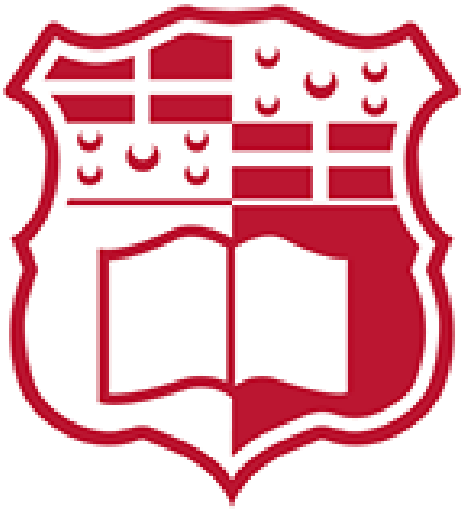
CPS2000

Compiler Theory and Practice

Assignment (Part 1) 2019/2020

Documentation



Daniel James Sumler, 0107297A

Bsc. Computing Science, Yr. 2

Table of Contents

Limitations and Bugs

In this part of the documentation, I will be going over bugs and limitations which are present in my compiler.

I was not able to get inline comments to successfully work without having parser errors, so they have been removed entirely from the program to avoid errors.

SubExpressions may not execute, as there was a problem during the recursion process when calculating expressions in the interpreter. Many attempts and solutions were tried, but ultimately to no avail.

Function Calls do not return values in the interpreter, and parameters cannot be passed to functions. Many different attempts were tried, but unfortunately, I could not find a way to make them execute. Functions without parameters which do some other operation (such as print a value) DO work. Nevertheless, my attempt on executing the function calls remains in the Interpreter class.

Some bugs arose in the development of the Unary type, and therefore these were removed from the final program to avoid any errors.

A known bug does not allow the multiplication of variables sometimes. It is not known how to fix it, but a quick workaround can be to use a third variable to compute the arithmetic, as seen in sample2.txt.

Floats and integers CANNOT be combined. Floats MUST contain a "." Character in them (e.g. 5.0).

Everything in the compiler has been tested multiple times, and for the most part, it is highly functional.

Task 1 – Table-driven lexer

In this task I was required to create an implementation of a table-driven lexer such that the given EBNF rules are follows, and lexical errors are reported.

My implementation starts by declaring custom enumeration classes in order to declare the possible states and tokens that the lexer will make use of. These were used as it was easier to create a transition table and easier to avoid any dead states, so the program will never crash, no matter what symbol is fed into it. These include the LexerStates, TKN and TOKEN classes.

The transition table

The transition table itself was made by first planning out the possible transitions, which involved planning each possible transition from every state. The tokens which are processed using the transition table are detailed below:

Letter

$$\langle Letter \rangle \quad ::= \text{ [A-Za-z]}$$

The **Letter** EBNF is utilized whenever the next character is a letter between A and Z. This is implemented by checking the character using the built-in **isLetter** function in Java.

Digit

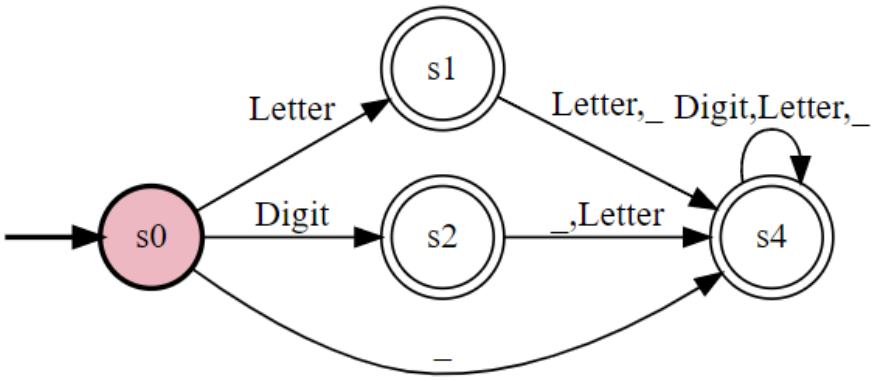
$$\langle Digit \rangle \quad ::= \text{ [0-9]}$$

The **Digit** EBNF is utilized whenever the character currently being processed is a digit between 0 and 9. This is implemented by checking the character using the built-in **isDigit** function in Java.

Identifier

$$\langle Identifier \rangle \quad ::= \text{ ('\_' | } \langle Letter \rangle \text{) \{ '\_' | } \langle Letter \rangle \text{ | } \langle Digit \rangle \text{ \}}$$

The **Identifier** EBNF is utilized whenever a series of **Letters** starts with a "_" character, a String contains a combination of **Letters** and **Digits**, or whenever a sequence of **Letters** is entered. The transitions of states in respect to an **Identifier** occur with the help of the transition table, which was previously declared. A DFSA of these possible transitions can be seen below:



Type

$$\langle Type \rangle \quad ::= \text{`float'} \mid \text{`int'} \mid \text{`bool'}$$

The **Type** EBNF is utilized whenever the inputted sequence of characters is identical to “float”, “int” or “bool”. This is implemented by processing every **Identifier**. If an **Identifier** is seen to be one of these keywords, the **Type** token is returned by the lexer. This comparison is done in the **SymbolTable** function, as seen below:

```
if(State == LexerStates.S4) {
    switch (lexeme) {
        case "float":
        case "bool":
        case "int":
            return TKN.Type;
```

Auto

$$\langle Auto \rangle \quad ::= \text{`auto'}$$

The **Auto** EBNF is utilized in exactly the same way the **Type** EBNF is. Each **Identifier** is compared in the **SymbolTable** function, and if found to be equal to the String “auto”, the **Auto** token is returned. The part of the function used to make this comparison can be seen below:

```
if(State == LexerStates.S4) {
    switch (lexeme) {
        case "auto":
            return TKN.Auto;
```

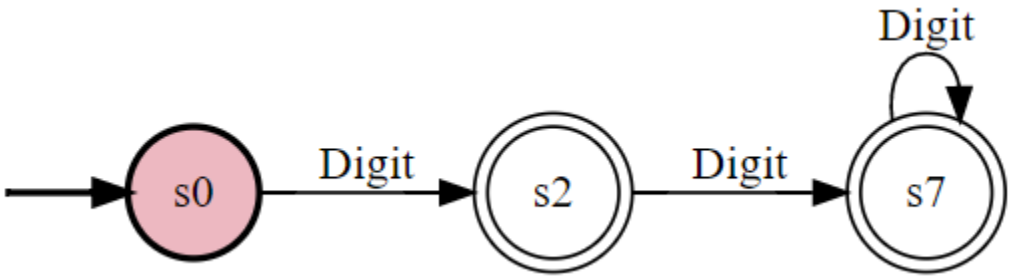
BooleanLiteral

The **BooleanLiteral** EBNF is implemented in the same way as the **Auto** and **Type** EBNF. Each **Identifier** is compared in the **SymbolTable()** function, and if the String matches “true” or “false”, the **BooleanLiteral** token is returned, as seen below:

```
if(State == LexerStates.S4) {
    switch (lexeme) {
        case "true":
        case "false":
            return TKN.BooleanLiteral;
```

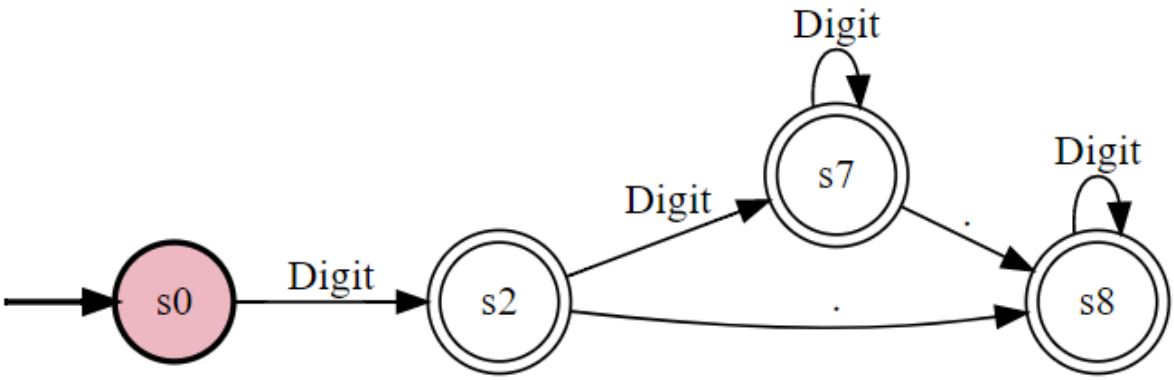
IntegerLiteral

The **IntegerLiteral** EBNF is utilized when a String is entered, exclusively containing more than one **Digit**. A single **Digit** is not considered an **IntegerLiteral** in my implementation, and instead is processed as a singular **Digit**. This is done in order to separate the singular **Digits** from the **Integers**. Both of these tokens are treated the same way in the parser, but are processed like this in order to make the AST more accurate. An Automata on how an **IntegerLiteral** is transitioned to can be seen below:



FloatLiteral

The **FloatLiteral** EBNF is utilized similarly to the **IntegerLiteral** EBNF, with the added difference that a **FloatLiteral** must contain at most one "." Character. This character must be entered after a **Digit** or an **IntegerLiteral** and not as the start of a String, else an error State will be invoked. At least a single **Digit** must also be present after the character. If more than one "." Character is found in the String, the program will again be put in an error state. These transitions occur by navigating the **Transition Table** array. The described transition can be seen in a simplified diagram below:



MultiplicativeOp

In order to implement the **MultiplicativeOp** EBNF, my lexer constantly checks whether the current or next character in the input program is equal to "*" or "/". If it is seen that the current character is equal to one of these two, the **MultiplicativeOp** token is returned immediately, regardless of whether a whitespace has been reached or not. If, for example, an **Identifier** is currently being processed and the program sees that the *next* symbol is one of a **MultiplicativeOp**, then the lexer will immediately return the **Identifier** token, and move on to the **MultiplicativeOp** symbol when the next token is requested. The way this is done can be seen below:

```
//if the current character is its own token, meaning for example, a
// ; or , symbol, it should be returned immediately as its own
//separate token.
if (returnChar(this.inputString.charAt(charLocation))) {
    charLocation++;
    return tok;
}
```

The *returnChar* function checks the current character, and if it matches any of the reserved symbols in the language, true is returned, else false.

