L-Università ta' Malta
Faculty of Information & Communication Technology
Department of Computer Information Systems

Compiler Theory and Practice Course Assignment (Part I)

Logan Formosa(434901L)
B.Sc.It (Hons) Artificial Intelligence

Study-unit: **Compiler Theory and Practice**
Code: **CPS2002**
Lecturer: **Mr Sandro Spina**

Contents

Introduction

In the following report, we will go through the Compiler built for the fictitious TeaLang language. The syntax and grammar rules for the TeaLang programming language were provided using an EBNF syntax, from which a program written fully in TeaLang can be built.

This first part of the assignment is split into 5 main tasks, in order these are:

1. Lexer
2. Parser and AST Building
3. XML Visitor
4. Semantic Visitor
5. Interpreter Visitor

All parts were attempted and successfully implemented as you will see as we go along through the documentation of the process as well by testing out the implemented artefact itself with the use of the supplementary test files provided.

The compiler was written in Java, each main task has its own class file as well as another auxiliary file for the AST Class. To run this compiler, select an example.txt file from the list of example files found in the respective directory, and place it in the run directory to run the specified text file.

Apart from the fully functional program files, I have also included some test files that were used during the implementation of specific components, the lexer_test file only reads the tokens, the parser_test produces the AST and checks the syntax, and the semantic analysis test file only checks for the semantics. The rest of the files operate normally.

Now let's take a look at the first component, which starts of the entire compilation process, the Lexer.

Lexical Analsysis

The objective of the lexer is to process each individual character within the supplied file, and construct tokens. These tokens would be individual words, or groups of characters that form valid words or rather tokens within our language. The list of tokens accepted by TeaLang can be found within the EBNF of the language and examining only the terminal components of the EBNF.

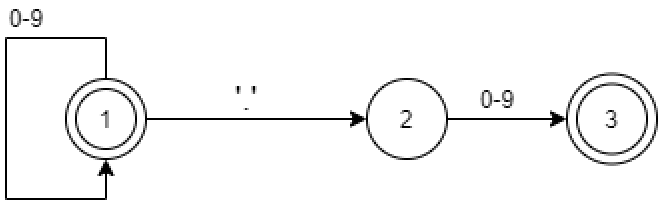
When determining what types of tokens can be accepted , careful consideration had to be made on how such token can be constructed as well, what characters can be a part of this type of token, what character can or cannot be included, if a character is included maybe it deviates to another type of token. All these considerations can be summed up to building a DFSA. Constructing a DFSA allows us to keep track the order of characters and seeing what type of token will be generated from a sequence of characters. A new character would indicate a transition from one state to another, and we may encounter characters which provide successful and valid transitions and others which do not. If it is the former, we carry on as usual, however if it is the latter we must consider whether or not this erroneous was brought about by a wrong character in the sequence, or if it was because a new character was included after a natural termination of the token. Final states will help us decide which of these two cases has been encountered, if we have arrived at a final state, then this sequence of tokens can be accepted as a fully-constructed token, and any character which follows is implied to belong to the following token, otherwise if the current state is not a final state, then if the character that follows gives an erroneous transition, then our sequence of characters is incorrect and not compliant with the TeaLang specifications.

Let us look at the specific transitions for different types of tokens.

Building DFSA's and Transitions

Digits

In the TeaLang language number may be int's or whole numbers, or they may be floats , representing numbers including a decimal point. So our first types of characters must be allowing the digits from 0-9 and allow for the possibility of a '.' After at least encountering a number, but after transitioning with the '.' Character, we must indicate that the number must not end here, so this intermediary state would not be marked as final but rather, only after another digit has it been encountered do we include a transition to a final state.



Here we see the final states, 1 and 3, with the intermediary state 2 being used to hold account for the decimal point found in between floating-point numbers.

Operators

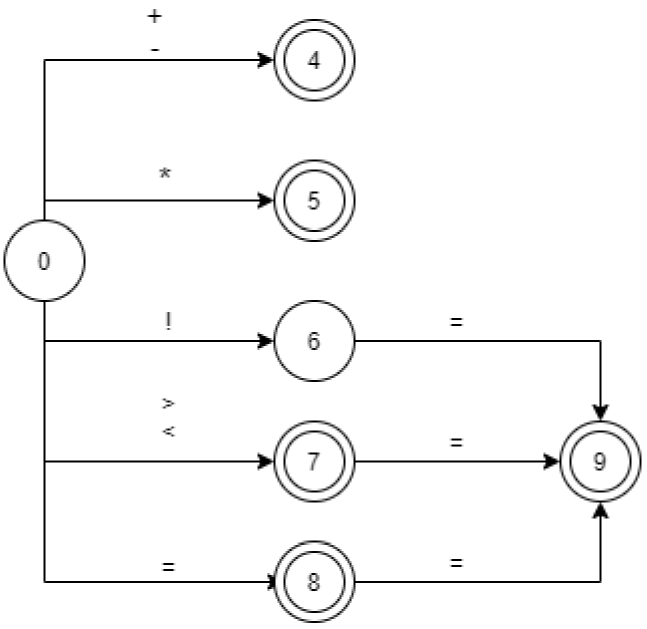
Next, the transition tables for operators were constructed. Since the majority of these were only a single character, all that was required for these was a single transition to an immediate final state, and then negating any future transitions. The operators in question are as follow:

+ - * > < >= <= = == !=

Apart from the final, relation operators which include a common end character of =, all previous operators require only a single state and transition, the rest however will need another transition, which leads to the same final state through the inclusion of an = character.

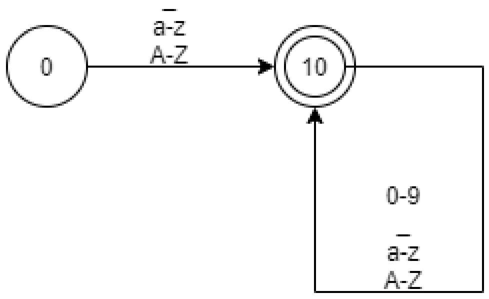
The division operator is missing from this section as it appears later on along with comments, since TeaLang makes use of C-Style comments featuring // and /* for multi-line comments.

The constructed DFA for these included operators is seen here.



Identifiers

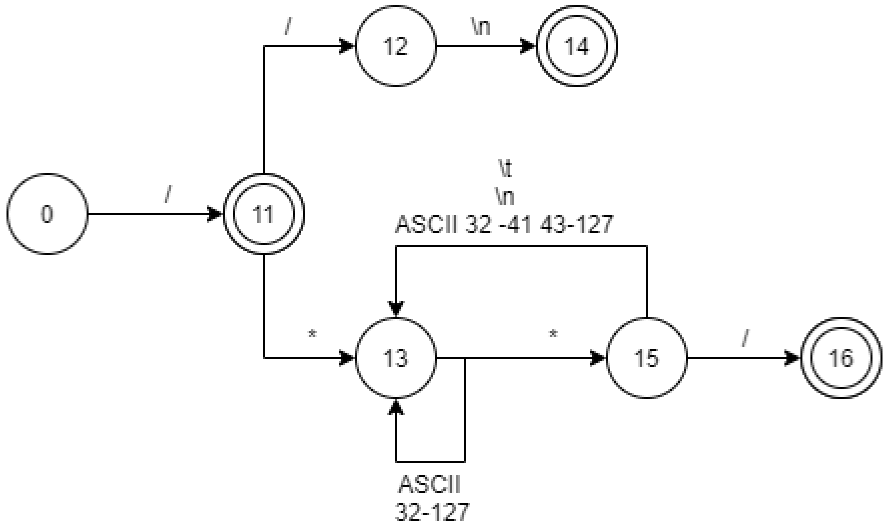
Letters can be all grouped together under a single DFSA for now. Future analysis on the extracted token outside of this DFSA will allow us to decipher whether the supplied string is a reserved keyword such as keywords used for types, loops, declarations and returns. Apart from these reserved keywords we can also declare our own variables and function names using any specified string of letters and underscored, followed by another letter, underscore or even digit, if the identifier does not start with a digit. So, our DFSA needs to be able to handle any repeated number of letters, underscores and after at least a single character , can also include any repetition of digits.



Comments

Comments as explained earlier are indicated by a starting /, and can be followed by either a single / indicating a comment which last until the final \n encountered at the end of a line, or followed by a * which indicates a multi-line comment that spans over multiple lines until the */ characters are encountered denoting the end of the multi-line comment sections.

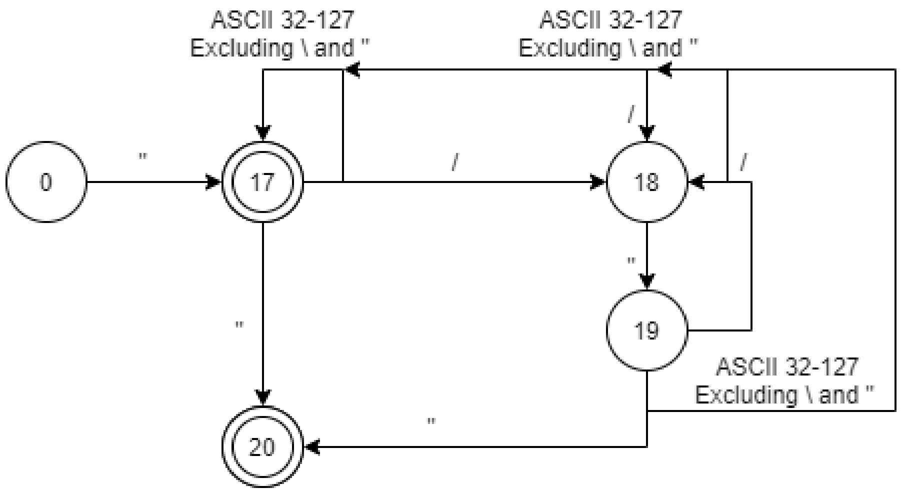
Apart from these we must remember to include the / on its own as a separate final state to account for the missing operator and allow the lexer to recognize it as a division operator. Comments can include any possible character, and setting single characters manually would not be feasible, so instead we make use of character ASCII values to denote the range of set of characters which can be followed in a comment.



