Compiler Theory and Practice Course Assignment Part 1

Aiden Williams
Aiden.williams.19@um.edu.mt

Contents

Introduction

This first project describes the implementation and development of the TeaLang interpreter. The interpreter was coded in C++20 and built using CMake version 3.16. The TeaLang language follows the EBNF included on page 2. This document introduces the language and goes over each part of the interpreter and assignment in detail, providing source code explanation, design details, figures, tables, and test cases.

Test Cases

5 test cases are provided with the submission, the results of these cases are logged in this document. The test cases are:

Test Case 1 - Declaration and Scope Handling (Semantic Fail)

```
let x : bool = true;

if(x){
    let y : float = 10.0;
    print y;
}

print y;
```

Test Case 2 - Lexical Fail

```
let 10 : bool = true;

if(x){
    let 11 : float = .xyz;
    print 11;
}
```

Test Case 3 – Recursive Function Declaration and Call

```
float Factorial( x : float) {
    if (x >= 1){
            return x * Factorial(x - 1);
    }else{
            return 1;
    }
}

print Factorial(5.0);
```

Test Case 4 – Parser Fail

```
float Factorial( x : float)
    if (x >= 1)
            return x * Factorial(x - 1);
    else
            return 1;

print Factorial(5.0);
```

Test Case 5 – Semantic Fail

```
float Factorial( x : int) {
    if (x >= 1){
            return x * Factorial(x - 1);
    }else{
            return 1;
    }
}

print Factorial(5.0);
```

Test Case 6 – Large Program

```
float Square (x : float) {
    return x*x;
}

bool XGreaterThanY (x : float , y : float) {
    let ans : bool = true;
    if (y > x) {ans = false ;}
    return ans;
}
    bool XGreaterThanYv2 (x : float , y : float) {
    return x > y;
}
float AverageOfThree (x : float , y : float , z : float) {
    let total : float = x + y + z ;
    return total / 3.0;
}

string JoinStr (s1 : string , s2 : string) {
    let s3 : string = s1 + s2 ;
    return s3 ;
}

let x : float = 2.4;
let y : float = Square (2.5);

print y ;                                           //6.25
print XGreaterThanY (x , 2.3 ) ;                    // t r u e
print XGreaterThanYv2 (Square ( 1.5 ) , y ) ;       // f a l s e
print AverageOfThree (x , y , 1.2) ;               //3.28
print JoinStr("Hello" , "World") ;             // H e l l o World

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

let ii : int = 0;

while (ii < 10){
   print ii;
   ii = ii + 1;
}
```