Below, one can see the *returnChar* function:

```
static boolean returnChar(char ch){
    //if true is returned, it means that a special character was encountered that must be returned immediately
    if(ch == '(' || ch == ')' || ch == ',' || ch == '{' || ch == '}' || ch == '+' ||
    ch == '-' || ch == '*' || ch == '/' || ch == ' ' || ch == ':' || ch == ';' ||
            ch == '=' || ch =='<' || ch == '>'){
        return true;
    } else{
        return false;
    }
}
```

My lexer is designed this way in order to make it possible for a user to create an **Expression** without the use of spaces, for example, “id=id*5;” and “id = id * 5;” will both be processed in the exact same way. This makes the language more useable and less strict in terms of how the language’s syntax operates.

In order to process the “and” part of the **MultiplicativeOp** EBNF, the same *SymbolTable* function as used above is utilized. This returns the **MultiplicativeOp** token when this particular word is found, as seen below:

```
static TKN SymbolTable(String lexeme, LexerStates State){
    //function to determine the current token given the current processed string and
    //the current state
    if(State == LexerStates.S4) {
        switch (lexeme) {
            case "and":
                return TKN.MultiplicativeOp;
```

AdditiveOp

The **AdditiveOp** EBNF is implemented in the exact same way the **MultiplicativeOp** is implemented, with the added difference that the *SymbolTable* recognizes the word “or” as an **AdditiveOp** token, as seen below:

```
static TKN SymbolTable(String lexeme, LexerStates State){
    //function to determine the current token given the current processed string and
    //the current state
    if(State == LexerStates.S4) {
        switch (lexeme) {
            case "or":
                return TKN.AdditiveOp;
```

RelationalOp

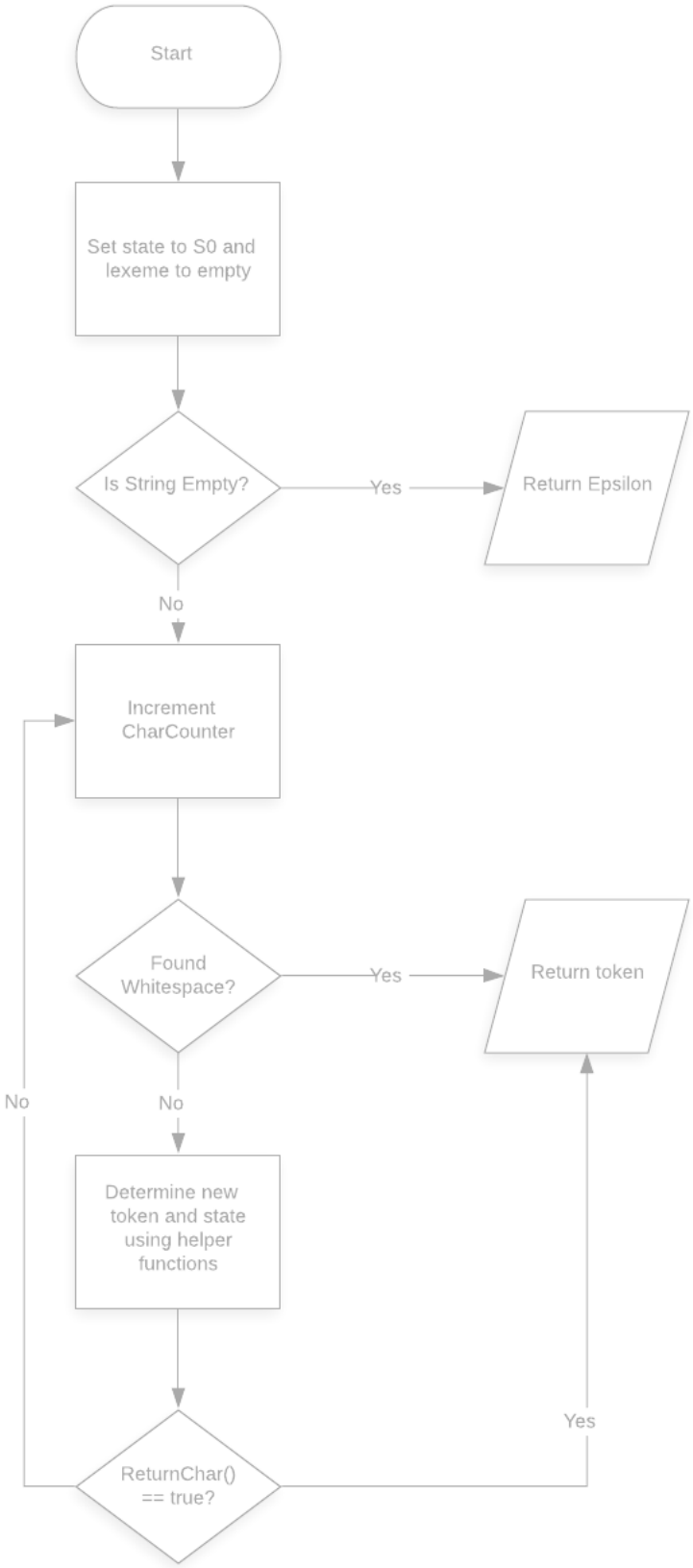
The **RelationalOp** EBNF is implemented in the exact same was as the **MultiplicativeOp** and **AdditiveOp** EBNF, with the added difference that there are no reserved words in this EBNF, and therefore only a relational symbol will activate the **RelationalOp** token.

The NextWord function

The **NextWord()** function is the main function in the lexer. Once this is called, the lexer will start operating by examining the given string one character at a time. The first thing it will do is check whether the String is empty or not, and if it is, then it should return an **Epsilon** token. This symbolizes the end of a String. Next, the lexer's state is set as the start state (**S0**) and the character processing loop is entered. While in the loop, the program iterates to the next character, while also calculating the next lexer state based off the current state and the new token (meaning the newly encountered character). This is done by traversing the transition table and returning said state. This loop will continue until a *whitespace* or *special symbol* is encountered.

A *special symbol* is a symbol which can be defined as a reserved symbol. For example, **( ) , { } + - * / : ; = >** are all special symbols and the lexer should return these as their own token. The way this is done is by looking ahead to the next character, and if it is a special symbol, to return the current token immediately. The lexer returns a custom class, **TOKEN** which contains a String value (to store the processed lexeme) and a token (which is defined in the **TKN** class).

A very simple explanation of how this function works can be found on the next page in the form of a Flowchart diagram.


Start
Set state to S0 and lexeme to empty
Is String Empty?
Yes
Return Epsilon
No
Increment CharCounter
Found Whitespace?
Yes
Return token
No
No
Determine new token and state using helper functions
ReturnChar() == true?
Yes

Testing the Lexer

In this part of the assignment, I will be creating a temporary main class in order to test the lexer. This main class will create an instance of the lexer class and allow me to enter any number of symbols I want and will continue to output these symbols until the end of the String is reached (**Epsilon**). This main class can be seen below:

```
public static void main(String[] args){
    System.out.println("Syntax Checker:");
    Scanner scn = new Scanner(System.in);
    String str = scn.nextLine();
    Lexer l = new Lexer(str);
    TOKEN token;
    token = l.NextWord();
    while(token.getToken() != TKN.EPSILON){
        System.out.println("Token = " +token.getToken()+ " | String = "+token.getString());
        token = l.NextWord();
    }

}
```

I will be testing each possible EBNF and comparing every test result with an expected output result.

Test 1 – Letter

Input: The letter ‘a’ will be inputted into the lexer.

Expected Output: The lexer is expected to output the Token **Letter**.

```
Syntax Checker:
a
Token = Letter | String = a
```

Result: The actual output matched the expected output and the test passed.

Test 2 – Digit

Input: The Digit '2' will be inputted into the lexer.

Expected Output: The lexer is expected to output the Token **Digit**.

```
Syntax Checker:
2
Token = Digit | String = 2
```

Result: The actual output matched the expected output and the test passed.