Strings

Finally, we have to account for String literals, these would be any sequence of characters that can be held within a String variable enclosed between two inverted commas, “. Initially the plan was to include escape characters such as \t to indicate a tab, \n for a new line, and \" to hold the string identifier character, this functionality was implemented at the lexical analysis stage however was later forgotten.

The implemented DFSA accounts for escape characters through the \ character and diverts the DFSA to an intermediary state with its own unique transition and handling. Illustrated below is the DFSA which handles String literals.


ASCII 32-127 Excluding \ and "
ASCII 32-127 Excluding \ and "
0
"
17
/
18
/
/
"
19
ASCII 32-127 Excluding \ and "
"
"
20

Auxiliary States

Additionally, I have included some auxiliary states which help in the analysis of the file. Bracket characters, commas, Colons and Semi-Colons were added as a single character transition to a final state 21, and additionally a special EOF character will be placed at the end of a file to denote the end of the file, this will also have a specialised final state 22, that allows a single transition to it to denote that we have reached the end of the file. The functionality for including tabs within multi-line comments was also added and represented through the transition table for a specific \t character.

With all these components we have constructed DFSA, and the necessary states and transitions to handle all possible chracter inputs and sequences recognised by our language, finally we connect all of these components to a common start state 0, which is featured in all of these drawings and extend the transitions and definition of the DFSA, to include all of these components in conjunction.

Table-Driven Approach

Now that we have realised our DFSA for the characters within our language, we must encode it as a process that would allow us to compute whether the supplied character sequence contain valid or invalid tokens.

The table-driven approach simulates the functions of a DFSA by first constructing a table, that holds mxn values, n will be the total number of states within our DFSA, and m will be the total number of unique character types, not token types. The process of populating the table goes like this.

Start at column 0, this will represent our start at state 0. Identify the characters allow a valid transition from state 0 to any other transition and populate their table-value at column 0 with the resulting state following the transition from the current state with that character. Taking for example the above String literal DFSA, at location [0][“] we would find the value 17, as starting from state 0 and following a “ we arrive at state 17. All other characters which do not include a valid transition are left untouched or set to -1, to indicate an error state since we do not have a state with a negative value. In our implementation, this table would be represented as a 2-Dimensional Array, the first index corresponding to the current state, and the second index corresponding to the character type, obviously we will have to convert these character types into a form of integer id so that we can efficiently call the table and check the transition that the current character from the current state gives us.

When fully constructed, the transition table in its final form as a 2D Array was this.

```
{
{ 1, 1, 3, 3, -1, -1, -1, -1, -1, -1, 10, -1, 12, 13, -1, 13, -1, 17, 17, 17, -1, -1, -1 }
{ 2, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 12, 13, -1, 13, -1, 17, 17, 17, -1, -1, -1 },
{ 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 12, 13, -1, 13, -1, 17, 17, 17, -1, -1, -1 },
{ 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 13, 12, 15, -1, 15, -1, 17, 17, 17, -1, -1, -1 },
{ 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 12, 13, -1, 13, -1, 17, 17, 17, -1, -1, -1 },
{ 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 12, 13, -1, 13, -1, 17, 17, 17, -1, -1, -1 },
{ 8, -1, -1, -1, -1, -1,  9,   9,   9, -1, -1, -1, 12, 13, -1, 13, -1, 17, 17, 17,  -1, -1, -1 },
{ 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, 10, -1, 12, 13, -1, 13, -1, 17, 17, 17, -1, -1, -1 },
{ 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 12, 12, 13, -1, 16, -1, 17, 17, 17, -1, -1, -1 },
{ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 12, 13, -1, 13, -1, 18, 18, 18, -1, -1, -1 },
{ 17, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 12, 13, -1, 13, -1, 20, 19, 20, -1, -1, -1 },
{ 21, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 12, 13, -1, 13, -1, 17, 17, 17, -1, -1, -1 },
{ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 14, 13, -1, 13, -1, -1, -1, -1, -1, -1, -1 },
{ 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, 10, -1, 12, 13, -1, 13, -1, 17, 17, 17, -1, -1, -1 },
{ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 12, 13, -1, 13, -1, 17, 17, 17, -1, -1, -1 },
{ 22, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 },
{ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 13, -1, -1, -1, -1, -1, -1, -1, -1, -1 }
}
```

Columns and column number correspond to the current state that had been reached by the previous states, whilst the different rows correspond to the different characters.

The order of character types, corresponding with the columns is as follows:

Digits, Dot, Addition Operators, Asterisk, Exclamation, Comparison, Equals, Underscore, Forward Slash, Back Slash, Inverted Commas, Brackets and Commas and Colons (these were grouped together as they were a handful of single character tokens), \n, Letters, Printable Characters (these include many of the previous but as well many that have not been included like space characters) , End Of File, \t.

Lexer()

Within the Lexer Class Java File, we first take the supplied .txt file and append a \n at then of each line, and an EOF character at the end of the file. This is done so that for the case of single-line comments we can recognize the end of the line, as well as being able to recognise the end of the file and terminate the search for the next character.

After, our tokenization algorithm starts, and follows this process.

If we are currently only looking at blank space, tabs or new-lines and have not encountered a character yet, skip to the first encountered character. Upon encountering a new character, transition to a new state by calling a transition function on the implemented table, and passing the current character and state as arguments, from there we recognize the resultant state that has been transitioned by this character. If this resultant state is -1, so therefore an erroneous transition, check if the current state is a final state, if it is not a final state terminate the algorithm and report a lexical error, if it is a final state, save the sequence of characters right before the current and construct a token from that sequence, start the algorithm again from the character that caused this negative state transition. If the resultant state is any other number, append the character to the current lexeme which we are building, and update the current state to that number which ahs been returned by the valid transition.

When we have terminated an iteration and found the sequence of characters belonging to a single character, we create an instance of the Token Class and assign the following variables.

String type – The type of token, depending on the last final state encountered, our type of token is decided, there cannot be tokens of the same type coming from different final states.

String value – The actual sequence of characters that belong to the token.

Int line_number – The line number at which this token has been encountered.

The full list of token types, and their corresponding final states are retrieved by the getType() function within the Token class, which acts on a switch statement on the final state. These will be included in Appendix 1.

After processing every character, each Token object will be created and placed in a list, and at the end this list will be returned back to the main() function so that it may be passed on to the Parser for Syntax Analysis.

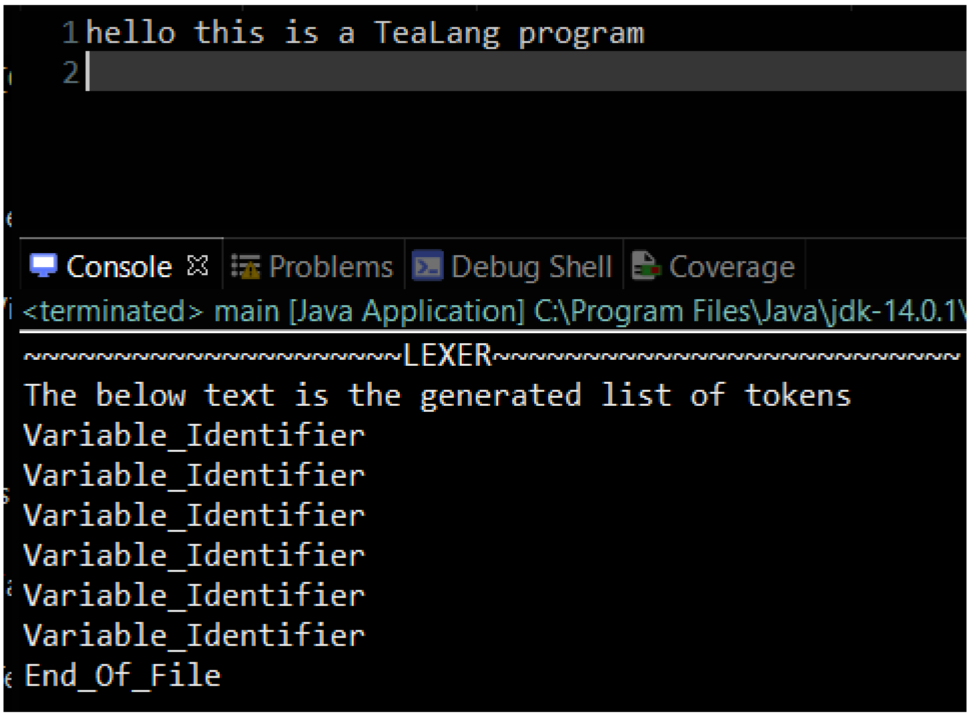
Testing

Testing this component was done by supplying both valid and invalid TeaLang example files, so that all tokens can be identified as being successfully extracted. There was no need for correct syntax, but rather making sure that when specific keywords were written, they were being extracted successfully as correct token types.