For these 6 cases, I am expecting a return of 0 for the non-fail ones and the error code for the failed ones.

```
<Letter> ::= [A-Za-z]
<Digit> ::= [0-9]
<Printable> ::= [\textbackslash x20-\textbackslash x7E]
<Type> ::= `float' | `int' | `bool' | `string'
<BooleanLiteral> ::= `true' | `false
<IntegerLiteral> ::= <Digit> \{ <Digit> \}
<FloatLiteral> ::= <Digit> \{ <Digit> \} `.' <Digit> \{ <Digit> \}
<StringLiteral> ::= `"' \{ <Printable> \} `"'
<Literal> ::= <BooleanLiteral> |
              <IntegerLiteral> |
              <FloatLiteral>   |
              <StringLiteral>
<Identifier> ::= ( `\_' | <Letter> )  \{ `\_' | <Letter> | <Digit> \}
<MultiplicativeOp> ::= `*' | `/' | `and'
<AdditiveOp> ::= `+' | `-' | `or'
<RelationalOp> ::= `<' | `>' | `==' | `!=' | `<=' | `>='
<ActualParams> ::= <Expression> \{ `,' <Expression> \}
<FunctionCall> ::= <Identifier> `(' [ <ActualParams> ] `)'
<SubExpression> ::= `(' <Expression> `)'
<Unary> ::= ( `-' | `not' ) <Expression>
<Factor> ::=  <Literal>       |
              <Identifier>    |
              <FunctionCall>  |
              <SubExpression> |
              <Unary>
<Term> ::= <Factor> \{ <MultiplicativeOp> <Factor> \}
<SimpleExpression> ::= <Term> \{ <AdditiveOp> <Term> \}
<Expression> ::= <SimpleExpression> \{ <RelationalOp> <SimpleExpression> \}
<Assignment> ::= <Identifier> `=' <Expression>
<VariableDecl> ::= `let' <Identifier> `:' <Type> `=' <Expression>
<PrintStatement> ::= `print' <Expression>
<RtrnStatement> ::= `return' <Expression>
<IfStatement> ::= `if' `(' <Expression> `)' <Block> [ `else' <Block> ]
<ForStatement> ::= `for' `(' [ <VariableDecl> ] ';' <Expression> ';'
                  [ <Assignment> ] `)' <Block>

<WhileStatement> ::= `while' `(' <Expression> `)' <Block>
<FormalParam> ::= <Identifier> `:' <Type>
<FormalParams> ::= <FormalParam> \{ `,' <FormalParam> \}
<FunctionDecl> ::= <type> <Identifier> `(' [ <FormalParams> ] `)' <Block>
<Statement> ::=   <VariableDecl> `;'   |
                <Assignment> `;'     |
                <PrintStatement> `;' |
                <IfStatement>        |
                <ForStatement>       |
                <WhileStatement>     |
                <RtrnStatement> `;'  |
                <FunctionDecl>       |
                <Block>
<Block> ::= `{' \{ <Statement> \} `}'
<Program> ::= \{ <Statement> \}
```

Lexer

The first task in development for a complete interpreter for TeaLang is the development of a table driven lexer. The lexer is tasked to split the input program into various types of lexemes, labelled by an assigned token. Each token is an object of the Token class, that has a type, value, and line number as its attributes.

Design

A requirement of the task was to design the lexer in a DFSA – table driven way. This was achieved by designing a complete DFSA for the TeaLang language. Figure 1 shows a subset of this DFSA, which describes the possible transitions from one state to another in order to exit with a float literal at state 8 or earlier as an integer literal at state 6. Transitions between states that are not defined here default to state 24.

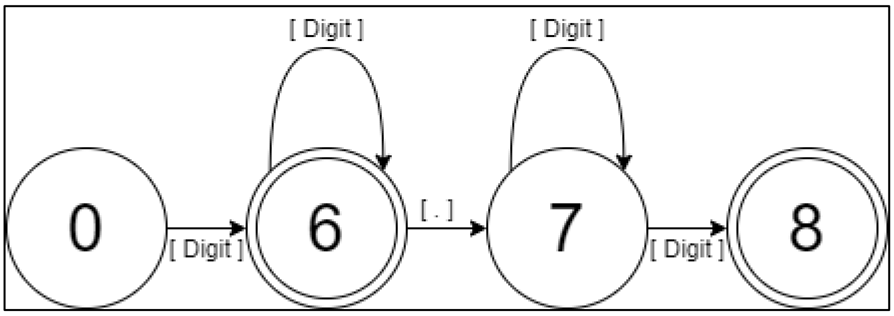

Figure 1 DFSA That Handles Integer & Float Literals

A complete DFSA can be seen in Figure 2 which shows the complete DFSA for the entire TeaLang language. For the sake of neatness transition to state 24 are omitted visually.

Table Driven DFSA Approach

DFSAs are formally defined as a quintuple $(S, \Sigma, \delta, S_0, F)$ with S being the set of states, Σ is the alphabet, δ is the set of transition functions, S_0 is the starting state and F the set of final states. These size of the elements in this quintuple can be reduced by grouping and good design.

S – The Set of States

The set of states can be $!|\Sigma|$ large, where a state for every item in the alphabet is represented. Instead, only states that lead up to an accepted final token are designed, with the 24th state acting as final state for any transition that is not defined for any state A to any state B.

