CS4201 Practical 1 Report

matriculation number: 160021429

1. Overview

The aim of this practical is to implement a parser that parses the Oreo source code by doing the lexical and syntactical analysis. The main task is to implement the lexer for lexical analysis, and syntax parser for syntactical analysis.

1 - 1. Instruction

Basically, I used Java to implement the parser.

To compile to code, you need to type “javac *.java”.

When you run the parser, there will 2 options. You could use either standard input stream or file input stream to input the Oreo source code. To run the program with standard input stream, use “java Parser” to run the parser. Otherwise, you need to add a command line argument, where the first command line argument should be the file path of the Oreo source code file.

- Standard Input : java Parser
- File Input : java Parser <FILE_PATH>

If the first command line argument is not a valid file path of the Oreo source code, then the program will throw the IOException.

When you use the program with standard input, if the program not prints out the AST even if you write all source codes, please type enter. My lexer reads a line until the source code ends, so you will need to press enter after writing the oreo code.

2. Design and Implementation

2 - 1. Lexer

To implement the lexer, I used Finite State Automata (FSA). So, when the program reads the input, the FSA will check the word, and change the state. The main reason that I used FSA for designing the lexer is because the FSA starts from the initial state, and change the state by processing the input. This means that we could check if the source code starts with the suitable keyword, and we could guarantee that the sequence of codes by changing the state of FSA.

Basically, what my Lexer does is read a line via input stream, split the line to words, and pass each word to FSA. Then, the lexer will change the state and wait for next input until the program reads all input lines.

Basically, I made a class for each state of FSA, so that the FSA could parse the word with suitable object. There are 9 classes for FSA states - AssignmentState, CompoundState, FunctionState, IfState, PrintState, ProgramState, StatementState, VariableState, and WhileState. Each class parses and checks the basic syntax of the corresponding statements. For example, if the current state is a PrintState, then the FSA will only look for the print statement, and print out the error if the read line is not related to the print statement. And all these classes are children of LexerFSA java interface.

Since I used the FSA to implement the lexer, my lexer could check some basic syntax. The lexer will print out the suitable error message, and skip that line. For example, if the source code contains the “var a := 3;;”, the lexer will print out the error message "SyntaxError::Too many semicolons", and skip that variable statement.

The SymbolTable class contains the codes for symbol table. My lexer uses the symbol table to store the variable and functions. So, the lexer will use the symbol table to check if the given name of function or variable is declared. Furthermore, if the source code contains some function call, then the lexer will use the symbol table to check the number of arguments that the given name of the function requires. If the actual number of arguments that are used for function call is not equal to the expected number, then the lexer will print out suitable error message.

The ExpressionUtils class contains some static methods that helps the lexer to parse the expressions. For instance, if the source code contains the expression “1 + 2 - (3 * 5)”, then the lexer will use the static methods of ExpressionUtils class to generate stream of tokens “CONST_NUM 1, AROP +, CONST_NUM 2, AROP -, LPAREN, CONST_NUM 3, AROP *, CONST_NUM 5, RPAREN”. When the ExpressionUtils class converts the expressions to tokens, it validates the name of variables and function calls by using the symbol table. (Please see the Appendix section to check what each token name means)

The Lexemes class contains the list of SymbolToken objects, where each SymbolToken object contains the generated lexeme token.

2 - 2. Syntax Analyser

If you see the SyntaxParser class, you could see the methods called parse, which returns the AbstractSyntaxTreeNode object. This method reads the stream of lexeme tokens, and generate the syntax tree by using the AbstractSyntaxTreeNode objects, and return the root node of the tree.

Basically, the SyntaxParser parses the tokens by using the LL(1) parser. It reads the first token, and execute the corresponding method to generate the suitable sub tree. The parse method will keep iterate the list of lexeme tokens until the SyntaxParser generates all nodes for the syntax tree.