This was evaluated by outputting the contents of the final ArrayList and looking at what token types have been included, and if any are missing.

Here are two examples:

Example 1:

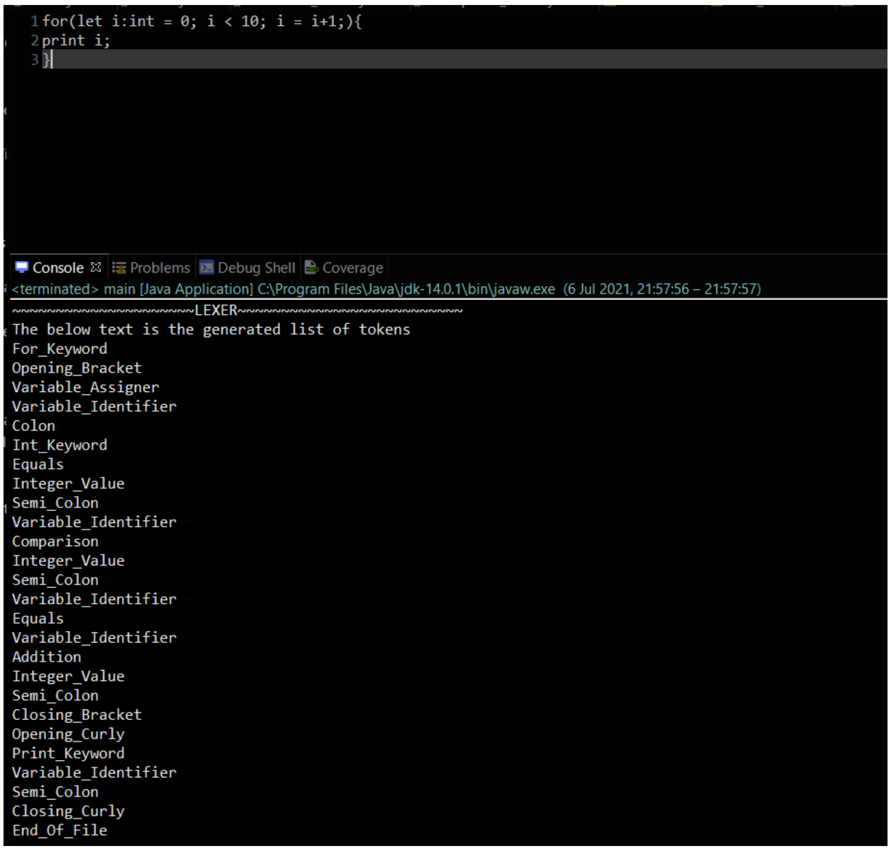


```
hello this is a TeaLang program
```

```
<terminated> main [Java Application] C:\Program Files\Java\jdk-14.0.1\
~~~~~~~~~~~~~~~~~~~~~~~~~~~LEXER~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
The below text is the generated list of tokens
Variable_Identifier
Variable_Identifier
Variable_Identifier
Variable_Identifier
Variable_Identifier
Variable_Identifier
End_Of_File
```

Supplying a sequence of plain text, produces the intended effect of each separate word being extracted as a variable identifier.

Example 2:



```
for(let i:int = 0; i < 10; i = i+1;){
print i;
}
```

```
<terminated> main [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (6 Jul 2021, 21:57:56 – 21:57:57)
~~~~~~~~~~~~~~~~~~~~~~~~~~LEXER~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
The below text is the generated list of tokens
For_Keyword
Opening_Bracket
Variable_Assigner
Variable_Identifier
Colon
Int_Keyword
Equals
Integer_Value
Semi_Colon
Variable_Identifier
Comparison
Integer_Value
Semi_Colon
Variable_Identifier
Equals
Variable_Identifier
Addition
Integer_Value
Semi_Colon
Closing_Bracket
Opening_Curly
Print_Keyword
Variable_Identifier
Semi_Colon
Closing_Curly
End_Of_File
```

The above TeaLang sample also produced the correct Tokens, seeing a more syntactically correct example also helps illustrate validity of the implemented Lexer and in all of its possible Token Generations.

Testing for each specific Token was also done, and proved to return the expected results.

Hand Crafted LL(k) Parser

The next component to be implemented is the Parser. Following from the Lexer, and its resultant list of token types, we can now perform syntax analysis on the generated list and check to see whether the list of tokens are presented in the right order according to the syntax and grammar rules of the TeaLang programming language, provided by the EBNF.

The type of Parser that was implemented, was a Predictive Parser, in which based on the current token and reserving space for a single lookahead token, we can predict what the next token types will be according to the syntax of the program. To be able to this we must construct a Parse Table out of First and Follow Sets of Non-Terminal Symbols, and the associated Terminal Symbols. The Parse Table dictates what type of non-Terminal Symbol can be accepted as "input" when awaiting a certain non-Terminal Symbol.

For example, if we are expecting a Variable Assignment Statement, the non-Terminal Symbol for Variable Assignment accepts only the "let" Terminal Symbol, if we have predicted that a Variable Assignment comes next in the syntax, and we receive any other Terminal apart from the "let" keyword, then we know that there lies a Syntax Error within the input program.

Constructing the Parse Table

Before we set out to identify the FIRST and FOLLOW sets, we must first identify what Terminal and non-Terminal symbols are represented in our language. This is done by taking a look at the EBNF and performing the following check.

If the current term can be expressed as a combination of other terms within a production rule and be written out in terms of other symbols, then it is a Non-Terminal. An example of this is the Block Symbol, which can be represented as a repetition of Statement Symbols.

〈Block〉::= '{' { 〈Statement〉} '}'

Another Example would be the Statement Symbols themselves.

〈Statement〉::= 〈VariableDecl 〉';'

| 〈Assignment〉';'

| 〈PrintStatement〉';'

| 〈IfStatement〉

| 〈ForStatement〉

| 〈WhileStatement〉

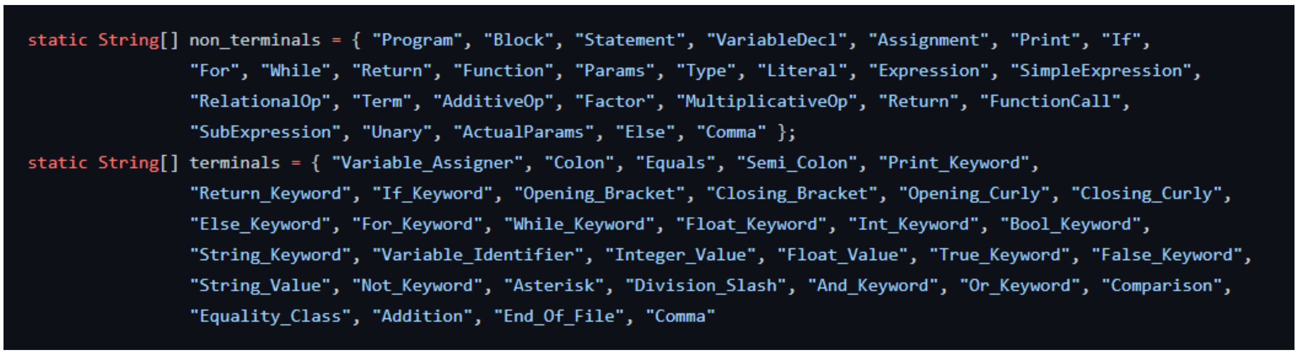
| 〈RtrnStatement〉';'

| 〈FunctionDecl 〉

| 〈Block〉

A Terminal Symbol on the other hand would be a Symbol, at its lowest level which cannot be represented in terms of other symbols, such as literal values, operators, brackets and semi-colons, specific keywords like print, if or else. We do not include that these keywords can then be represented as a sequence of characters, as we have tokenised these characters and so our we only consider the EBNF that represent to this point, after the Tokenization process.

The final Generated list of terminal and Non-Terminal Symbols is the following:



```
static String[] non_terminals = { "Program", "Block", "Statement", "VariableDecl", "Assignment", "Print", "If",
                "For", "While", "Return", "Function", "Params", "Type", "Literal", "Expression", "SimpleExpression",
                "RelationalOp", "Term", "AdditiveOp", "Factor", "MultiplicativeOp", "Return", "FunctionCall",
                "SubExpression", "Unary", "ActualParams", "Else", "Comma" };
static String[] terminals = { "Variable_Assigner", "Colon", "Equals", "Semi_Colon", "Print_Keyword",
                "Return_Keyword", "If_Keyword", "Opening_Bracket", "Closing_Bracket", "Opening_Curly", "Closing_Curly",
                "Else_Keyword", "For_Keyword", "While_Keyword", "Float_Keyword", "Int_Keyword", "Bool_Keyword",
                "String_Keyword", "Variable_Identifier", "Integer_Value", "Float_Value", "True_Keyword", "False_Keyword",
                "String_Value", "Not_Keyword", "Asterisk", "Division_Slash", "And_Keyword", "Or_Keyword", "Comparison",
                "Equality_Class", "Addition", "End_Of_File", "Comma"
```

FIRST and FOLLOW Sets are then generated in this manner.