Test 3 – Identifier

Input: The non-reserved String 'HelloDrSpina' will be inputted into the lexer.

Expected Output: The lexer is expected to output the Token **Identifier**.

```
Syntax Checker:
HelloDrSpina
Token = Identifier | String = HelloDrSpina
```

Result: The actual output matched the expected output and the test passed.

Test 4 – Type

Input: The reserved Strings 'int', 'float' and 'bool' will be inputted into the lexer.

Expected Output: The lexer is expected to output the Token **Type** for every String.

```
Syntax Checker:
int float bool
Token = Type | String = int
Token = Type | String = float
Token = Type | String = bool
```

Result: The actual output matched the expected output and the test passed.

Test 5 – Auto

Input: The reserved String ‘auto’ will be inputted into the lexer.

Expected Output: The lexer is expected to output the Token **Auto**.

```
Syntax Checker:
auto
Token = Auto | String = auto
```

Result: The actual output matched the expected output and the test passed.

Test 6 – BooleanLiteral

Input: The reserved Strings ‘true’ and ‘false’ will be inputted into the lexer.

Expected Output: The lexer is expected to output the Token **BooleanLiteral** for both Strings.

```
Syntax Checker:
true false
Token = BooleanLiteral | String = true
Token = BooleanLiteral | String = false
```

Result: The actual output matched the expected output and the test passed.

Test 7 – IntegerLiteral

Input: The String ‘1234’ will be inputted into the lexer.

Expected Output: The lexer is expected to output the Token **IntegerLiteral**.

```
Syntax Checker:
1234
Token = IntegerLiteral | String = 1234
```

Result: The actual output matched the expected output and the test passed.

Test 8 – FloatLiteral

Input: The String '1234.5678' will be inputted into the lexer.

Expected Output: The lexer is expected to output the Token **FloatLiteral**.

```
Syntax Checker:
1234.5678
Token = FloatLiteral | String = 1234.5678
```

Result: The actual output matched the expected output and the test passed.

Test 9 – MultiplicativeOp

Input: The String '*/and' will be inputted into the lexer.

Expected Output: The lexer is expected to output the Token **MultiplicativeOp** for every symbol in the String.

```
Syntax Checker:
*/and
Token = MultiplicativeOp | String = *
Token = MultiplicativeOp | String = /
Token = MultiplicativeOp | String = and
```

Result: The actual output matched the expected output and the test passed.

Test 10 – AdditiveOp

Input: The String '+-or' will be inputted into the lexer.

Expected Output: The lexer is expected to output the Token **AdditiveOp** for every symbol in the String.

```
Syntax Checker:
+-or
Token = AdditiveOp | String = +
Token = AdditiveOp | String = -
Token = AdditiveOp | String = or
```

Result: The actual output matched the expected output and the test passed.

Test 11 – RelationalOp

Input: The String '<>=' will be inputted into the lexer.

Expected Output: The lexer is expected to output the Token **RelationalOp** for every symbol in the String.

```
Syntax Checker:
<>=
Token = RelationalOp | String = <
Token = RelationalOp | String = >
Token = RelationalOp | String = =
```

Result: The actual output matched the expected output and the test passed.

Test 12 – Reserved Keywords

This test is being conducted on select statement reserved statement keywords, namely ‘let’, ‘print’, ‘if’, ‘while’, ‘ff’, ‘return’ and ‘for’. These are purposely separated from normal **Identifiers** due to their important use when declaring statements.

Input: The String ‘let print if while ff return for id’ will be inputted into the lexer.

Expected Output: The lexer is expected to output the Token **RESERVED_KEYWORD** for every symbol in the String except the last one, which is an **Identifier**.

```
Syntax Checker:
let print if while ff return for id
Token = RESERVED_KEYWORD | String = let
Token = RESERVED_KEYWORD | String = print
Token = RESERVED_KEYWORD | String = if
Token = RESERVED_KEYWORD | String = while
Token = RESERVED_KEYWORD | String = ff
Token = RESERVED_KEYWORD | String = return
Token = RESERVED_KEYWORD | String = for
Token = Identifier | String = id
```

Result: The actual output matched the expected output and the test passed.

Test 13 – Statements containing specific symbols

This test is being conducted in order to test whether the lexer can properly identify each individual symbol in a statement sufficiently.

Input: The String 'if(5 > 3){ print hello; }' will be inputted into the lexer.

Expected Output: The lexer is expected to output the following Tokens in order:

RESERVED_KEYWORD, OPN_BRACKET, Digit, RelationalOp, Digit, CLOSE_BRACKET, OPN_CURLY, RESERVED_KEYWORD, Identifier, Delimeter, CLOSE_CURLY

```
Syntax Checker:
if(5 > 3){ print hello; }
Token = RESERVED_KEYWORD | String = if
Token = OPN_BRACKET | String = (
Token = Digit | String = 5
Token = RelationalOp | String = >
Token = Digit | String = 3
Token = CLOSE_BRACKET | String = )
Token = OPN_CURLY | String = {
Token = RESERVED_KEYWORD | String = print
Token = Identifier | String = hello
Token = Delimeter | String = ;
Token = CLOSE_CURLY | String = }
```

Result: The actual output matched the expected output and the test passed.

Test 14 – Statements containing specific symbols

This test is being conducted in order to test whether the lexer can properly identify each individual symbol in a statement sufficiently.

Input: The String 'for(let i : int = 0; i > 5; i = i + 1){ print i; }' will be inputted into the lexer.

Expected Output: The lexer is expected to output the following Tokens in order:

RESERVED_KEYWORD, OPN_BRACKET, RESERVED_KEYWORD, Letter, Other, Type, RelationalOp, Digit, Delimeter, Letter, RelationalOp, Delimeter, Letter, RelationalOp, Letter, AdditiveOp, Digit, CLOSE_BRACKET, OPN_CURLY, RESERVED_KEYWORD, Letter, Delimeter, CLOSE_CURLY

```
Syntax Checker:
for(let i : int = 0; i > 5; i = i + 1){ print i; }
Token = RESERVED_KEYWORD | String = for
Token = OPN_BRACKET | String = (
Token = RESERVED_KEYWORD | String = let
Token = Letter | String = i
Token = Other | String = :
Token = Type | String = int
Token = RelationalOp | String = =
Token = Digit | String = 0
Token = Delimeter | String = ;
Token = Letter | String = i
Token = RelationalOp | String = >
Token = Digit | String = 5
Token = Delimeter | String = ;
Token = Letter | String = i
Token = RelationalOp | String = =
Token = Letter | String = i
Token = AdditiveOp | String = +
Token = Digit | String = 1
Token = CLOSE_BRACKET | String = )
Token = OPN_CURLY | String = {
Token = RESERVED_KEYWORD | String = print
Token = Letter | String = i
Token = Delimeter | String = ;
Token = CLOSE_CURLY | String = }
```

Result: The actual output matched the expected output and the test passed.

Test 15 – Testing Invalid Symbols

This test is being conducted in order to test whether the lexer can properly identify symbols which are lexically incorrect (meaning they do not exist in the SmallLang language). Invalid symbols are passed on to the parser, where the error is reported. The decision to make the error reporting work this way was to keep the program as neat as possible, and only throw Exceptions in the parser.

Input: The String 'print $;' will be inputted into the lexer.

Expected Output: The lexer is expected to output the following Tokens in order:

RESERVED_KEYWORD, Invalid, Delimeter

```
Syntax Checker:
print $;
Token = RESERVED_KEYWORD | String = print
Token = Invalid | String = $
Token = Delimeter | String = ;
```

Result: The actual output matched the expected output and the test passed.

Task 2 – Hand-crafted LL(1) parser

In this task I was required to develop a hand-crafted predictive parser to work alongside the lexer built in Task 1.

My approach to this task was to find a way to implement the EBNF rules provided in the assignment specifications. My final solution was to create multiple methods which would each have a role to play in building the AST.