While implementing the LL(1) parser, I realised that it would be necessary to add a custom precedence rules for parentheses and not operator. As you know the expression in the parentheses should be executed before than the other expressions. To overcome this problem, I call the parsing method recursively when the current terminal is a left parenthesis. So, my parser will parse the expression in the parentheses first, and then parse the other expressions, just like shunting yard algorithm. Moreover, the if we don't add a precedence rule for the not operator, then some boolean operations which looks not ambiguous might be ambiguous. For example, "not true or true" can be interpreted to "(not true) or true" or "not (true or true)". This means that if we don't add a precedence rule, then the parser will not be able to parse the boolean expression like "not true or true". So, I gave higher precedence to the not operator, so that the parse parses the "not true or true" to "(not true) or true".

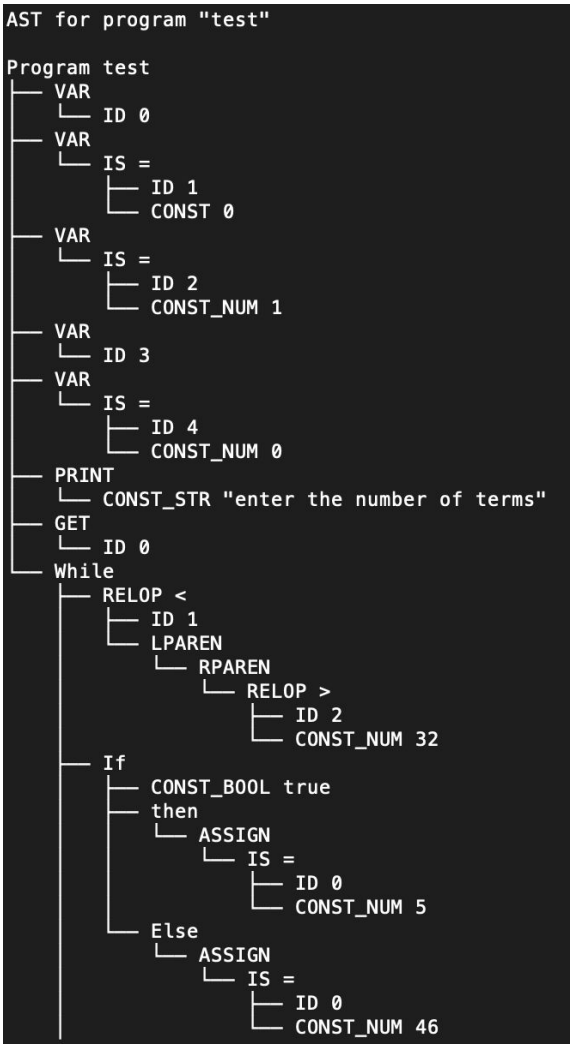
Furthermore, when the SyntaxParser generates the syntax tree, if the program founds the syntax error in the specific statement, then the program will print out the error message.

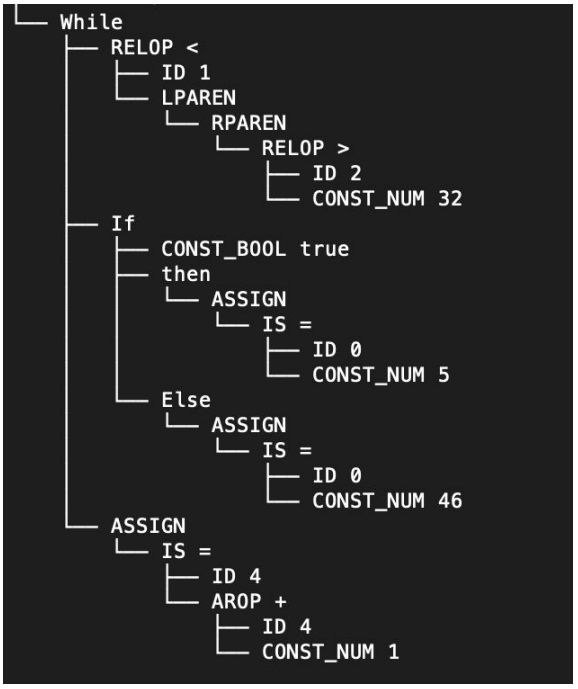
When the SyntaxParser finishes generating the syntax tree, the program will print out the syntax tree via standard output stream. The format of the syntax tree is similar to the result of "tree" command in linux.