For FIRST sets, take each Non-Terminal Symbol and look up the EBNF function responsible for its syntax. At each rule, the first Terminal Symbol encountered at every branching path, optional construction method , and syntax separated | , is part of the FIRST set for that non-Terminal. If a non-Terminal is encountered, then traverse the Production Rule for that non-Terminal, and return its FIRST set, as well to be included as a part of the initial non-Terminal symbol's FIRST set.

FOLLOW sets consider only the Non-Terminal Symbols that may be presented as optional in the syntax. For example when constructing a sequence of operations, the list of operations is optional and may be of any arbitrary length. There are no restrictions imposing the syntax that the next token must only be one that is in the FIRST set, but rather the non-Terminal Symbol itself is optional. In this case we see what types of terminal symbols may follow, hence the name, the non-Terminal Symbol, and if any of these elements from the FOLLOW set are encountered, then this would indicate that the current non-Terminal Symbol was optional and has been "skipped" from our predictive process. The follow sets are generated by looking at the terminal or non-terminal symbol that is first encountered after escaping the conditional-block containing the non-terminal in question.

Illustrating by example:

〈ForStatement〉::= 'for' '(' [〈VariableDecl 〉] ';' 〈Expression〉';' [〈Assignment〉] ')' 〈Block〉

In the above EBNF, we see that the non-Terminal symbol, VariableDecl, is optional and the terminal symbol which immediately follows it is the Semi-Colon, so when constructing the predictive parsing for a for-loop, and we have reached the VariableDecl statement, if we encounter a Semi-Colon instead of the "let" keyword, we are instructed that the current statement has been omitted, and we move on to the next token after the Semi-Colon.

I would like to make mention of the fact, that in the current implementation of the Parser, I had misread the EBNF rule for the For Loop and set it that Variable Declaration and Assignment are not optional , but must be included with the statement otherwise a syntax error will be raised. The reason for this was simply because I had realised too late, when writing this report in fact, that I have misread the EBNF. Another deviation from the EBNF was the removal of redundant Blocks, that is if the program encounters a sequence of {} with no tokens in between, these blocks are essentially redundant and a syntax error is shown. This was added so that escaping from nested blocks could be implemented more easily, in a way that upon encountering a closing curly, we immediately escape to the outer block. Some troubles were initially discovered when nested blocks with redundant blocks were passed as an input program, raising errors due to these redundant blocks however “solved” the problem without causing any issues.

After we configure the FIRST and FOLLOW sets for each non-Terminal symbol, we construct a table, or in this case a 2-D array, where each unique co-ordinate corresponds to a different pairing of terminal and non-terminal symbol, similar to how the table was implemented in the Lexer. If a Terminal symbol is not within the FIRST or FOLLOW sets, then populate the corresponding index as 0, otherwise set it equal to the production rule number, so that the syntax may be predicted based on that particular code later on.

The entire table was populated in this manner manually, by going through each non-Terminal symbol, figuring out its FIRST and FOLLOW sets, and setting the values of the corresponding Terminal symbols within the table equal to the production rule case number, so that the syntax may be constructed based on that rule.

When implementing the table itself, the values themselves corresponded to a case rather than a specific EBNF, this is so that I could avoid numerous entries within the table having the same value, and referring to the same production rule. This made debugging and testing significantly easier as the specific production rule code that fires would be unique for every different state.

The Parse Table is included in Appendix 2.

Parsing Algorithm

The parsing algorithm utilises a stack, which houses the current predicted symbols. After receiving a token from the Lexer, the stack is popped , and then we push onto the stack the symbols that have been predicted depending on the production rule that has fired.

We initialise the stack with a Program, symbol and a terminating $. Then the next two tokens from the generated sequence are retrieved from the list supplied by the Lexer. We then lookup the value in the parse table, by looking at the index corresponding to the current content of the stack on the top, and the current token type. During this process, we ignore single and multi-line comments since they have no effect on the syntax, no matter wherever they are within the program.

If the current contents of the stack is a terminal symbol, we check the type of token, if they match then we pop the stack, and move on to the next token. Otherwise we report a syntax error, as since we are expecting a terminal symbol, then it is 100% that symbol that must be presented next in the sequence, similar to presenting a semi colon at the end of a statement, the same principle is being applied here behind the scenes.

If the current top of the contents of the stack is a non-terminal symbol, then we retrieve the production rule code from the parse table, if it is 0 then this indicates that the supplied token is neither in the FIRST nor in the FOLLOW set, and is a wrong syntax, and then a syntax error is raised. If the number retrieved from the table, we apply the appropriate case, by performing f a switch statement on the retrieved value from the parse table, and update the contents of the stack depending on the type of production rule.

This process continues until we reach the EOF token. If after we parse the EOF token, we are left with only the $ at the top of stack, then we have successfully predicted that the program has reached its natural conclusion in accordance to the syntax, this could have been implemented by making the $ as a symbol which accepts the EOF Token, but they reach the same conclusion, one finishes with an empty stack and another finishes with only the $ symbol.

Building the AST

During the process of updating the contents of the stack, with the predicted syntax elements we are also simultaneously building an AST that represents the input program. The AST has been implemented as a tree data structure, using the single AST Class, and having recursive child AST Objects nodes as variables which hold other AST nodes.

Instead of having different instances of inherited AST Objects, a single AST Objects holds the node_type variable which lets us determine what type of node we are currently examinining. AST Objects also have a parent pointer, pointing to the parent AST Object, a value which holds the value held within the token, such as the case for literals and variable or function names. AST Objects also have an ArrayList of other AST Objects, which are its children within the represented structure.

The AST stars off by a single root AST Object, with the type of ProgramNode, then depending on the cases which are fired from retrieving the parse table, similar to how we update the contents of the stack in accordance to the syntax and the current case, we similarly build the AST by updating the root node at different hierarchies depending on wether we have entered or exited statements, entered different blocks, are need to represent information belonging to a single StatementNode.

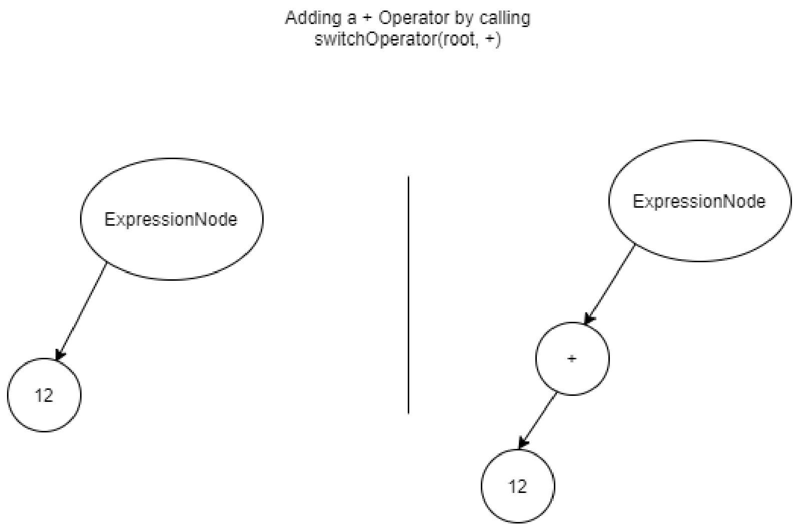
Some other methods have been designed to better help when dealing with the construction of the AST and these methods are as follows:

Three Constructors, one for creating a normal Node, one for creating a leaf Node that also holds the value of the token, in the case of Function Calls this intermediary node also houses a value, and a Copy Constructor used to create deep copies of AST Objects within other functions referencing other AST Objects.

addNode(), which takes the token type and optionally the attached value and adds a AST Object to the current root node, by adding the object to its child directory , and updating the newly created object's parent to be the root.

switchRoot(), switches the root of the current sub-tree to be the newly added child of the current root. This shifts the focus of all other operations such as adding new nodes, to be centralised around this new child. This methods is called when entering a new Block or Statement so that the next nodes are added to this child instead of the original sub-tree root.

operatorSwitch(), takes the current root and the current operator as arguments, when adding expressions we are building their syntax in the AST , as an expression tree, so that when evaluated in a Post-Order traversal, the original expression as t was originally written in the program can be extracted, therefore when we add an operator to the the expression tree, we must switch the current child with the operator, and add the old child to the operator so that the result will be that the operator has been inserted in place of the original value, and the original value has gone one level lower. Illustrated below is the result of the method, as it is better explained when seeing the operation visually.



This function also then sets, the operator as the new root, so if the expression was 12+5, the 5 node would be added as the child of the + node, so in post-order traversal we would retrieve 12+5.

orderOfOperations(), returns a similar results, but depending on the hierarchy of operators; Relational Operators should be on top, followed by Additional Operators, and at the bottom we should only find Multiplicative Operators. Depending on the type of Operator, we recursively switch the root, until we find the optimal position within the hierarchy of operators on which to call, switchRoot().

expressionEscape(), from the current root, this method is called when we have reached the end of an expression, there are no more operators or terms to be added and the program will move on to another statement, expression or parameter etc. The root however is still currently in the last position of the tree, ready to add the next operator or value to the expression tree, so this method recursively sets the parent node as the root, until we have exited the expression Node, and can resume adding the remainder of the program at the correct interval past the expression node.

Whilst building the actual tree as a data structure is tedious, the benefits of having such an approach will be having full control of the AST itself, and will make implementing Visitors much easier. For example when implementing the XML Visitor, the functionality needed to only be added once, and it was never touched again or updated as it worked with any new additions that might be included to the AST Structure that we never there before.

The entire set of cases for the Predictive Parser, updating the contents of the stack and building the AST nodes along with it will be included in Appendix 3.