Σ – The Alphabet

Similarly, the alphabet can be large. Representing and designing for every computer processable character would produce a set of extreme proportion. Instead, the alphabet is limited to that of what is accepted by the TeaLang language. This alphabet is defined in Table 1 as a table of character groups, each of these groups paired with their identifying token. Any other character that is read is assigned the INVALID label.

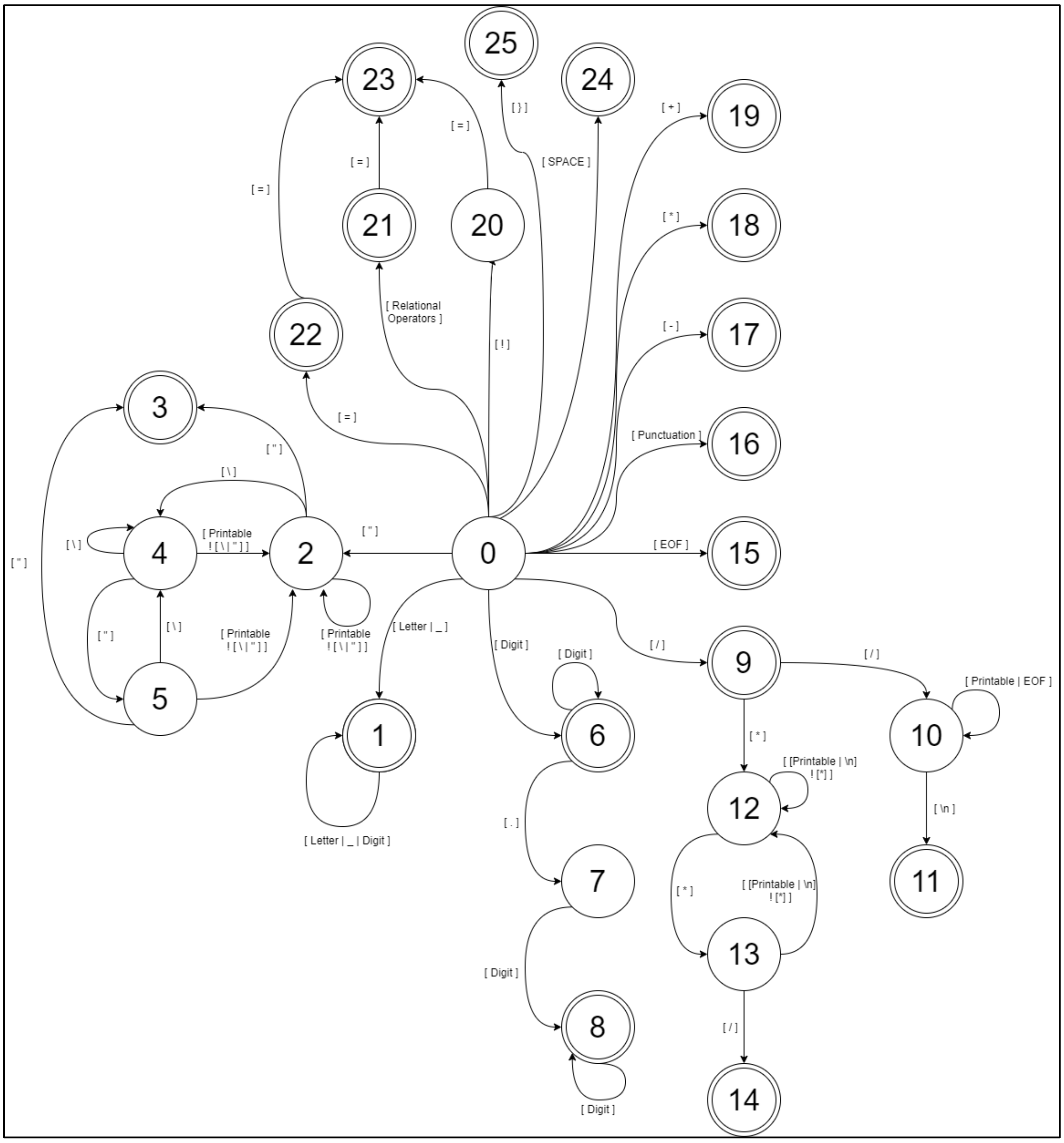

Figure 2 DFSA for the TeaLang Programming Language

Table 1 Character to Transition Type

Label	Condition	Transition Type	Assigned ID
Letter	[A-Za-z]	LETTER	TT0
Digit	[0-9]	DIGIT	TT1
Fullstop	‘.’	FULLSTOP	TT2
Underscore	‘_’	UNDERSCORE	TT3
Asterisk	‘*’	ASTERISK	TT4
Plus	‘+’	PLUS	TT5
Relational	'<' | '>'	RELATIONAL	TT6
Minus	‘-’	MINUS	TT7
Forward Slash	‘/’	FORWARD_SLASH	TT8
Back Slash	‘\’	BACK_SLASH	TT9
Closing Curly	‘}’	CLOSING_CURLY	TT10
Punctuation	'{' | '(' | ')' | ',' | ':' |';'	PUNCTUATION	TT11
Quotation Mark	‘”’	QUOTATION_MARK	TT12
Newline	‘\n’	NEWLINE	TT13
Equals	‘=’	EQUALS	TT14
Exlamanation Mark	‘!’	EXCLAMATION	TT15
Space	‘ ’	SPACE	TT16
Printable	[\x20-\x7E]	PRINTABLE	TT17
Other	![\x20-\x7E]	INVALID	TT18

δ – The Transition Function

Given the defined set of states and alphabet, δ is used as the transition function between any 2 states when given any character from the alphabet. This transition function is complete, i.e. it has a defined transition for any given character in the alphabet, from any given state. These transitions have been visualized in tabular form in Table 2. Shown in Table 2, the first column represented the individual states, with the states with a green background being the final states. The first row represented the Transition Types, for easier representation they have been assigned an id as in Table 1. Then the resulting state is inserted in the cell corresponding to a from state and Transition Type, also for the sake of better representation the S24 state has been blurred.

S_0 – The Start State

The start state is defined as state 0. No transitions into this state are defined as it is used as the starting point of a new token. Apart from being the first state this state has no special attributes