3. Testing

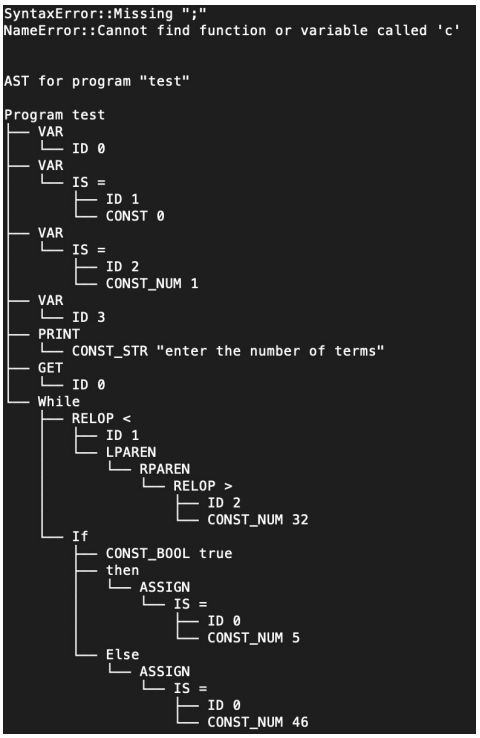
To test my program, I used the test cases that are uploaded on the studres. Below are the outputs of each test case.

1) test.oreo



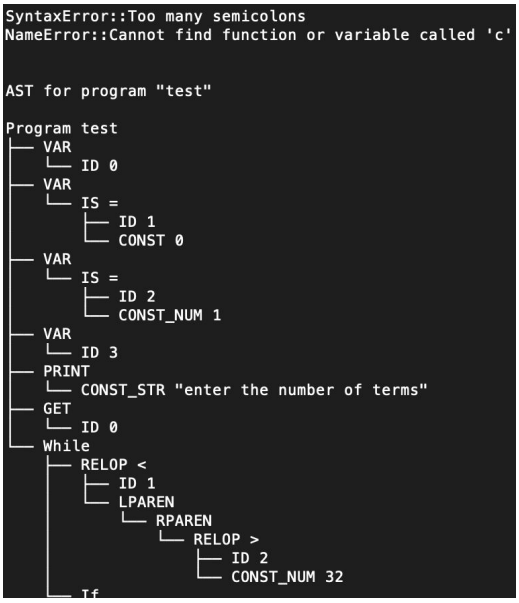


2) test2.oreo



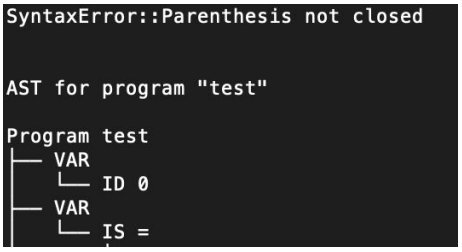
```
SyntaxError::Missing ";"
NameError::Cannot find function or variable called 'c'


AST for program "test"

Program test
├── VAR
│   └── ID 0
├── VAR
│   └── IS =
│       ├── ID 1
│       └── CONST 0
├── VAR
│   └── IS =
│       ├── ID 2
│       └── CONST_NUM 1
├── VAR
│   └── ID 3
├── PRINT
│   └── CONST_STR "enter the number of terms"
├── GET
│   └── ID 0
└── While
    ├── RELOP <
    │   ├── ID 1
    │   └── LPAREN
    │       └── RPAREN
    │           └── RELOP >
    │               ├── ID 2
    │               └── CONST_NUM 32
    └── If
        ├── CONST_BOOL true
        ├── then
        │   └── ASSIGN
        │       └── IS =
        │           ├── ID 0
        │           └── CONST_NUM 5
        └── Else
            └── ASSIGN
                └── IS =
                    ├── ID 0
                    └── CONST_NUM 46
```

3) test3.oreo



```
SyntaxError::Too many semicolons
NameError::Cannot find function or variable called 'c'


AST for program "test"

Program test
├── VAR
│   └── ID 0
├── VAR
│   └── IS =
│       ├── ID 1
│       └── CONST 0
├── VAR
│   └── IS =
│       ├── ID 2
│       └── CONST_NUM 1
├── VAR
│   └── ID 3
├── PRINT
│   └── CONST_STR "enter the number of terms"
├── GET
│   └── ID 0
└── While
    ├── RELOP <
    │   ├── ID 1
    │   └── LPAREN
    │       └── RPAREN
    │           └── RELOP >
    │               ├── ID 2
    │               └── CONST_NUM 32
    └── If
```