Testing for both the Parser and the resultant AST happening in conjunction by displaying the resultant AST, in XML form using a Visitor class. This was implemented as Task 3 for this Assignment, and all testing for this section carried over to Task 3.

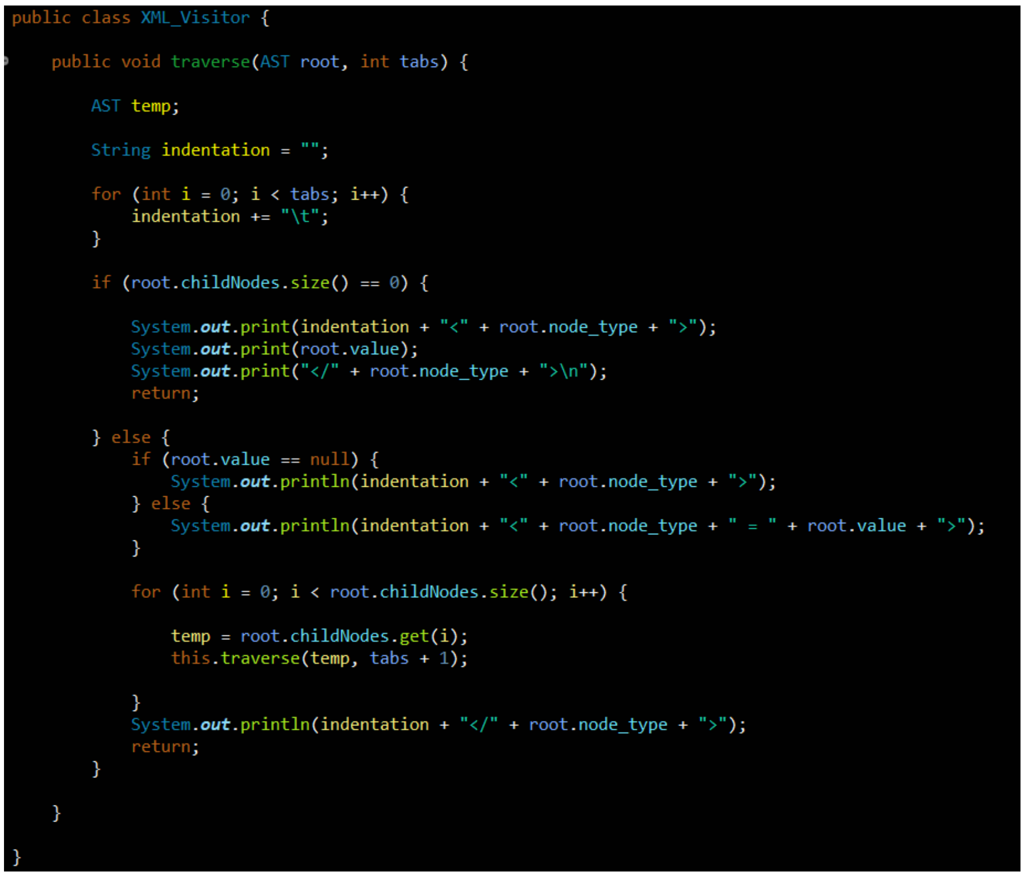
AST XML Generation Pass

Since we have implemented the AST as an independent data structure, the Visitor Classes had to be implemented in a different manner. This way , I was able to implement my own version of a Visitor Class, that features many of the functionalities offered by the ones shown in the example, but rather through the use of different inherited Object states, and with overloaded visit and accept functions, all calculations and necessary processing is done on single AST Object instances and the attributes allocated to them and their children.

In the case for XML Generation, an advantage which my own implementation provided was, that once it had been implemented, it never had to be changed again. Even after implementing different additions in Part 2, the XML Visitor class code left completely untouched, as only the building of the relevant nodes in the AST Structure had to be changed, which was still something that had to be tackled within the Parser Phase. Moreover, the implementation for generating an XML Representation of an AST was sufficiently easy to implement, since we are dealing with an actual tree structure from our root object.

The different levels and hierarchies of nodes in the tree can be represented and taken count of, by keeping track of how deep in the tree the current node is in, this can be done by either counting how many nodes away the current node is from the root, or by passing an argument3 tot the traversal function that indicates the current level within the tree. This value, would then be used too properly indent our representation, allowing different indentations at different child nodes , of sub root within subtrees etc.

The entire implementation of XML Generation for an AST, was provided through a recursive traversal function, which can be viewed in its entirety here:



```java
public class XML_Visitor {

    public void traverse(AST root, int tabs) {

        AST temp;

        String indentation = "";

        for (int i = 0; i < tabs; i++) {
            indentation += "\t";
        }

        if (root.childNodes.size() == 0) {

            System.out.print(indentation + "<" + root.node_type + ">");
            System.out.print(root.value);
            System.out.print("</" + root.node_type + ">\n");
            return;

        } else {
            if (root.value == null) {
                System.out.println(indentation + "<" + root.node_type + ">");
            } else {
                System.out.println(indentation + "<" + root.node_type + " = " + root.value + ">");
            }

            for (int i = 0; i < root.childNodes.size(); i++) {

                temp = root.childNodes.get(i);
                this.traverse(temp, tabs + 1);

            }
            System.out.println(indentation + "</" + root.node_type + ">");
            return;
        }

    }

}
```

We can see the current level of the tree being incremented with each method call of the traversal function, by calling the traversal method with the childNode as a new root and tabs+1 to indicate we have entered a childNode at a lower level. Depending on the level number within the tree, we print that many tabs, or \t, and we perform the following checks.

If the current node is a leaf node, output the node type tag, its value, and close the tag with the node type,and return to the previous method call.

If the current node is not a leaf, check wether it has an embedded value, if there is none, simply print the node tag, recursively call the function, on each of its child Nodes , passing an incremented integer to indicate a lower level, and after this loop terminates, print the closing tag for the node type. If the node in question does not have an embedded value associated with it, it does not output it.

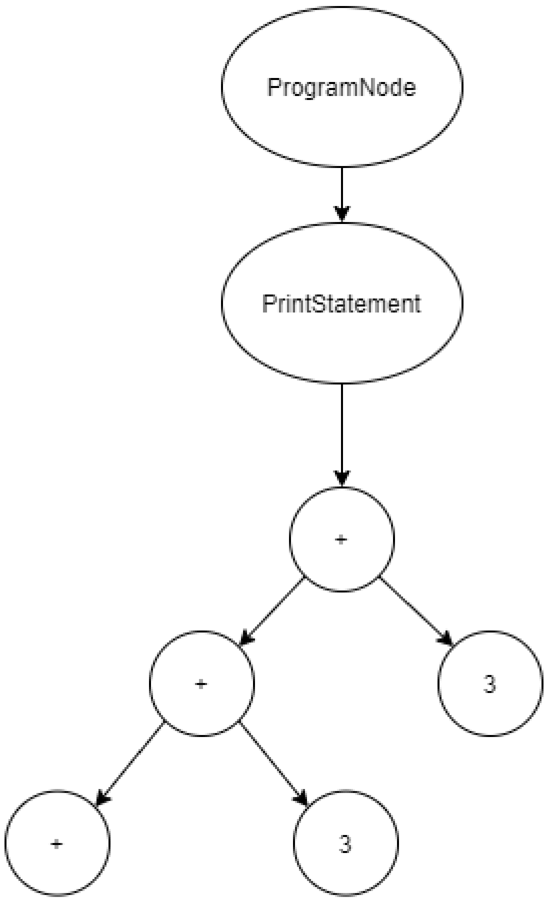
Testing

Testing the XML also lets us test the Parser as well as the AST generation, in this section I will illustrate some code examples, the expected AST, and the outputted XML Representation.