First of all, the *GetNextToken()* function works by declaring an instance of the lexer class and calling the *NextWord()* function until no other tokens were being returned. In this case, the **Epsilon** token would be returned instead. This is also where the lexical errors are reported. If an **Invalid** token is passed through by the lexer, an Exception is thrown and the program exits gracefully. The *GetNextToken()* function can be seen below:

```
static TOKEN getNextToken() throws Exception {
    currentToken = l.NextWord();
    if(currentToken.getToken() == TKN.Invalid){
        throw new Exception("Unidentified symbol found! "+currentToken.getString() +" is not a valid symbol!");
    }
    if(currentToken.getToken() == TKN.EPSILON){
        currentToken.setString("");
    }
    return currentToken;
}
```

The parser starts out in the **ParseProgram()** method which leads into the **ParseStatement()** method. This is method is continuously called while tokens are still being supplied by the lexer. The root node is declared in the *ParseProgram* method and every time the *ParseStatement* method is called, its return value is added as a child to the not rode. This is the beginning of the parse tree. Multiple statements make up the program as stated by the below EBNF:

$$\langle Program \rangle \quad ::= \{ \langle Statement \rangle \}$$

The **ParseStatement()** method declares a new node and sets its name to be “STATEMENT” in order to be labelled for when it makes up part of the AST. The next token is requested, and it is checked to see whether it is a **RESERVED_KEYWORD** or an **Identifier**. Anything other than these two will cause the parser to report a syntax error. If the token is of **RESERVED_KEYWORD** and it carries a String “let”, “print” or “return”, a semicolon will be expected after the statement, as per the **Statement** EBNF. This process also applied if the token is an **Identifier**.

$$
\begin{aligned}
\langle Statement \rangle \quad ::= \; & \langle VariableDecl \rangle \text{ ';'} \\
| \; & \langle Assignment \rangle \text{ ';'} \\
| \; & \langle PrintStatement \rangle \text{ ';'} \\
| \; & \langle IfStatement \rangle \\
| \; & \langle ForStatement \rangle \\
| \; & \langle WhileStatement \rangle \\
| \; & \langle RtrnStatement \rangle \text{ ';'} \\
| \; & \langle FunctionDecl \rangle \\
| \; & \langle Block \rangle
\end{aligned}
$$

If the **ParseStatement()** function finds an identifier, the **parseAssignment()** function is called. This constantly calls the **GetNextToken()** function, and matches each token with the required syntax, as seen in the EBNF below. Every character that matches the syntax, is added to a newly created node, and eventually added to the AST. If a character is missing or does not match the syntax pattern, an Exception is thrown, detailing the exact error so that the user knows what is wrong with their program's syntax.

$$
\langle Assignment \rangle \quad ::= \langle Identifier \rangle \text{ '='} \langle Expression \rangle
$$

The **parseExpression()** function works together with three other functions, namely the **parseSimpleExpression()**, **parseTerm()** and the **factor()** functions. These are coded exactly as required by the below EBNF:

$$
\langle Expression \rangle \quad ::= \langle SimpleExpression \rangle \; \{ \; \langle RelationalOp \rangle \; \langle SimpleExpression \rangle \; \}
$$

$$
\langle SimpleExpression \rangle \; ::= \langle Term \rangle \; \{ \; \langle AdditiveOp \rangle \; \langle Term \rangle \; \}
$$

$$
\langle Term \rangle \quad ::= \langle Factor \rangle \; \{ \; \langle MultiplicativeOp \rangle \; \langle Factor \rangle \; \}
$$

$$
\begin{aligned}
\langle Factor \rangle \quad ::= \; & \langle Literal \rangle \\
| \; & \langle Identifier \rangle \\
| \; & \langle FunctionCall \rangle \\
| \; & \langle SubExpression \rangle \\
| \; & \langle Unary \rangle
\end{aligned}
$$

The **Expression** function loops the **SimpleExpression** function every time a **RelationalOp** token is found. Similarly the **SimpleExpression** function loops the **Term** function every time an **AdditiveOp** token is found, and the **Term** function repeatedly calls the **Factor** function every

time a **MultiplicativeOp** token is found. The **Factor** function works by using a switch-case statement on the token. If it is seen to be a Literal, the respective node is created and added as a leaf node in the AST. If an identifier is found, this is parsed in order to check whether it is a FunctionCall or a normal identifier. If it is found to be a FunctionCall, a respective node with the title FunctionCall is created and added to the AST. SubExpressions and Unary are handled in a similar way, in order to make the AST as accurate as possible. Whatever factor is found, the newly created node is named accordingly and added as a leaf in the AST. I chose to implement the four functions this way to stay as true to the EBNF as possible, as seen above.

Going back to the **parseStatement()** function, if the token is seen to be a **RESERVED_KEYWORD**, the **parseKeyword()** function is called instead. This function uses a switch-case statement in order to choose whether the statement is one of **Variable Declaration, Print, Return, If, For, While,** or **Function Declaration**. These all have their respective functions in order to test the syntax and process any expressions, using the same function as described above. The following EBNF are used strictly in this application:

⟨*VariableDecl*⟩ ::= '`let`' ⟨*Identifier*⟩ '`:`' (⟨*Type*⟩ | ⟨*Auto*⟩) '`=`' ⟨*Expression*⟩

⟨*PrintStatement*⟩ ::= '`print`' ⟨*Expression*⟩

⟨*RtrnStatement*⟩ ::= '`return`' ⟨*Expression*⟩

⟨*IfStatement*⟩ ::= '`if`' '`(`' ⟨*Expression*⟩ '`)`' ⟨*Block*⟩ ['`else`' ⟨*Block*⟩]

⟨*ForStatement*⟩ ::= '`for`' '`(`' [⟨*VariableDecl*⟩] ';' ⟨*Expression*⟩ ';' [⟨*Assignment*⟩] '`)`' ⟨*Block*⟩

⟨*WhileStatement*⟩ ::= '`while`' '`(`' ⟨*Expression*⟩ '`)`' ⟨*Block*⟩

⟨*FunctionDecl*⟩ ::= '`ff`' ⟨*Identifier*⟩ '`(`' [⟨*FormalParams*⟩] '`)`' '`:`' (⟨*Type*⟩ | ⟨*Auto*⟩) ⟨*Block*⟩

Each of the above functions work very similarly. Each keyword, together with each character from their respective statement is placed in the AST.

The **Block** token seen in the above EBNF is implemented by iterating each token and checking whether it corresponds with its respective EBNF syntax. Inside the **parseBlock()** function, the **parseStatement()** function is looped until all the statements in the block have been processed, as per the below EBNF:

⟨*Block*⟩ ::= '`{`' { ⟨*Statement*⟩ } '`}`'

If one looked at the above statement EBNF, one could notice that the **FunctionDecl** statement contains a unique token, the **FormalParams**. These are parsed as per their EBNF statements. This is done by calling the **parseFormalParams()** function. This, in turn, repeatedly calls the

parseFormalParam() function while there are still more parameters to be processed. This function then checks the syntax, and if correct, creates a new node with all of the required branches and leaves added. This is then added to the AST. This follows the above EBNF:

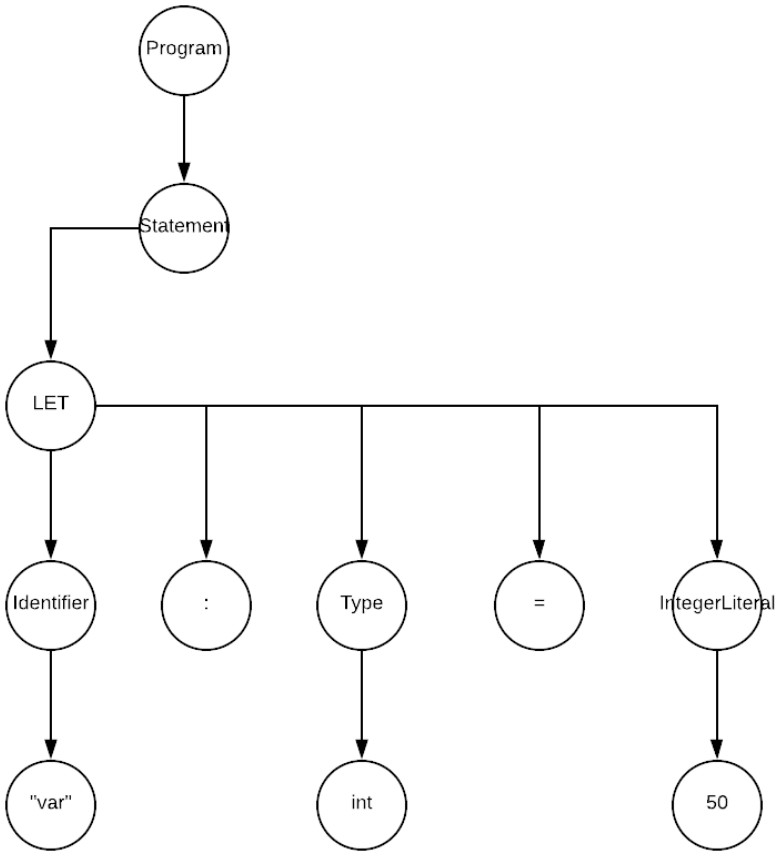
$$\langle FormalParam \rangle \quad ::= \langle Identifier \rangle \text{ ':' } \langle Type \rangle$$

$$\langle FormalParams \rangle \quad ::= \langle FormalParam \rangle \{ \text{ ',' } \langle FormalParam \rangle \}$$

Every time a statement is executed in the program/block, a syntax check determines whether each statement ends with a **; delimeter**. The If statement, For statement, While statement and Function Declaration statement are excluded from this check.

Once the parsing has been successfully completed, the XML, Semantic Analysis an Interpreter functions are called. If there is an error in the syntax, an Exception will be thrown and the program will exit for the user. Testing in the program suggests that there is no way for the parser to crash, regardless of the input.

As an example, if the statement “let var : int = 50;” was inputted into the program, the resulting AST would look like this:



Testing the parser

In order to test the parser, a temporary main class has been created. This main class will call and run an instance of the lexer with a String. This String will be occupied with the statement that will be tested. Both syntactically correct and incorrect statements will be tested in order to check whether the parser can successfully identify specific errors and throw Exceptions as intended. The said main class can be seen below:

```
public static void main(String[] args) throws Exception {
    System.out.println("Syntax Checker:");

    ASTNode rootnode = new ASTNode();
    rootnode.data = "PROGRAM";
    rootnode = ParseProgram();

    SemanticAnalyser.Semantic(rootnode);
}
```

Test 1 – Variable Declaration Statement

In this test, a syntactically correct Variable Declaration statement will be inputted into the parser.

