Term>
<Expression Prime> | <Epsilon>

Token: IDENTIFIER       Lexeme: b
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: SEPARATOR        Lexeme: ;
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>
        <Expression Prime> --> + <Term> <Expression Prime> | - <Term>
<Expression Prime> | <Epsilon>
        <More Statements> --> ; <Statement> <More Statements> | <Epsilon>

Token: KEYWORD  Lexeme: else
        <Else Statement> --> else <Multiple Statements> | <Epsilon>

Token: IDENTIFIER       Lexeme: a
```

```
        <Multiple Statements> --> <Statement> <More Statements>
        <Statement> --> <Assign> | <Expression Semicolon> | <Declarative> |
<If Statement> | <While Statement> | <Begin Statement>
        <Identifier> --> id
        <Assign> --> <Identifier> = <Expression Semicolon>
        <Identifier> --> id

Token: OPERATOR         Lexeme: =

Token: IDENTIFIER       Lexeme: c
        <Expression> --> <Term> <Expression Prime>
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: OPERATOR         Lexeme: -
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>
        <Expression Prime> --> + <Term> <Expression Prime> | - <Term>
<Expression Prime> | <Epsilon>

Token: IDENTIFIER       Lexeme: b
        <Term> --> <Factor> <Term Prime>
        <Factor> --> ( <Expression> ) | <Identifier>
        <Identifier> --> id

Token: SEPARATOR        Lexeme: ;
        <Term Prime> --> * <Factor> <Term Prime> | / <Factor> <Term Prime>
| <Epsilon>
        <Expression Prime> --> + <Term> <Expression Prime> | - <Term>
<Expression Prime> | <Epsilon>
        <More Statements> --> ; <Statement> <More Statements> | <Epsilon>

Token: KEYWORD  Lexeme: endif
```