Test 1 – print statement and expression;

```
print 12+34+3;
```

Expected AST



Generated XML

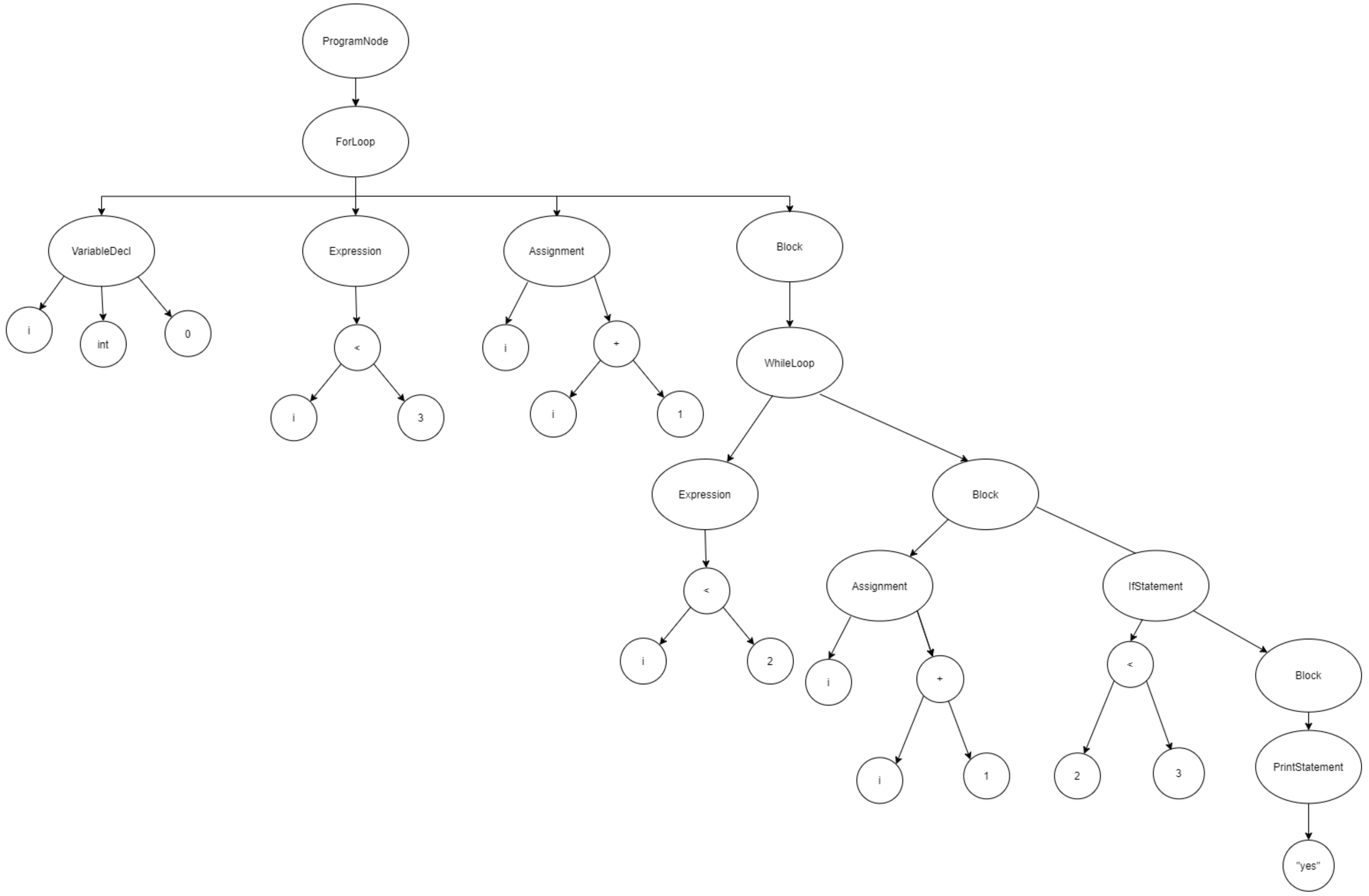
```
<ProgramNode>
	<PrintStatements>
		<Operator = +>
			<Integer_Value>12</Integer_Value>
			<Operator = +>
				<Integer_Value>34</Integer_Value>
				<Integer_Value>3</Integer_Value>
			</Operator>
		</Operator>
	</PrintStatements>
</ProgramNode>
```

Test 2 – For Loop, While and if

```
for(let i:int =0; i < 3; i=i+1;){

while(i<2){
i=i+1;
if(2<3){
print "yes";
}
}
```

Expected AST can be found on the next page.

Expected AST



Generated XML

```
<ProgramNode>
       <ForLoop>
              <VariableDecl>
                     <Variable_Identifier>i</Variable_Identifier>
                     <Type>int</Type>
                     <Integer_Value>0</Integer_Value>
              </VariableDecl>
              <Operator = <>
                     <Variable_Identifier>i</Variable_Identifier>
                     <Integer_Value>3</Integer_Value>
              </Operator>
              <VariableAssignment>
                     <Variable_Identifier>i</Variable_Identifier>
                     <Operator = +>
                            <Variable_Identifier>i</Variable_Identifier>
                            <Integer_Value>1</Integer_Value>
                     </Operator>
              </VariableAssignment>
              <Block>
                     <WhileLoop>
                            <Operator = <>
                                   <Variable_Identifier>i</Variable_Identifier>
                                   <Integer_Value>2</Integer_Value>
                            </Operator>
                            <Block>
                                   <VariableAssignment>
                                   <Variable_Identifier>i</Variable_Identifier>
                                          <Operator = +>
                                                 <Variable_>i</Variable_Identifier>
                                                 <Integer_Value>1</Integer_Value>
                                          </Operator>
                                   </VariableAssignment>
                                   <IfStatements>
                                          <Operator = <>
                                                 <Integer_Value>2</Integer_Value>
                                                 <Integer_Value>3</Integer_Value>
                                          </Operator>
                                          <Block>
                                                 <PrintStatements>

       <String_Value>yes</String_Value>
                                                 </PrintStatements>
                                          </Block>
                                   </IfStatements>
                            </Block>
                     </WhileLoop>
              </Block>
       </ForLoop>
</ProgramNode>
```

Slightly edited the tags, so that the XML Indentation can be preserved. This program is found in the example files and you can check out its integrity within the task.

From these tests, and other generated tests where I tested incorrect syntax and see if warning were generated, I concluded that the parser and the sufficient generated AST structure was fit for purpose to be used in the next stage, Semantic Analysis.

Semantic Analysis

Before this stage we only have worked out that the current input program consists of words that are within the defined language, and that these tokens are placed in the right order. However, we have not deduced wether or not the right combination of tokens make sense. While tokens may be syntactically correct, they might be semantically incorrect. This is the case for assigning wrong values to mismatching types, referencing variables that have not been declared yet or declaring a variable which has already been declared, we also check that function calls have matching parameter types, and that functions include a return statement.

Using scopes, we can keep track of what variables have been declared and their associated type. This functionality also extends to having localised and global scopes across different blocks that are created throughout the program. A scope is implemented by using a stack structure, upon encountering a variable declearation node , that variable and associated type is added to the scope that lies on top of the stack. When the program enters a new block, enclosed by curly brackets as is found in conditional statements and loops, a new scope is pushed onto the stack and any new variables that have been declared will instead be placed and the new scope that is on top of the stack. Furthermore, when the program encounters a closing curly, this indicates that the block has ended , and so we must forgot all variables that were being made use of within that scope, and so we push the top of the stack and remove those variables from memory.

This implementation will serve as a symbol table that associates variable identifiers with their expected type, and all operations including type checking will be dependent on wether the identifier is within the symbol table, and its corresponding value.

The data structures that are being used are a stack, to house all scopes, two maps which takes a String that holds a variable identifier and returns a string representing its type, one for variables and another for functions and their returns. Another map is included, which also takes a String representing the function identifier, but instead returns an ArrayList of the parameter types it expects in order.

Visitor Class

The Visitor Class traverses the generated AST structure by initially starting off at the root, and iterating through all its children. The children will always be a form of Statement node and the visitor will only check statement nodes since a program is a repated list of statement nodes, all other nodes that belong to a statement are either handled automatically by virtue of checking the statement itself or will traverse to it by a recursive function call if a new statement node is to be expected within its children such as in the case of functioin declaration, loops blocks etc. Conditional breaks have been implemented which divert the visitor to different routines depending on the type of statement node. The type of statement node is retrieved by checking the value of the node_type attribute for the child AST objects. Let us take a look at how value types are evaluated.

typeCheck()

The typeCheck function takes as arguments a subtree AST root, that will be the start of an <Expression> , and all of its associated definitions in accordance with the EBNF, and a string containing the expected type of the expression when fully evaluated. It will raise a semantic error if the evaluated type does not match with the expected type.

An expression can be presented in many different forms and so we perform different routines based on the passed AST node.

If the node passed represents a literal value, identifey by the following coniditions:

if (node.node_type == "Integer_Value" || node.node_type == "Float_Value"

|| node.node_type == "String_Value" || node.node_type == "True_Keyword"

|| node.node_type == "False_Keyword")

Then we return from the function if the expected type matches the literal that has been found, otherwise, we report a semantic error and exit.

If the passed node is a variable identifier, or a function call, we return the associated type found within the symbol table that belongs to that identifier. This function will always be called after first checking that the variable identifier in question exists within the symbol table.

If the passed node is neither of these, then it as operator belonging to an entire expression tree, and we must evaluate the entire expression by then recursively traversing this expression subtree in a post-order fashion.

This traversal is done by expressionOperationTraversal(), which is recrusively called on an AST node objects, and has three cases.

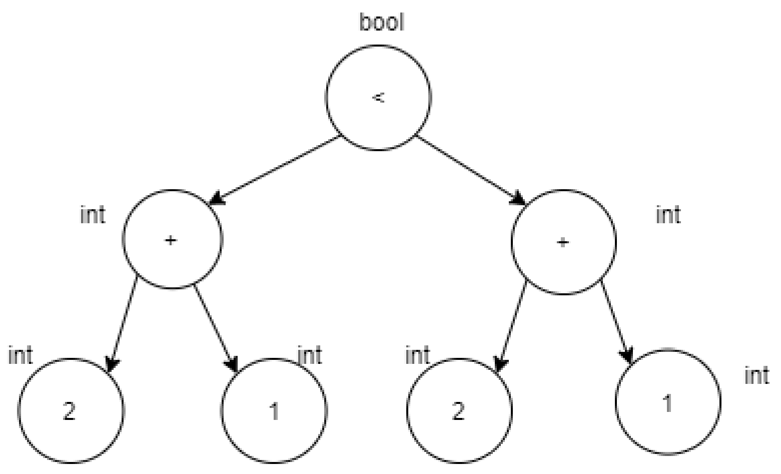
If the current root has 0 children, then perform the typeChecks previously mentioned and return the associated type, as I am writing this documentation it has just occurred to me that I could have used this single function all together and forego the previous typeCheck() function, but what happened during development was that I left expression to be the last case that I evaluate and test out and so was implemented at a later stage on its own. This can be seen as an improvement that can possibly be added to a future release of TeaLang!