Table 2 DFSA Transition Table

SID	TT0	TT1	TT2	TT3	TT4	TT5	TT6	TT7	TT8	TT9	TT10	TT11	TT12	TT13	TT14	TT15	TT16	TT17	TT18
S0	S1	S24	S24	S1	S18	S19	S21	S17	S9	S24	S25	S16	S2	S24	S23	S20	S20	S24	S24
S1	S1	S24	S24	S1	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S2	S2	S2	S2	S2	S2	S2	S2	S2	S2	S4	S2	S2	S3	S3	S2	S2	S2	S2	S24
S3	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S4	S2	S2	S2	S2	S2	S2	S2	S2	S2	S4	S2	S2	S5	S5	S2	S2	S2	S2	S24
S5	S2	S2	S2	S2	S2	S2	S2	S2	S2	S4	S2	S2	S3	S3	S2	S2	S2	S2	S24
S6	S24	S6	S7	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S7	S24	S8	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S8	S24	S8	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S9	S24	S24	S24	S24	S12	S24	S24	S24	S10	S10	S24	S24	S24	S24	S24	S10	S10	S24	S24
S10	S10	S10	S10	S10	S10	S10	S10	S10	S10	S10	S10	S10	S10	S11	S10	S10	S10	S10	S24
S11	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S12	S12	S12	S12	S12	S13	S12	S12	S12	S12	S12	S12	S12	S12	S13	S12	S12	S12	S12	S24
S13	S12	S12	S12	S12	S14	S12	S12	S12	S14	S14	S12	S12	S12	S12	S12	S14	S14	S14	S24
S14	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S15	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S16	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S17	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S18	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S19	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S20	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S23	S24	S24	S24	S24	S24
S21	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S23	S24	S24	S24	S24	S24
S22	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S23	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24
S25	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24

F – The Set of Final States

This set represents the set of valid end points. A valid token construction is done when a final state transitions to the state 24. This valid pass from S_0 to a state in F results in a token as defined in the Token class. A table representation of these tokens would be too large to include in this report, instead the logic for defining these tokens is explained in the implementation heading.