Input: String containing “let var : int = 50; let var1 : float = 35.5; let var2 : bool = true;”

Expected Output: The program outputs that the program is syntactically correct.

```
Syntax Checker:
let var : int = 50; let var1 : float = 35.5; let var2 : bool = true;
Valid parser syntax!
```

Result: The actual output matches the expected output and the test passed.

Test 2 – Assignment Statement

In this test, a syntactically correct Assignment statement will be inputted into the parser.

Input: String containing “var = 15;”

Expected Output: The program outputs that the program is syntactically correct.

```
Syntax Checker:
var = 15;
Valid parser syntax!
```

Result: The actual output matches the expected output and the test passed.

Test 3 – Print Statement

In this test, a syntactically correct PrintS statement will be inputted into the parser.

Input: String containing “print 15 + 16; print 16 * 30; print 14 > 10;” in order to test the Expression, SimpleExpression and Term functions as well.

Expected Output: The program outputs that the program is syntactically correct.

```
Syntax Checker:
print 15 + 16; print 16 * 30; print 14 > 10;
Valid parser syntax!
```

Result: The actual output matches the expected output and the test passed.

Test 4 – If Statement

In this test, a syntactically correct If statement will be inputted into the parser.

Input: String containing “if(10 > 5){ print 5; }”

Expected Output: The program outputs that the program is syntactically correct.

```
Syntax Checker:
if(10 > 5){ print 5; }
Valid parser syntax!
```

Result: The actual output matches the expected output and the test passed.

Test 5 – For Statement

In this test, a syntactically correct For statement will be inputted into the parser.

Input: String containing “for(let var : int = 0; var < 10; var = var + 1){ print var; }”

Expected Output: The program outputs that the program is syntactically correct.

```
Syntax Checker:
for(let var : int = 0; var < 10; var = var + 1){ print var; }
Valid parser syntax!
```

Result: The actual output matches the expected output and the test passed.

Test 6 – While Statement

In this test, a syntactically correct While statement will be inputted into the parser.

Input: String containing “while(var < 10){ print var; var = var + 1; }”

Expected Output: The program outputs that the program is syntactically correct.

```
Syntax Checker:
while(var < 10){ print var; var = var + 1; }
Valid parser syntax!
```

Result: The actual output matches the expected output and the test passed.

Test 7 – Return Statement

In this test, a syntactically correct Return statement will be inputted into the parser.

Input: String containing “return 5 + 10;”

Expected Output: The program outputs that the program is syntactically correct.

```
Syntax Checker:
return 5 + 10;
Valid parser syntax!
```

Result: The actual output matches the expected output and the test passed.

Test 8 – Function Declaration Statement

In this test, a syntactically correct Function Declaration statement will be inputted into the parser.

Input: String containing “ff function(var : int, var1 : int):int{ return var + var1; }”

Expected Output: The program outputs that the program is syntactically correct.

```
Syntax Checker:
ff function(var : int, var1 : int):int{ return var + var1; }
Valid parser syntax!
```

Result: The actual output matches the expected output and the test passed.

Test 9 – Syntactically incorrect Statement

In this test, a syntactically incorrect Variable Declaration statement will be inputted into the parser.

The expected output will be that the Exception thrown will highlight the syntactical error, instead of throwing a generic error message.

Input: String containing “let var : int = 5”

Expected Output: The program outputs that the program is syntactically incorrect, and highlights the lack of a ‘;’ symbol.

```
Syntax Checker:
let var : int = 5
Exception in thread "main" java.lang.Exception: Syntax error! ';' expected! Found
    at Parser.ParseStatement(Parser.java:494)
    at Parser.ParseProgram(Parser.java:610)
    at Parser.main(Parser.java:628)
```

Result: The actual output matches the expected output and the test passed.

Test 10 – Syntactically incorrect Statement

In this test, a syntactically incorrect Assignment statement will be inputted into the parser.

The expected output will be that the Exception thrown will highlight the syntactical error, instead of throwing a generic error message.

Input: String containing “var == 3;”

Expected Output: The program outputs that the program is syntactically incorrect, and highlights the lack of a ‘;’ symbol.

```
Syntax Checker:
var == 3;
Exception in thread "main" java.lang.Exception: Incorrect syntax! Unexpected symbol: =
```

Result: The actual output matches the expected output and the test passed.

Test 11 – Syntactically incorrect Statement

In this test, a syntactically incorrect If statement will be inputted into the parser.

The expected output will be that the Exception thrown will highlight the syntactical error, instead of throwing a generic error message.

Input: String containing “if(){ print 5; }” (no Expression inside the brackets).

Expected Output: The program outputs that the program is syntactically incorrect, and highlights the lack of a ‘;’ symbol.

```
Syntax Checker:
if(){ print 5; }
Exception in thread "main" java.lang.Exception: Incorrect syntax! Unexpected symbol: )
```

Result: The actual output matches the expected output and the test passed.

Test 12 – Syntactically incorrect Statement

In this test, a syntactically incorrect If statement will be inputted into the parser.

The expected output will be that the Exception thrown will highlight the syntactical error, instead of throwing a generic error message.

Input: String containing “if(5 > 3){ print 5; ” (no closing ‘}’).

Expected Output: The program outputs that the program is syntactically incorrect, and highlights the lack of a ‘;’ symbol.

```
Syntax Checker:
if(5 > 3){ print 5;
Exception in thread "main" java.lang.Exception: Syntax Error! Expected '}', found EPSILON
```

Result: The actual output matches the expected output and the test passed.

Task 3 – AST XML Generation Pass

In this task, I was meant to program a function which would visit each node in the AST and display the contents of that node in XML form, including indentation. The way I chose to tackle this problem was to create different functions which would each perform a different action, depending on what node was currently being processed. In simpler terms, if for example a node named “Assignment” was fed into the XML generator, the **genAssignment()** function would be called in order to process this node and output its contents to the user. Since every node has a set structure, it is possible to hardcode all the required procedures to obtain the node’s data, which is why this method was chosen. It shares similarities with the visitor design pattern due to its process of visiting each node and executing different instructions for each scenario. In my version, it is just as easy to add new symbols or objects to the code and have them displayed in an XML format.

Firstly, in order to help with the indentations, a function named **addTab()** was created. This accepts an integer and will indent however many times needed. Therefore, to accompany this function, a global variable named *indent* was also created to keep track of the current indentation.

In order to generate the XML output, one must call the **generateXML()** function while passing the root node of the AST to it. This function then calls the **genStatement()** function for every child stemming out of the root node. This function will then use a switch-case statement in order to determine the data present in the child node. Depending on what data is found in the node, one of the following functions will be called:

genVarDecl()

First, this function increases the indent. Next, it displays its tag (<VarDecl>) and extracts the data it needs from the nodes in the AST, meaning the variable type and identifier. This data is outputted to the user in XML format and the **genExpression()** function is called to output the Expression too. The function’s output is pictured below.

```
Syntax Checker:
let var : int = 50;
Valid parser syntax!
<PROGRAM>
    <STATEMENT>
            <VarDecl>
                <Var Type = "int">var</Id>
                <IntegerLiteral>"50"</IntegerLiteral>
            </VarDecl>
    </STATEMENT>
</PROGRAM>
```

genPrint()

Firstly, this function prints its tag (<Print>) and increases its indent. Next, the **genExpression()** function is called to correctly display the Expression, as per the EBNF. Below is an example of the function's output.

```
Syntax Checker:
print 15;
Valid parser syntax!
<PROGRAM>
    <STATEMENT>
        <Print>
            <IntegerLiteral>"15"</IntegerLiteral>
        </Print>
    </STATEMENT>
</PROGRAM>
```

genReturn()

Firstly, this function prints its tag (<Return>) and increases its indent. Next, the **genExpression()** function is called to correctly display the Expression, as per the EBNF. Below is an example of the function's output.

```
Syntax Checker:
return 25.5;
Valid parser syntax!
<PROGRAM>
    <STATEMENT>
        <Return>
            <FloatLiteral>"25.5"</FloatLiteral>
        </Return>
    </STATEMENT>
</PROGRAM>
```

genIfStatement()

Firstly, this function prints its tag (<IfStatement>) and increases the indent. Next, the **genExpression()** function is called to display the if statement's condition in XML format. After this, the function calls the **genBlock()** function in order to display the If Statement's Block in XML format. Below is an example of the function's output.

```
Syntax Checker:
if(var > 3){ print var; }else{ print false; }
Valid parser syntax!
<PROGRAM>
    <STATEMENT>
        <IfStatement>
            <BinExprNode Op = ">">
                <Id>"var"</Id>
                <Digit>"3"</Digit>
            </BinExpNode>
        <Block>
            <STATEMENT>
                <Print>
                    <Id>"var"</Id>
                </Print>
            </STATEMENT>
        </Block>
            <Else>
                <Block>
                    <STATEMENT>
                        <Print>
                            <BooleanLiteral>"false"</BooleanLiteral>
                        </Print>
                    </STATEMENT>
                </Block>
            <Else>
        </IfStatement>
    </STATEMENT>
</PROGRAM>
```

genForStatement()