If the current root has 1 child, then the current operator is Unary, all Unary operators do not cause a change in the type of a value when they are performed. For example, writing -5, still returns an integer results, and writing not true, still returns a bool results, so we call the function recursively on the childNode, if it is another unary operator, the function will keep recursively calling until the leaf node holding the value is encountered. There cannot be a unary operator which has two child nodes, since that would be syntactically incorrect and the parser would raise a syntax error.

If the current root has 2 children, first we traverse to the first child on the left and then on the second child on the left, and retrieve their types. At this point all the recursive calls occur, and when a both child nodes return the associated type, we check the current node value. Since the noded has two children then the current node is always an operator, we then retrieve the value of the operator when acted upon those two types, and return this value. If at any point, two child nodes return different types then a semantic error is raised as no operations are defined for typecasting variable types. This also eliminates the possibility of having a comparison between three elements such as 1<2<3, as this would evaluate to bool<int. The operator results are defined in the

checkOperatorConstraint() function which returns the resultant type when an operator acts on two different types, this therefore allows us to return a bool value, when comparing two variables of another type, and all the other necessary calculations.

Finally the resultant type from the operation is returned and we escape to the previous function call, and this process repeats with the new operator root node on top. An annotated AST example shows the different type evaluations, and how these types carry forward, on two different nodes where there are no discernable values but there are instead operator nodes.



Now that we have defined how values' type shall be checked we can move on to analysing each AST statement node throught the visitor by traversing the tree and perfoming specific routines for every type of Statement Node.

Variable Decleration

For Variable Decleration, we first check to see if the variable we are declaring exists by attempting to look up its identifier within the symbol table, and if it exists we report an error. Otherwise we move on to evaluating the value and checking if the type from the assigned expression matches the type we are assigned to the variable

Blocks

In this case, we take a look at all the statements that produe a new scope by entering a new block enclosed by curly brackets. These statement nodes are the loops, if and else, and block statement nodes. If the current node type is an if or while statement node, then we check to see if the expression that holds the condition for the block evalues to a Boolean, this way we raise errors if we encounter something like if(2+2), and allow only expression which evaluate to either true or false to be a part of a valid program.

Then we traverse to the following block node by recursively calling the same traversal function within the semantic visitor. This will then start the same process of iterating through its children nodes and evaluating any encountered Statement nodes, and if needed traversing to another nested block if any of these mentioned nodes are encountered.

Function Decleration

First we check to see if a function with the same name already exists by the symbol table, and if not we add a new entry to the current scope by accessing the function name, and its return which are found at index 1 and 0 respectively. We then recursively call the function again on the Block node which houses the statement contained within the function, so that we may analyse the contents of the function.

Return Statement

Initially we check to see wether a return statement, is found within a function decleration. This is done as to not allow return statements outside of function declerations, and so by recursively check the parent nodes of AST objects, until we either hit a function decleration node, or reach the root of the entire AST structure we can determine wether or not the current return statement is found within a function decleration.

If the return statement is indeed found within a function name, we retrieve the function return by looking up the symbol table with the function name, as we have just found the function declaration node, the function name is easily accessed. The type of the expression that is being returned is evaluated and checked to see if it is in line with the expected type of the function, set during the function initalisation.

Formal Params

We iterate through all the parameter names and perform the following check, has the current variable identifier already been declared within the current scope ? If yes return an error, so that no parameters with the same name can be allowed, otherwise append the function header map, and append that variable's type to the arrayList so that we can later on recognize that a function call requires this type of parameter in this order.

Variable Assignment

We check if the variable we are assigning has been declared before, if not we are referencing a variable that does not exist yet and raise an error. If it exists, we check that the evaluated type of the expression we are assigning to the variable is in line with the expected type of the variable found within the symbol table.

Print Statements

For print statements we perform no checks if the expression we are printing is a literal, if it is a variable we check if the variable exists, for function calls we traverse to the AST node so that the function call case described later may be applied, and finally if it is an Operator we call the expressionTravel function to evaluate its type.

Function Calls

For function calls, we first check that the function being called exists, then we iterate through all parameters, check their evaluated type and look up the arraylist from function_headers to see if that is the next variable type we are expecting from the list of parameters.

Misc

A miscellanoues case has been added to evaluate the expression function for a for loop condition

Testing

Testing was done by supplying correct and incorrect test examples and checking to see if the intended conclusions have been reached. Contents of the symbol table, as well as both function headers were printed to the console at different traversal function calls so that I may evalue the contents of the stack as well as what currently being analyzed during the semantic analysis process. Further proof will that semantic analysis works as intended will be shown when we are finally evaluating the input program, and outputting its results as the interpreter visitor is mostly based around the same architecture that the semantic analysis visitor makes use of, and exhibits almost identical cases to retrieve and check for variables.

If after traversing all nodes, no errors have been reported, then the input program is of a valid semantics and there are no wrong declerations, variable or function calls and type evaluations. This allows us to move on to the next and final stage of Interpreting the program and returning results to finally execute the program.

Interpreter Visitor

For the interpreter visitor, as previously mentioned, we implemented a similar recursive traversal function that visits every statememtn node and this time retrieves the values from variables, expressions, and function calls. When declaring function we must also store the Block node within the function declaration, as this will serve as our point of reference to enter the function when the interpreter encounters a function call.

In addition to the previous symbol table we also now, make use of a value table to keep track of declared variables and the current value associated to them.

Many of the node cases here are identical to the cases and operations performed in the previous stage, however they now include retrieving the value and not just the type. Some other key differences are made which I will now go into detail with.

getValue()

The get type function has been revamped to return both the type and value of an expression, literal, variable or function call in the form an array list of two elements. The same architecture as the previous getType() function can still be seen here, where each case is first checked, otherwise if we encounter an operator we must perform a post-order traversal on the expression tree returning the value and type at each recusive level.

A key difference when evaluating expression is that now we have defined what operations on values of certain types perform. What this entailed is manually, checking the operator, and if the current values area of a certain type, we return the calculation based on the operator and those types. For example, if we the current operator is +, and we have integer values,
The string value is read, and converted to an int type, and we return the addition of the two int types as a String.

This process repeats as it is making its way back up to the root of the expression tree, with the final returned value being that of the full expression tree being evaluated.

Function Call

For Function Calls, the node containing the block of statements that represent the called function is identified by retrieiving it from the map, that maps a function name to the AST node. Then a new scope is pre-emptively added which will house the parameter values. The function call's parameters are evaluated, and their values are pushed onto the value_table along with the corresponding parameter, that is making reference of these values as seen from the function declaration. Essentially, if we are calling foo(1) on a function int foo(x :int), we are pushing x=1 onto the value table. The traverse function call is then called, on the loaded function node, so that the statements found within the function may be evaluated.

Print Statement

After evaluating the necessary values, variables or function calls, the print statement as expected prints the returned value, and automatically sets a new line so that the next print will be printed with the new line. No functionality has been added to print new lines if the string houses a \n or \t for tabs.

Loops, and If-Else

For loops and if-else statements that house a conditional expression that must evaluate to true before entering a block of statements, we evaluate this condition and depending on wether the conditiom returned through our evaluation is equal to true or false, we direct the flow of the interpreter to visit those statements. In the case of loops, the condition is checked and if it is true, we traverse to the block node, and this is repeated until we check the condition and it returns a false value. In the case of for-loops the assignment node is called before every iteration of visiting the block node and its children. For if-else statements, depending on the value of the expression we choose wether to visit the if-block, or the else-block if and only if it exists and the condition evaluates to false.

All other statements and cases behave as expected, where values are updated and appropriately set to the corresponding variables, or calls.

Testing

For testing, I will be providing the XML representation of the inputted program as well as the intended results. At this stage if the interpreter is outputting the expected results, than all previous components are working in conjunction together as expected as well!

Test 1 – powers

```
float Pow(x:float, n:int){

print "The function is going to compute the power of:";
print x;
print "Raised by";
print n;


let y:float = 1.0;

if(n > 0){

for(let z: int = n; z>0; z=z-1;){

y = y*x;

}

}
else
{



for(let z: int = n; z<0; z=z+1;){


y = y/x;
}

}
return y;

}

let x: float = Pow(6.4,3);


print x;

print Pow(5.0,-2);
```

The above function calculates the power of an integer, the expected results are to be outputted this way:

```
The function is going to compute the power of:

6.4

Raised by

3

262.144

The function is going to compute the power of:

5.0

Raised by

-2

0.04
```