Implementation

The implementation of the lexer includes the definition and implementation of 2 classes; the Lexer and Token classes, as well as the definition of the lexer namespace, which is occupied by the aforementioned classes, character to alphabet and token validation functionality, the delta transition function, and the programming implementation of the DFSA as described in Figure 2.

lexer Namespace

The lexer namespace was defined as a means of defining common functionality and attributes that are used in the Lexer and Token classes, which are not explicitly tied only with these classes. Here the Transition Type and Token Type represented in Tables 1 and 2, are implemented as an enum.

Data Validation

Within the namespace several data validation functionalities are implemented. These can be split into 2 types; character to alphabet and token validation functionality. Character to alphabet validation is used in between the transitions in the delta function and only accept a char type as an argument. These functions are also used in determining the transition type, i.e. pre-validation of said determination. A typical function `isFullstop` defines whether a character is a full stop or not.

```
bool isFullstop(char c) {
return c == '.';
}
```

Grouping is also done in with these functionalities as is the case for isLetter. In this function a regex statement is used to validate that a passed character is a readable letter as defined in the EBNF.

```
std::regex letter("[A-Za-z]");
bool isLetter(char c) {
      std::string _c;
      _c.push_back(c);
      return (std::regex_match(_c, letter));
}
```

The other set of validation functions, the token validation set, are used to validate a constructed token's value. Each token's value is built by the Lexer and Token classes by following the DFSA in Figure 2 and transitions in Table 3. As a final step these tokens are also validated, this also acts as a confirmation that the lexer implementation is correct. As with the previous set there are 2 subsets of functions, one which validates a specific string, like `isIntType`, and the other which validate a string based on a regex statement like `isInt`.

```
bool isIntType(const std::string &s) {
return s == "int";
}

std::regex intLiteral("^[0-9]*$");
bool isInt(const std::string &s) {
return (std::regex_match(s, intLiteral));
}
```

δ Function

The delta function accepts a state id fromState and a character c. It then returns the current_state variable, which is the state that has been transitioned to, given a state fromState and alphabet item c. The first operation is determining the transition type of c, and this is done via the determineTranstionType function. This function checks and matches c with the valid transition types by using the previously explained character to alphabet function set. An important note here is that the character is checked whether it is printable after all other transition types have been checked for, since all these other types are printable by TeaLang. Lastly if no transition type is defined, the character is labeled as INVALID.

```
TRANSITION_TYPE determineTransitionType(char c) {
if (isLetter(c)) return LETTER;
      if (isDigit(c)) return DIGIT;
      if (isFullstop(c)) return FULLSTOP;
. . .
      if (isPrintable(c)) return PRINTABLE;
      return INVALID;
}
```

Now that the transition type is defined, a switch statement is used to determine to which state the current state can move to. For example, the cases that can be considered in the DFSA shown in Figure 1 are as follows.

```
case DIGIT:
if (fromState == 1) {
      current_state = 1;
} else if (fromState == 0 || fromState == 6) {
            current_state = 6;
} else if (fromState == 7 || fromState == 8) {
            current_state = 8;
} else {
            current_state = 24;
}
break;
case FULLSTOP:
      (fromState == 6)
      ? current_state = 7 : current_state = 24;
break;
```

Like determineTranstionType another check is done for the printable case at the end of the function, before returning the current state. While conducting research on similar projects and work I noticed how many implement the delta function as a 2D – Array. This implementation would have been a faster operation and truer to the 'table driven' requirement, but I feal that my implementation is more readable.