Firstly, this function displays its tag (<ForStatement>) an increments the indent. Next, if a variable declaration statement is present, the **genVarDecl()** function is run. After this, the **genExpression()** function is called, and the *Expression* node from the AST is passed to it. Once this function returns, the **genAssignment()** function is called if there is an assignment statement present in the for statement. Finally, the **genBlock()** function is called to display the For statement *Block* and the function returns. Below is an example of the function's output.

```
Syntax Checker:
for(let var : int = 0; var < 10; var = var + 1){ print var; }
Valid parser syntax!

<PROGRAM>
    <STATEMENT>
        <ForStatement>
            <VarDecl>
                <Var Type = "int">var</Id>
                <Digit>"0"</Digit>
            </VarDecl>
            <Expression>
                <BinExprNode Op = "<">
                    <Id>"var"</Id>
                    <IntegerLiteral>"10"</IntegerLiteral>
                </BinExpNode>
            </Expression>
            <Assignment>
                <Id>var</Id>
                <AssignmentOp = "=">
                    <BinAddNode Op = "+">
                        <Id>"var"</Id>
                        <Digit>"1"</Digit>
                    </BinAddNode>
                </AssignmentOp>
            </Assignment>
            <Block>
                <STATEMENT>
                    <Print>
                        <Id>"var"</Id>
                    </Print>
                </STATEMENT>
            </Block>
        </ForStatement>
    </STATEMENT>
</PROGRAM>
```

genWhileStatement()

This function starts out by increasing the indent and displaying its tag (<WhileStatement>). After this, it calls the **genExpression()** function to display its *Expression* node, and also calls the **genBlock()** function to display its *Block* node. Below is an example of this function's output.

```
Syntax Checker:
while(var < 5){ print var; var = var + 1; }
Valid parser syntax!
<PROGRAM>
    <STATEMENT>
        <WhileStatement>
            <BinExprNode Op = "<">
                <Id>"var"</Id>
                <Digit>"5"</Digit>
            </BinExpNode>
            <Block>
                <STATEMENT>
                    <Print>
                        <Id>"var"</Id>
                    </Print>
                </STATEMENT>
                <STATEMENT>
                    <Assignment>
                        <Id>var</Id>
                        <AssignmentOp = "=">
                            <BinAddNode Op = "+">
                                <Id>"var"</Id>
                                <Digit>"1"</Digit>
                            </BinAddNode>
                        </AssignmentOp>
                    </Assignment>
                </STATEMENT>
            </Block>
        </WhileStatement>
    </STATEMENT>
</PROGRAM>
```

genFunctionCall()

This function displays the contents and details of a Function Declaration statement. It first extracts the required information from the AST nodes and displays them itself. Next, it calls the **genFormalParams()** function to display its parameters, and finally calls the **genBlock()** function to display its *Block* node. Below is an example of the outputted XML.

```
Syntax Checker:
ff function(var : int) : int{ return var * 5; }
Valid parser syntax!
<PROGRAM>
    <STATEMENT>
        <FunctionCall Type = "int">
            <FunctionName>
                <Id>function</Id>
            </FunctionName>
            <FormalParams>
                <FormalParam>
                    <Var Type = "int">var</Id>
                </FormalParam>
            </FormalParams>
                <Block>
                    <STATEMENT>
                        <Return>
                            <BinMultNode Op = "*">
                                <Id>"var"</Id>
                                <Digit>"5"</Digit>
                            </BinMultNode>
                        </Return>
                    </STATEMENT>
                </Block>
        </FunctionCall>
    </STATEMENT>
</PROGRAM>
```

genAssignment()

This function is used to generate the XML text of an Assignment statement. The information required is obtained from the AST nodes. This information is then formatted and displayed accordingly. An example of an output from this function is below.

```
Syntax Checker:
var = 50;
Valid parser syntax!
<PROGRAM>
    <STATEMENT>
        <Assignment>
            <Id>var</Id>
            <AssignmentOp = "=">
                <IntegerLiteral>"50"</IntegerLiteral>
            </AssignmentOp>
        </Assignment>
    </STATEMENT>
</PROGRAM>
```

Task 4 – Semantic Analysis Pass

In this task, I was requested to program a semantic analyzer whose job is to navigate through the previously-generated AST, visit each node and execute certain functions on each node, which determines whether the statements written by the user are semantically correct.

In order to correctly implement a semantic analyzer, a SymbolTable class was created. This class contains a nestled class named Row. A Row contains a variety of elements, including a classtype (such as "Variable", "Function", etc..), a type (such as int, float and bool), a variablename and a scope variable. Whenever a Row is declared, it is added to the stable ArrayList located in the SymbolTable class. The choice of using a dynamic array was taken in order to avoid any overflow possibilities which may occur when a static array is filled up too much. The SymbolTable methods allow the user to edit the type of a variable which is already in the table (using the editType function), delete every entry in the symbol table which exists in a scope (using the deleteScope function), find an entry in the SymbolTable using only a scope value and classtype String, find whether a Row containing the given classtype, variablename and scope exists in the SymbolTable, and the ability to add a new Row to the ArrayList using the addRow function.

Once the **Semantic()** function is called, a rootnode must be passed to it and the program will call the **scanStatement()** function for every child stemming from the root node. Each child's data will be examined, and the according function will be called. These functions are as follows:

scanVarDecl()

This function is used to scan the semantics of a Variable Declaration statement. It's main task is to find whether the declared variable is being assigned the correct type, based on its variable type. For example, a variable declared as an *int* can only be assigned integer numbers, and so on. This function first checks whether there is already an instance of a variable with the same name as this one. This is done by using the **symboltablelookup()** function. If there already exists a variable with the same name in the current scope, an appropriate error message is displayed and the program is stopped. Else, the function continues into the type checking. If the variable is declared as an *int*, *float* or *bool*, the program will determine whether the given value matches the type of the variable using the **scanExpression()** function. If, in fact the variable is declared as an *Auto*, the program uses the **scanExpression()** function to determine what type of value it is, and will change the variable's type in the AST and SymbolTable accordingly.

scanPrint()

This function is a very simple one whose only task is to check whether the Expression supplied is semantically correct. It does this by calling the **scanExpression()** function.

scanReturn()

This function has three main tasks. The first one is to check whether the *return* statement lies inside a function or not, as one cannot have a *return* statement outside of one. The second task if to check whether the Expression attached to the return statement is a valid one. This is done by calling the **scanExpression()** function. The third task is to check that the return statement returns a type which matches the function type. For example, a function of type *int* cannot return a *float*. This is done by extracting the function's Row from the SymbolTable and making sure that the return type and function type match. On the other hand, if the function's type is set to *Auto*, the function's type will be set to the return statement's type in both the AST and the SymbolTable accordingly. A Boolean variable is also used to check whether each function contains a return statement. This is set to *true* whenever the **scanReturn()** function is called.

scanIfStatement()

The first thing this function does, is increment the scope variable, due to the fact that the *If Statement* can contain local variables that will only be available for use within that scope. Once this is done, the **scanExpression()** function is called, to check that the Expression within the *If Statement*'s syntax is semantically correct. Once this is done, the **scanBlock()** method is called, which processes the *If Statement*'s Block to check if it semantically correct.

scanForStatement()

First, this function calls the **scanVarDecl()** function if there is a variable declaration present in the statement. Next, the **scanExpression()** function is called, in order to check the semantics of the Expression in the *For Statement*. Finally, if any *Assignments* are present, these are checked semantically too, using the **scanAssignment()** function. Once all this has been executed, the **genBlock()** function is called to check the Block of the statement. Once this has been completed, the scope value is returned to normal, and the loop and all of its variables are removed from the SymbolTable.

scanWhileStatement()

As soon as it is called, this function increments the scope as the *While Statement* may contain variables that can only be utilized within it. Next, the **scanExpression()** function is called to check the *While Statement's* Expression, and after this, the **scanBlock()** function is called to scan the *While Statement's* Block. Once all this is done, the scope is returned back to normal, and the function ends.

scanFunctionCall()

The first thing that this function does is that it adds a new Row to the SymbolTable, of type *Function*. Once this is added, the **scanFormalParams()** function is called to check the semantics of the function's parameters. The **scanBlock()** function is now called to check the semantics of

the function's Block. Once these are done, and the *Return* Boolean is true (meaning that there was a return statement present in the block, the function ends.

scanAssignment()

Once this function is called, the variable's name is extracted from the AST and the symboltablelookup function is used to determine whether the variable exists in the scope. If the variable has not been declared, a detailed Exception is thrown and the program is aborted. Else, the value entered by the user is extracted from the AST and if the variable's type matches the value's type, the function returns normally.