Generated XML

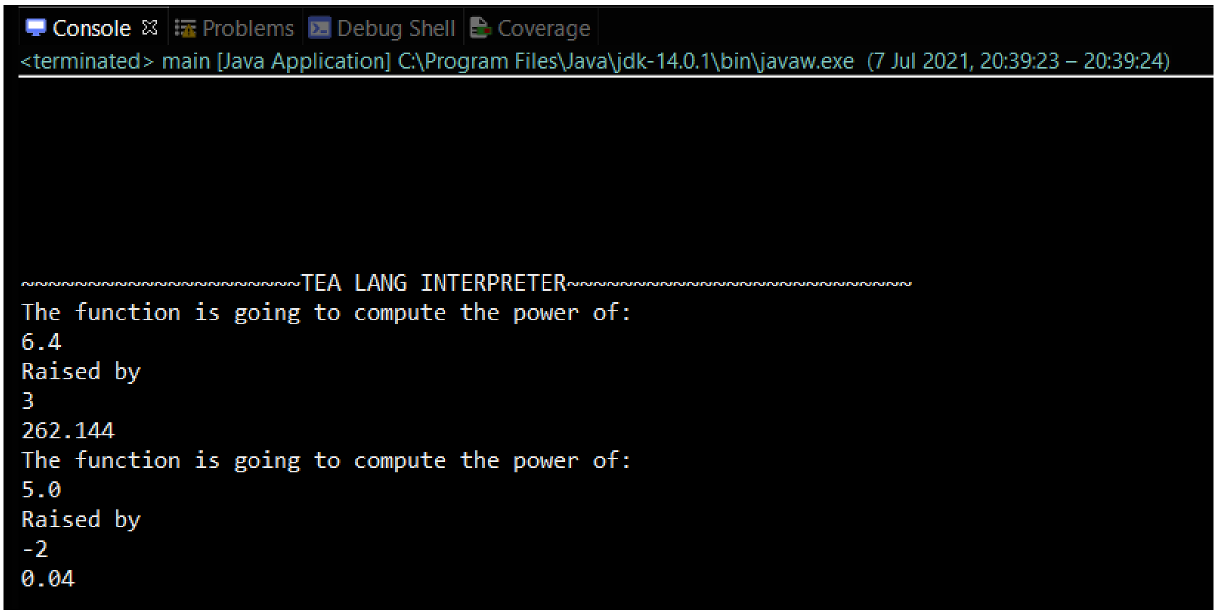
```
<ProgramNode>
	<FunctionDecl>
		<Type>float</Type>
		<Variable_Identifier>Pow</Variable_Identifier>
		<FormalParams>
			<Variable_Identifier>x</Variable_Identifier>
			<Type>float</Type>
			<Variable_Identifier>n</Variable_Identifier>
			<Type>int</Type>
		</FormalParams>
		<Block>
			<PrintStatements>
				<String_Value>The function is going to compute the power
of:</String_Value>
			</PrintStatements>
			<PrintStatements>
				<Variable_Identifier>x</Variable_Identifier>
			</PrintStatements>
```

```
            <PrintStatements>
                  <String_Value>Raised by</String_Value>
            </PrintStatements>
            <PrintStatements>
                  <Variable_Identifier>n</Variable_Identifier>
            </PrintStatements>
            <VariableDecl>
                  <Variable_Identifier>y</Variable_Identifier>
                  <Type>float</Type>
                  <Float_Value>1.0</Float_Value>
            </VariableDecl>
            <IfStatements>
                  <Operator = >>
                        <Variable_Identifier>n</Variable_Identifier>
                        <Integer_Value>0</Integer_Value>
                  </Operator>
                  <Block>
                        <ForLoop>
                              <VariableDecl>

<Variable_Identifier>z</Variable_Identifier>
                                    <Type>int</Type>

<Variable_Identifier>n</Variable_Identifier>
                              </VariableDecl>
                              <Operator = >>

<Variable_Identifier>z</Variable_Identifier>
                                    <Integer_Value>0</Integer_Value>
                              </Operator>
                              <VariableAssignment>

<Variable_Identifier>z</Variable_Identifier>
                                    <Operator = ->

<Variable_Identifier>z</Variable_Identifier>

<Integer_Value>1</Integer_Value>
                                    </Operator>
                              </VariableAssignment>
                              <Block>
                                    <VariableAssignment>

<Variable_Identifier>y</Variable_Identifier>
                                          <Operator = *>

<Variable_Identifier>y</Variable_Identifier>

<Variable_Identifier>x</Variable_Identifier>
                                          </Operator>
                                    </VariableAssignment>
                              </Block>
                        </ForLoop>
                  </Block>
                  <ElseBlock>
                        <ForLoop>
                              <VariableDecl>

<Variable_Identifier>z</Variable_Identifier>
```

```
                            <Type>int</Type>

    <Variable_Identifier>n</Variable_Identifier>
                        </VariableDecl>
                        <Operator = <>

    <Variable_Identifier>z</Variable_Identifier>
                            <Integer_Value>0</Integer_Value>
                        </Operator>
                        <VariableAssignment>

    <Variable_Identifier>z</Variable_Identifier>
                            <Operator = +>

    <Variable_Identifier>z</Variable_Identifier>

    <Integer_Value>1</Integer_Value>
                            </Operator>
                        </VariableAssignment>
                        <Block>
                            <VariableAssignment>

    <Variable_Identifier>y</Variable_Identifier>
                                <Operator = />

    <Variable_Identifier>y</Variable_Identifier>

    <Variable_Identifier>x</Variable_Identifier>
                                </Operator>
                            </VariableAssignment>
                        </Block>
                    </ForLoop>
                </ElseBlock>
            </IfStatements>
            <ReturnStatement>
                <Variable_Identifier>y</Variable_Identifier>
            </ReturnStatement>
        </Block>
    </FunctionDecl>
    <VariableDecl>
        <Variable_Identifier>x</Variable_Identifier>
        <Type>float</Type>
        <FunctionCall = Pow>
            <Float_Value>6.4</Float_Value>
            <Integer_Value>3</Integer_Value>
        </FunctionCall>
    </VariableDecl>
    <PrintStatements>
        <Variable_Identifier>x</Variable_Identifier>
    </PrintStatements>
    <PrintStatements>
        <FunctionCall = Pow>
            <Float_Value>5.0</Float_Value>
            <Operator = ->
                <Integer_Value>2</Integer_Value>
            </Operator>
        </FunctionCall>
    </PrintStatements>
</ProgramNode>
```

(Note that word is slightly skewing the indentation levels, because they are overlapping onto other lines, but the XML is generated properly indented tags within the code, powers.txt is left in the example file directory to see.)

Actual Result



```
<terminated> main [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe  (7 Jul 2021, 20:39:23 – 20:39:24)

~~~~~~~~~~~~~~~~~~~~~TEA LANG INTERPRETER~~~~~~~~~~~~~~~~~~~~~~~~~~
The function is going to compute the power of:
6.4
Raised by
3
262.144
The function is going to compute the power of:
5.0
Raised by
-2
0.04
```

Test 2 – Factorial

Input Program

```
int factorial(n : int){

if(n == 1){

return 1;

}else{

return n*factorial(n-1);
}
}


print factorial(10);
```

The expected output is that of 10! Which is 3628800

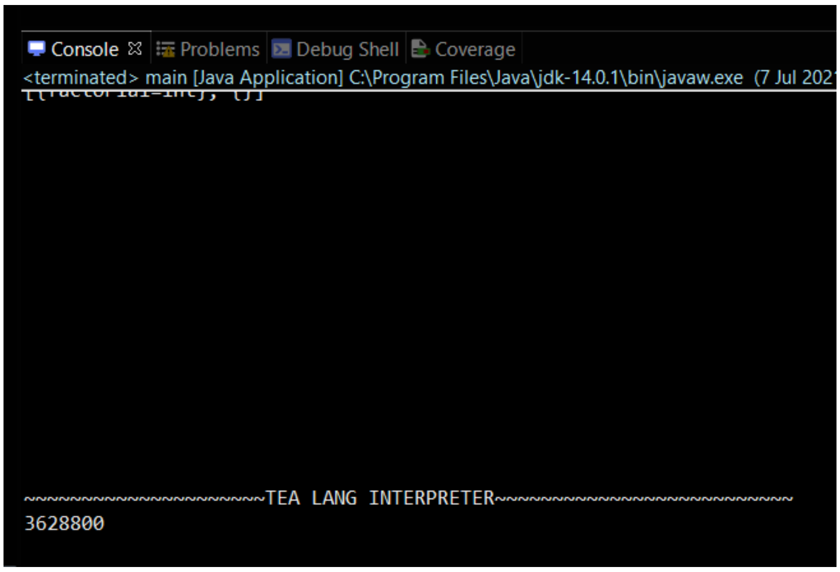
XML Generation

```
<ProgramNode>
       <FunctionDecl>
```

```
        <Type>int</Type>
        <Variable_Identifier>factorial</Variable_Identifier>
        <FormalParams>
            <Variable_Identifier>n</Variable_Identifier>
            <Type>int</Type>
        </FormalParams>
        <Block>
            <IfStatements>
                <Operator = ==>
                    <Variable_Identifier>n</Variable_Identifier>
                    <Integer_Value>1</Integer_Value>
                </Operator>
                <Block>
                    <ReturnStatement>
                        <Integer_Value>1</Integer_Value>
                    </ReturnStatement>
                </Block>
                <ElseBlock>
                    <ReturnStatement>
                        <Operator = *>
                            <Variable_r>n</Variable_Identifier>
                            <FunctionCall = factorial>
                                <Operator = ->
                                    <Var>n </Var>
                                    <Int>1</Int>
                                </Operator>
                            </FunctionCall>
                        </Operator>
                    </ReturnStatement>
                </ElseBlock>
            </IfStatements>
        </Block>
    </FunctionDecl>
    <PrintStatements>
        <FunctionCall = factorial>
            <Integer_Value>10</Integer_Value>
        </FunctionCall>
    </PrintStatements>
</ProgramNode>
```

(Again slightly modified some tags)

Actual Result



```
<terminated> main [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe  (7 Jul 202
[{factorial=int}, {}]

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~TEA LANG INTERPRETER~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3628800
```