4) test4.oreo



```
SyntaxError::Parenthesis not closed


AST for program "test"

Program test
├── VAR
│   └── ID 0
├── VAR
│   └── IS =
```

5) test5.oreo

```
{- missing end -}
program test5

begin

        var n;
        var first := 0;
        var second :=1;
        var next;
        var c :=0;

    print "enter the number of terms";

    get n;

    while ( first < (second > 32)  )
    begin

       if ( true ) then begin n := 5;  end
                           else begin n := 46; end;


    c := c + 1;

end

SyntaxError::Invalid number of "end"!
```

The test5.oreo is a test case to check if the parser could handle the syntax error of “missing end” issue. In this case, if you use the standard input to input the source code, you need to press “ctrl + d” to let the program know that the input stream ends. Otherwise, the program will keep wait for the end of the file.

6) test6.oreo

```
AST for program "test6"

Program test6
├── VAR
│   └── ID 0
├── VAR
│   └── IS =
│       ├── ID 1
│       └── CONST 0
├── VAR
│   └── IS =
│       ├── ID 2
│       └── CONST_NUM 1
├── VAR
│   └── ID 3
├── VAR
│   └── IS =
│       ├── ID 4
│       └── CONST_NUM 0
├── PRINT
│   └── CONST_STR "enter the number of terms"
├── GET
│   └── ID 0
└── While
    ├── RELOP <
    │   ├── ID 1
    │   └── LPAREN
    │       └── RPAREN
    │           └── RELOP >
    │               ├── ID 2
    │               └── CONST_NUM 32
    ├── If
    │   ├── LOGOP or
    │   │   ├── LOGOP not
    │   │   │   └── CONST_BOOL false
    │   │   └── CONST_BOOL true
    │   ├── then
    │   │   └── ASSIGN
    │   │       └── IS =
    │   │           ├── ID 0
```

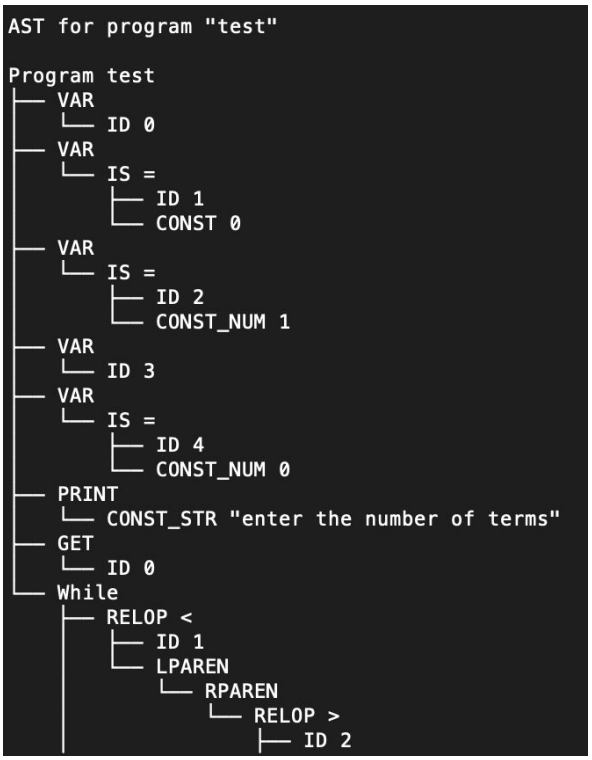
7) test7.oreo

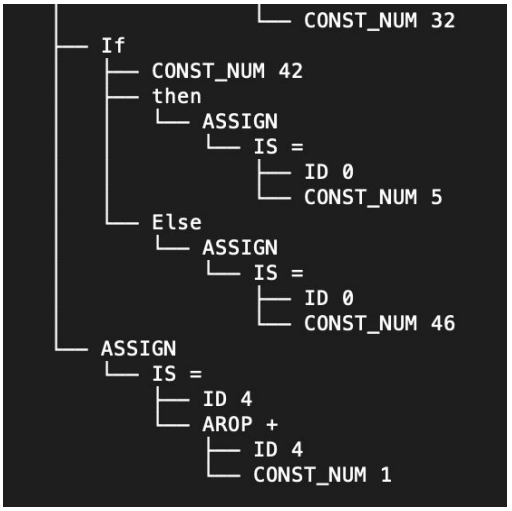
```
SyntaxError::Invalid number of double quotation mark - expected = 2 actual = 1


AST for program "test"

Program test
├── VAR
│   └── ID 0
```