Helper Functions

The functions below are functions which are utilized in the above statement functions. The functions are split out this way to stay as true to the initial EBNF as possible, and make it very simple to add another function, if one requires.

scanFormalParams()

This function is used exclusively in the **scanFunctionCall()** function, and is used to check the semantics of the parameters declared in that statement. It is possible that no parameters are declared, therefore if there are no *FormalParams* children, then the function instantly returns. Else, the **scanFormalParam()** function is called.

scanFormalParam()

This function is used in order to add any new variables declared in the function statement's parameters to the SymbolTable. If no variables have been declared, the function instantly returns. Else, the function adds these variables to the SymbolTable using the addRow function.

scanExpression()

This function is used to generate the *type* which an Expression will return. For example, the Expression "5 + 5" will return an *int*, whereas the Expression "5 + true" is a semantic error. Every time a RelationalOp is found in the AST, the next value in the Expression is extracted, and the **scanSimpleExpression()** function is called, passing that node. This goes on until every value in the Expression has been scanned, and the *type* is returned in the form of a String. Meanwhile, if any two *types* returned by the **scanSimpleExpression()** don't match, an Exception will be thrown, detailing the exact error.

scanSimpleExpression()

This function's task is to call the **scanTerm()** function for every child contained in the node. If the current *type* String returned by the **scanTerm()** function does not equal the previous one, a detailed Exception is thrown.

scanTerm()

This function's task is to call the **scanFactor()** function for every child contained in the node. If the current *type* String returned by the **scanFactor()** function does not equal the previous one, a detailed Exception is thrown.

Testing the Semantic Analyzer

In this section of the documentation, I will be using a temporary main class in order to test the semantic analyzer. This class will be located in the parser, and will call the **Semantic()** function together with the root node. An input will be specified, together with an expected output.

Test 1 – Variable Declaration Statement

In this test, a syntactically and semantically correct Variable Declaration statement will be inputted into the parser.

Input: String containing "let var : int = 50;"

Expected Output: The program outputs that the program is syntactically and semantically correct.

```
Syntax Checker:
let var : int = 50;
Valid parser syntax!
Statement is semantically correct!
```

Result: The actual output matches the expected output and the test passed.

Test 2 – Variable Declaration Statement

In this test, a syntactically and semantically correct Variable Declaration statement with the 'auto' type will be inputted into the parser.

Input: String containing "let var : auto = 50;"

Expected Output: The program outputs that the program is syntactically and semantically correct and that the type has been changed.

```
Syntax Checker:
let var : auto = 50;
Valid parser syntax!
Variable type has been changed!
Statement is semantically correct!
```

Result: The actual output matches the expected output and the test passed.

Test 3 – Variable Declaration Statement

In this test, a syntactically correct and semantically incorrect Variable Declaration statement.

Input: String containing "let var : int = 50.5;"

Expected Output: The program outputs that the program is syntactically correct and semantically incorrect.

```
Syntax Checker:
let var : int = 50.5;
Valid parser syntax!
Exception in thread "main" java.lang.Exception: Variable type error! Expected int Found : float
```

Result: The actual output matches the expected output and the test passed.

Test 4 – Re-Assigning a variable Statement

In this test, a syntactically correct and semantically correct Variable Declaration statement, and an assignment statement.

Input: String containing “let var : int = 50; var = 40;”

Expected Output: The program outputs that the program is syntactically and semantically correct.

```
Syntax Checker:
let var : int = 50; var = 40;
Valid parser syntax!
Statement is semantically correct!
```

Result: The actual output matches the expected output and the test passed.

Test 5 – Re-Assigning a variable Statement

In this test, a syntactically correct and semantically incorrect assignment statement. The variable being assigned to will not have previously been declared.

Input: String containing “var = 40;”

Expected Output: The program outputs that the program is syntactically correct and semantically incorrect due to the variable not having been declared.

```
Syntax Checker:
var = 40;
Valid parser syntax!
Exception in thread "main" java.lang.Exception: Variable var does not exist!
```

Result: The actual output matches the expected output and the test passed.

Test 6 – Print Expression Statement

In this test, a syntactically and semantically correct print statement.

Input: String containing “print 50 + 40 * 30 > 90;”

Expected Output: The program outputs that the program is syntactically and semantically correct.

```
Syntax Checker:
print 50 + 40 * 30 > 90;
Valid parser syntax!
Statement is semantically correct!
```

Result: The actual output matches the expected output and the test passed.

Test 7 – Print Declared Variable Statement

In this test, a syntactically and semantically correct print variable statement and variable declaration statement.

Input: String containing “let var : int = 50; print var + 20;”

Expected Output: The program outputs that the String is syntactically and semantically correct.

```
Syntax Checker:
let var : int = 50; print var + 20;
Valid parser syntax!
Statement is semantically correct!
```

Result: The actual output matches the expected output and the test passed.

Test 8 – Print Undeclared Variable Statement

In this test, a syntactically correct and semantically incorrect print variable statement and variable declaration statement.

Input: String containing “print var + 20;”

Expected Output: The program outputs that the String is syntactically correct and semantically incorrect due to ‘var’ not being a declared variable.

```
Syntax Checker:
print var + 20;
Valid parser syntax!
Exception in thread "main" java.lang.Exception: Variable var does not exist!
```

Result: The actual output matches the expected output and the test passed.

Test 9 – If Statement

In this test, a syntactically and semantically correct If statement.

Input: String containing “if(15 > 5){ print 5; }”

Expected Output: The program outputs that the String is syntactically and semantically correct.

```
Syntax Checker:
if(15 > 5){ print 5; }
Valid parser syntax!
Statement is semantically correct!
```

Result: The actual output matches the expected output and the test passed.

Test 10 – While Statement

In this test, a syntactically and semantically correct While statement.

Input: String containing “while(15 > 5){ print 5; }”

Expected Output: The program outputs that the String is syntactically and semantically correct.

```
Syntax Checker:
while(15 > 5){ print 5; }
Valid parser syntax!
Statement is semantically correct!
```

Result: The actual output matches the expected output and the test passed.

Test 11 – For Statement

In this test, a syntactically and semantically correct For statement.

Input: String containing “for(let var : int = 5; var < 10; var = var +1){ print var; }”

Expected Output: The program outputs that the String is syntactically and semantically correct.

```
Syntax Checker:
for(let var : int = 5; var < 10; var = var +1){ print var; }
Valid parser syntax!
Statement is semantically correct!
```

Result: The actual output matches the expected output and the test passed.

Test 12 – Function Declaration Statement

In this test, a syntactically and semantically correct Function Declaration statement.

Input: String containing “ff function(var : int):int{ return var + var * 2; }”

Expected Output: The program outputs that the String is syntactically and semantically correct.

```
Syntax Checker:
ff function(var : int):int{ return var + var * 2; }
Valid parser syntax!
Statement is semantically correct!
```

Result: The actual output matches the expected output and the test passed.

Test 13 – Function Declaration Statement

In this test, a syntactically correct and semantically incorrect Function Declaration statement, with no return statement.

Input: String containing “ff function(var : int):int{ print var + var * 2; }”

Expected Output: The program outputs that the String is syntactically correct and semantically incorrect.

```
Syntax Checker:
ff function(var : int):int{ print var + var * 2; }
Valid parser syntax!
Exception in thread "main" java.lang.Exception: Function requires a return statement!
```

Result: The actual output matches the expected output and the test passed.

Task 5 – Interpreter Execution Pass

In this task, I was instructed to create an interpreter to execute the code which was placed in the AST by the parser, and semantically checked by the semantic analyzer. The code seen in the interpreter program is very similar to the one used in the semantic analyzer, as the steps used to navigate through the AST are the same.

First of all, a new variation of the SymbolTable used in the semantic analyzer is being used, this time being labelled as the InterpreterSymbolTable. The difference between this class and the previous SymbolTable is that this one has the option to hold a value in a Row instance. This allows the program to keep track of variables and their values. Many of the class's functions remain the same, with the added **addFuncRow()** method, which stores a function together with its contents in a specialized Row. Other than this, the **editType()** function has been replaced with an **editValue()** function which is used to alter the values of variables if a new assignment takes place.

The Interpreter starts functioning as soon as the **Interpret()** function is called and is given a root node. For every child belonging to this root node, the **runStatement()** function is called, similar to the operation taken out in the semantic analyzer. This, once again, takes the data from the node and determines which function should be called from the ones listed below:

runVarDecl()

This function is tasked with creating a variable and assigning a value to it. First, the value is generated by the **runExpression()** function, with the appropriate node being passed to it. Once this value has been generated, a new Row is added to the SymbolTable with the variable name and value, which were extracted from their appropriate nodes from the AST.

runPrint()

This function simply outputs the returned value from the **runExpression()** function onto the screen for the user to see.

runReturn()

This function should only be used within a FunctionCall block.

runIfStatement()

This function firstly increments the global scope variable, as in the semantic analyzer, and only calls the **runBlock()** statement if the given condition is equal to true. In order to calculate whether this condition evaluates to true, the Expression node is passed to the **runExpression()**

function, and the output is compared to the String “true”. Once this function has executed, the scope is decremented, and the function returns.

runForStatement()

This function starts off by incrementing the global scope variable and calling the **runVarDecl()** function if any variable declaration statements are present in the statement. Once this is done, the condition is checked as to whether it is true, by using the **runExpression()** function and comparing the output to the String “true”. This prompts the program to enter a while loop of continuously calling the **runBlock()** function and **genAssignment()** function in order to simulate a working for loop, continuously running its block and incrementing the given value. Once the condition is no longer satisfied, the scope is decremented again, and the function returns.

runWhileStatement()

This function starts off by incrementing the global scope variable. The condition for the while loop to repeat is calculated by the **runExpression()** function, and if found to be true, is repeated until it is found to be false. Since my semantic analyzer does not check whether there is a variable which continuously updates itself in the loop, it is possible to fall into an infinite loop. This action was not taken as it would have limited the amount of possibilities a user could do with a while loop. Every time the condition is found to be true, the **runBlock()** function is called and this keeps happening until the condition evaluates to “false”. After this, the function decrements the scope variable and returns.

runFunctionDecl()

This function’s task is to add a declared function to the Interpreter Symbol Table. It does this by calling the **addFuncRow()** command, and passing through the classtype, function name, scope and its block node. This is passed through so that when a FunctionCall is made to this function, the block can easily be found and executed. Once this has been accomplished, the function returns.

runAssignment()

This function works in exactly the same way as its counterpart in the semantic analyzer. First, the value is determined with help from the **runExpression()** function. This value is then used in the **editValue()** function in order to re-assign the value to the already-existing variable in the symbol table.