Token Class

The Token class defines each token, or lexeme, with 3 attributes, the type, value, and line number. These tokens are created by the lexer class and stored in a vector object that is then returned as the output of the lexical process. As such a Token is constructed by being passed a string value, a state id, and the line number. The state id and string value is used to determine the token type for type via the determineTokenType function. This function, like determineTranstionType, uses a validation function set, this time the token set, to determine token types.

```
TOKEN_TYPE Token::determineTokenType(std::string &s, unsigned int state) {
switch (state) {
            case 0:
                  return TOK_END;
            case 1:
                  return fromState1(s);
            case 3:
                  return fromState3(s);
            . . .
            case 25:
                  return fromState25(s);
                  default:
            return TOK_INVALID;
}
}
```

Each final state has a dedicated function that checks the final value constructed by the lexer. Here the token validation function set is used to validate the string s.

```
TOKEN_TYPE fromState1(const std::string &s) {
// Keywords
if (isFloatType(s)) return TOK_FLOAT_TYPE;
      if (isIntType(s)) return TOK_INT_TYPE;
      if (isBoolType(s)) return TOK_BOOL_TYPE;
if (isStringType(s)) return TOK_STRING_TYPE;
. . .
      if (isIdentifier(s)) return TOK_IDENTIFIER;
      return TOK_INVALID;
}

TOKEN_TYPE fromState6(const std::string &s) {
// Integer
if (isInt(s)) return TOK_INT;
      return TOK_INVALID;
}
```

Note that, in the fromState1 function keywords and identifiers are defined, and like the printable transition, the identifier token encompasses all keywords in TeaLang, so the check for the identifier token is done at the end.

Lexer class

The Lexer class uses all the defined functionality so far to implement the actual TeaLang lexer. The heavy lifting work is done by the etractLexemes function. Below I will provide pseudo code for the algorithm developed.

```
extractLexemes(string& text):
// define variables
let ret[] : string
let value : string
let previous_state, current_state : int = 0
let lineNumber : int = 1
// go over each character in the passed text
for each(char c in text)
      // Do DFSA transition
      previous_state = current_state
      current_state = $\delta$(previous_state, c)
      if (current_state == 24)
            // Confirm from state is final
            if(previous_state not in finalStates)
                  throw "lexical error"
            // Add new token to ret
            if(value is not empty)
                  let t : Token(value, previous_state, lineNumber)
                  ret.push(t)
            // reset
            current_state = $\delta$(0, c)
            value = ""
            if(current_state == 24)
                  current_state = 0
            else if(current_state != 0)
                  value += c
      else
            if(current_state != 0)
                  value += c
      // Increment lineNumber with \n character discovery
      if(isNewLine(c))
            lineNumber++
// Text has been passed
// Push final valid token
ret.push(Token(value, current_state, lineNumber))
// Push end token
ret.push(Token("", 15, lineNumber))
return ret
```

The return of extractLexemes is set to the public tokens, vector variable in the Lexer class.

A complete implementation of the defined functionality can be found in the Lexer and Token header and source files, found in the Lexer folder. These files also define and occupy the lexer namespace.

Testing

Test 1

```
\cmake-build-debug\TeaLang.exe" -l -p ../Test1.tlng

Process finished with exit code 0
```

Test 2

```
\cmake-build-debug\TeaLang.exe" -l -p ../Test2.tlng

terminate called after throwing an instance of 'std::runtime_error'
  what(): Lexical error on line 1

Process finished with exit code 3
```

Test 3

```
\cmake-build-debug\TeaLang.exe" -l -p ../Test3.tlng

Process finished with exit code 0
```

Test 4

```
\cmake-build-debug\TeaLang.exe" -l -p ../Test4.tlng

Process finished with exit code 0
```

Test 5

```
\cmake-build-debug\TeaLang.exe" -l -p ../Test5.tlng

Process finished with exit code 0
```

Test 6

```
\cmake-build-debug\TeaLang.exe" -l -p ../Test6.tlng

Process finished with exit code 0
```

Parser

The second task in development of the TeaLang interpreter is developing the parser. The role of the parser is to go over the extracted lexemes and build an Abstract Syntax Tree (AST), in the process checking for syntax errors.