8) test8.oreo

```
AST for program "test"

Program test
├── VAR
│   └── ID 0
├── VAR
│   └── IS =
│       ├── ID 1
│       └── CONST 0
├── VAR
│   └── IS =
│       ├── ID 2
│       └── CONST_NUM 1
├── VAR
│   └── ID 3
├── VAR
│   └── IS =
│       ├── ID 4
│       └── CONST_NUM 0
├── PRINT
│   └── CONST_STR "enter the number of terms"
├── GET
│   └── ID 0
└── While
    ├── RELOP <
    │   ├── ID 1
    │   └── LPAREN
    │       └── RPAREN
    │           └── RELOP >
    │               ├── ID 2
    │               └── CONST_NUM 32
    ├── If
    │   ├── RELOP <
    │   │   ├── CONST_NUM 42
    │   │   └── CONST_NUM 100
    │   └── then
```

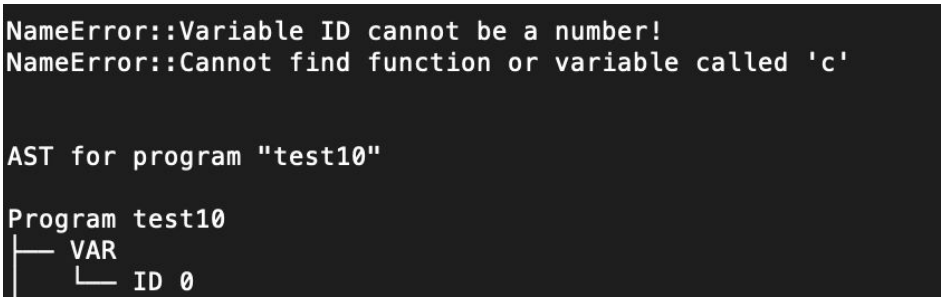
9) test9.oreo





If you see the test9.oreo file, the expression of the if statement is a number, not a boolean expression. However, type checking is not the thing that we need to implement for this practical. Thus, my program does not print out any error message, and generate a normal syntax tree, as you could see above.

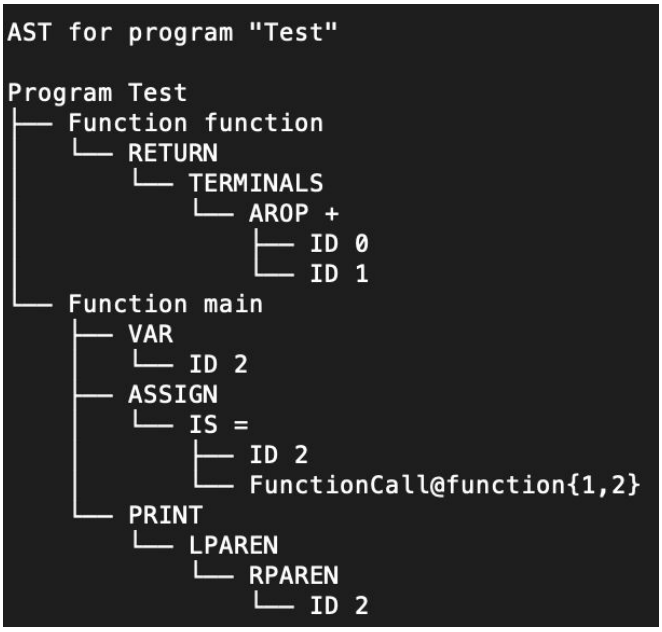
10) test10.oreo



As you could see above, my program could handle all test cases, and print out suitable error messages and generated syntax tree.