Helper functions

The below functions are used by the main functions in order to carry out their specific tasks. These functions exist in order to strictly follow the EBNF rules as closely as possible.

runExpression()

This function's task is to carry out the arithmetic operations between two *SimpleExpressions* using the RelationalOp token. For every child stemming from this *Expression* node, the *runSimpleExpression()* function is called. The returned value from this function will either be subject to an arithmetic operation (if there is more than one child stemming from the *Expression* node), or it will be returned as is (if there is only one child stemming from the *Expression* node).

runSimpleExpression()

This function runs exactly the same method as the **runExpression()** function with the added difference that AdditiveOp tokens are used to complete arithmetic operations on the returned values from the **runTerm()** function. This arithmetic operation is done by calling the **addition** function while passing through the values and the operator. This function will then return a new value in the form of a String. This process goes on until every child stemming from the *SimpleExpression* node has been acknowledged, and the new value is returned in the form of a String.

runTerm()

This function's job is to call the **runFactor()** function for each child stemming from the *Term* node. This function will also use the **addition()** function in order to calculate the new value using a MultipicativeOp, if found in the AST, similarly to how the **SimpleExpression()** function operates. Once every child has been visited and the new value has been calculated, this new value is returned by the function in the form of a String.

runFactor()

This function's job is to identify what factor is currently in the node being passed to it by the **runTerm()** function. This function returns a different String depending on what the node details the *Factor* as being. For example, if the *Factor* is an Identifier in the AST, the String "Identifier" is returned. This holds true for all other factor types, except the *FunctionCall* factor. This, instead calls the **runFunctionCall()** function in order to run the function stored in the symbol table. After this, the String "FunctionCall" is returned from the function.

Addition()

This function receives an operator, two values and a type. Its job is to calculate the arithmetic calculation between these two values given the operator. It does this by using a switch-case statement to find which operator is being used. Once found, another switch-case statement is used to check whether the variable type is a *float* or an *int*. Now, the two values are parsed into their type, the arithmetic operation is applied to them, they are converted back into a String and returned. The choice to calculate the arithmetic operations this way was taken because it is the easiest way found to manage different types and avoid any type interference.

runFunctionCall()

This function is used in order to execute a previously-defined function. The function simply searches for the called function by using the **getSymbol()** function and uses the **runBlock()** function on the content's of the *func* node obtained from the symbol table. As previously stated, the *func* variable contains the *Block* node from the function from when it was defined. Once this is executed, the scope is decremented again and the function returns.

runBlock()

This function is used to execute the *Block* from functions and loops. This simply iterates through all the children stemming from the *Block* node and calls the **runStatement()** function, passing through each child node one by one, until all children have been executed. Once this has been executed, the function returns.

Testing the Interpreter

In this section of the documentation, I will be creating a temporary main class which will be testing the functionality of the interpreter. This main class will still go through the process of going through the lexer, parser and semantic analyzer. For each test, I will be specifying an input String and an expected output String. If the actual output matches the expected output, then I can safely say that the test was successful.

Testing the Interpreter

In this section, I will be testing the Interpreter by presenting it with multiple statements and checking whether it will output the desired value. Due to the limitation on the execution of functions, they have NOT been included in this testing section.

Test 1 – Variable Declaration Statement

Input: A String containing “let var : int = 50; print var;”

Expected Output: The program should output the value “50”. This will indicate that the value has been saved to the variable correctly.

```
Syntax Checker:
let var : int = 50; print var;
Valid parser syntax!
Statement is semantically correct!
Print : 50
Program run complete!
```

Result: The actual output matched the expected output and the test was deemed successful.

Test 2 – Variable Assignment Statement

Input: A String containing “let var : int = 50; var = 65; print var;”

Expected Output: The program should output the value “65”. This will indicate that the value has been saved to the variable correctly, and that the assignment statement has successfully altered the value stored in the variable.

```
Syntax Checker:
let var : int = 50; var = 65; print var;
Valid parser syntax!
Statement is semantically correct!
Print : 65
Program run complete!
```

Result: The actual output matched the expected output and the test was deemed successful.

Test 3 – If Else Statement

Input: A String containing “let var : int = 5; if(var > 10){ print 10; }else{ print 15; }”

Expected Output: The program should output the value “15”. This will indicate that the value has been saved to the variable correctly, and that the if statement has correctly been calculated to “false” and therefore the else statement is executed.

```
Syntax Checker:
let var : int = 5; if(var > 10){ print 5; }else{ print 15; }
Valid parser syntax!
Statement is semantically correct!
Print : 15
Program run complete!
```

Result: The actual output matched the expected output and the test was deemed successful.

Test 4 – If Statement

Input: A String containing “let var : int = 5; if(var < 10){ print 10; }else{ print 15; }”

Expected Output: The program should output the value “10”. This will indicate that the value has been saved to the variable correctly, and that the if statement has correctly been calculated to “true” and therefore the statement is executed.

```
Syntax Checker:
let var : int = 5; if(var < 10){ print 10; }else{ print 15; }
Valid parser syntax!
Statement is semantically correct!
Print : 10
Program run complete!
```

Result: The actual output matched the expected output and the test was deemed successful.

Test 5 – If Statement

Input: A String containing “let var : int = 10; if(var == 10){ print 10; }else{ print 15; }”

Expected Output: The program should output the value “10”. This will indicate that the value has been saved to the variable correctly, and that the if statement has correctly been calculated to “true” and therefore the statement is executed.

```
Syntax Checker:
let var : int = 10; if(var == 10){ print 10; }else{ print 5; }
Valid parser syntax!
Statement is semantically correct!
Print : 10
Program run complete!
```

Result: The actual output matched the expected output and the test was deemed successful.

Test 6 – For Statement

Input: A String containing “for(let var : int = 5; var < 10; var = var + 1){ print var; }”

Expected Output: The program should output the value of var 5 times, meaning the values 5, 6, 7, 8 and 9.

```
Syntax Checker:
for(let var : int = 5; var < 10; var = var + 1){ print var; }
Valid parser syntax!
Statement is semantically correct!
Print : 5
Print : 6
Print : 7
Print : 8
Print : 9
Program run complete!
```

Result: The actual output matched the expected output and the test was deemed successful.

Test 7 – For Statement

Input: A String containing “for(let var : int = 5; var < 5; var = var + 1){ print var; }”

Expected Output: The program should output nothing, as the condition should have evaluated to “false”.

```
Syntax Checker:
for(let var : int = 5; var < 5; var = var + 1){ print var; }
Valid parser syntax!
Statement is semantically correct!
Program run complete!
```

Result: The actual output matched the expected output and the test was deemed successful.

Test 8 – While Statement

Input: A String containing “let var : int = 0; while(var < 5){ print var; var = var + 1; }”

Expected Output: The program should output 5 values of var, each one being incremented by 1 each time, as the condition of the while loop should evaluate to “true”.

```
Syntax Checker:
let var : int = 0; while(var < 5){ print var; var = var + 1; }
Valid parser syntax!
Statement is semantically correct!
Print : 0
Print : 1
Print : 2
Print : 3
Print : 4
Program run complete!
```

Result: The actual output matched the expected output and the test was deemed successful.

Test 9 – While Statement

Input: A String containing “let var : int = 0; while(var > 5){ print var; var = var + 1; }”

Expected Output: The program should output nothing as the while loop condition should evaluate to “false”.

```
Syntax Checker:
let var:int = 0; while(var > 5){ print var; var = var +1; }
Valid parser syntax!
Statement is semantically correct!
Program run complete!
```

Result: The actual output matched the expected output and the test was deemed successful.

Conclusion

Overall, I believe that my solutions to the problems at hand were very suitable. I believe that I stuck to the criteria very closely and I succeeded in making a user-friendly and efficient compiler. Although I didn't stick to the visitor design pattern, I believe my solution was very similar to this way of programming.

The EBNF was closely monitored and I made sure to create a solution which not only stuck closely to these EBNF statements, but also made it easy to add any new ones when required.

All in all, I believe my solution, despite its flaws, is a suitable solution to the problems at hand.