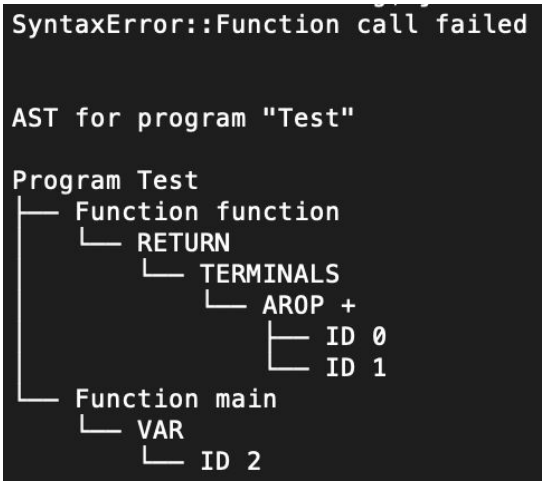
None of the test cases contain codes for procedure statement, which generates the function, I made several test cases to test if my program could handle the errors about procedure statement.

In the tests directory, I added test files to test the procedure statements. The procedure1.oreo file contains the valid codes, thus, the program will print out the syntax tree without printing the error message, as you could see below.



If you see the screenshot above, you could find a token called “FunctionCall@function{1,2}”. This token means that the program calls the function, whose name is function, with 2 arguments 1 and 2.

In the procedure2.oreo file, you could see that the program calls the function “function” with one arguement, where the procedure “function” requires 2 arguments. If you run the program with the procedure2.oreo file, then the program will print out the error message as below.



The procedure3.oreo file contains the source code where the procedure statement does not have parenthesis and arguments.

```
procedure function
begin
    return x + y;
```

Furthermore, you could see that the procedure main calls the function “function” just like below.

```
result := function(1, 2);
```

Henceforth, if you run the program with this file, then the parser will print out the suitable error messages.

```
SyntaxError::Cannot find ")" for function!
NameError::Cannot find function or variable called function
```

The first error message shows that the function does not have the parenthesis. And the second error message says the parser failed to find the function or variable called “function”. As you could see, my parser could actually handle the procedures, and print out suitable messages for syntax errors in the procedure.

Moreover, I also made a test file called nestedWhile.oreo, which contains the nested while statement. This file contains the nested while statements.

```
AST for program "Test"

Program Test
└── While
    ├── RELOP <
    │   ├── CONST_NUM 1
    │   └── CONST_NUM 2
    └── While
        ├── RELOP <
        │   ├── CONST_NUM 3
        │   └── CONST_NUM 4
        └── While
            ├── RELOP <
            │   ├── CONST_NUM 5
            │   └── CONST_NUM 6
            └── While
                ├── RELOP <
                │   ├── CONST_NUM 7
                │   └── CONST_NUM 8
                └── VAR
                    └── IS =
                        ├── ID 0
                        └── CONST 1
```

As you could see above, my parser could parse the nested while statements properly.

4. Appendix

a. Constants

CONST : constant - can be one of number, boolean and string
i.e. CONST 5, CONST true, CONST “Hello”
CONST_NUM : number (integer or decimal number)
i.e. CONST_NUM 5, CONST_NUM 4.24
CONST_BOOL : boolean (either true or false)
i.e. CONST_BOOL true, CONST_BOOL false
CONST_STR : string
i.e. CONST_STR “hello world”

b. Variable

ID : id of the variable
i.e. ID 0 (variable whose index in the symbol table is 0)

c. Operator

IS : “=” operator, that assigns the value to the variable.
i.e. IS “=”
AROP : arithmetic operators (+, -, *, /)
i.e. AROP +, AROP -, AROP *, AROP /
RELOP : relational operators (==, <, <=, >, >=)
i.e. RELOP ==, RELOP <, RELOP <=, RELOP >, RELOP >=
LOGOP : logical operators (and, or, not)
i.e. LOGOP and, LOGOP or, LOGOP not

d. Statements

VAR : variable statement
GET : get statement
PRINT : print statement
PRINTLN : println statement
ASSIGN : assign statement
While: while statement
If : if statement
then : then of the if statement
Else : else statement
Function : procedure statement

FunctionCall@function_name{[args]} : function call
i.e. FunctionCall@f1{“hello”, “world”} = f1(“hello”, “world”);