Design

Task 2 requires a hand-crafted predictive parser for the TeaLang language. This was achieved by defining numerous AST nodes and implementing the construction of such nodes in the parser. The parser is predictive in how it determines the set of possible AST nodes. For example, given a TOK_LET token, only the variable declaration AST node can be possibly constructed, while if given a TOK_IDENTIFIER token various AST nodes can be created based on the token ahead, such as the function call or identifier AST nodes. In a complete pass the parser produces a complete AST with the root node being the ASTProgramNode. The generation of these AST objects is handled by the Parser class defined in the Parser header and source files, found in the Parser folder.

AST Classes

An AST node was designed for each rule as defined in the EBNF of TeaLang. OOP and inheritance is used to structure these nodes. The original ASTNode defines the visitor requirement, I will get back to this design patter in the XML Generation section. Then 2 sub-classes are defined, the ASTStatementNode and ASTExpressionNode, afterwards the individual AST nodes are defined as either a statement or expression node. The key difference here is that statement nodes can exist as a complete line of code, while expression nodes form part of these statements. The AST has been represented in XML format, and examples can be seen in the documentation regarding that task.

Parser Class

The Parser class defines the set of parse functions, that are responsible for parsing and constructing the AST objects. It is in these functions that syntax errors are checked for and raised. The parsing of these AST statements and objects is done by observing a token window of size 2, which returns 2 tokens. An action is taken depending on the 2 tokens viewed. The moveTokenWindow handles the movement of this viewing window, given a step size.

Implementation

AST Classes

The AST classes are defined in the visitor header file, found in the Visitor folder. They are defined here as they are crucial for the visitor design pattern. These classes are then implemented in the AST header and source files. In the AST header file, a constructor for each AST Node is defined as well as laying out the attributes. For example, the ASTUnaryNode is defined as show below.

```
class ASTUnaryNode : public ASTExprNode {
public:
ASTUnaryNode(std::shared_ptr<ASTExprNode> exprNode
                  ,std::string op
                  ,unsigned int lineNumber)
                  :
                  exprNode(std::move(exprNode))
                  op(std::move(op)),
                  lineNumber(lineNumber)
                  {};
~ASTUnaryNode() = default;
      std::shared_ptr<ASTExprNode> exprNode;
      std::string op;
      unsigned int lineNumber;
      void accept(visitor::Visitor* v) override;
};
```

To give another example, this time for a statement node, the very simple ASTPrintNode is included below.

```
class ASTPrintNode : public ASTStatementNode {
public:
ASTPrintNode(std::shared_ptr<ASTExprNode> exprNode
                  ,unsigned int lineNumber) :
exprNode(std::move(exprNode)),
lineNumber(lineNumber)
      {};
      ~ASTPrintNode() = default;

      std::shared_ptr<ASTExprNode> exprNode;
      unsigned int lineNumber;
      void accept(visitor::Visitor* v) override;
};
```

For these classes the shared pointer was used to reduce memory leaks and have a more efficient memory management implementation. The visit function is then implemented in the source file as can be seen below for the shown classes. This functionality, which forms part of the visitor design pattern, will be first utilized by the XML generation, and so will be documented at that stage. As a final note on the AST classes, function calling has been implemented both for the ASTStatement and ASTExpression nodes.

```
void ASTUnaryNode::accept(visitor::Visitor *v) {
v->visit(this);
}
void ASTPrintNode::accept(visitor::Visitor *v) {
v->visit(this);
}
```

Parser Class

The parser class uses the EBNF provided to construct the defined AST objects. On initialization the class initializes the token view window in the form of 2 variables.

```
explicit Parser(std::vector<lexer::Token> tokens) : tokens(tokens) {
// Initialise the currentToken and nextToken
      currentToken = tokens.front();
      nextToken = tokens.at(1);
      currentLoc = 0;
}
```

Then starting with the root ASTProgramNode, an AST tree is constructed via the parse set of functions. Each function in this set is directly linked to an AST class, as it acts as its builder for the AST construction. In addition, the parseType function returns a string value of the type.

```
std::string Parser::parseType() const {
switch (currentToken.type) {
      case lexer::TOK_INT_TYPE:
      case lexer::TOK_FLOAT_TYPE:
      case lexer::TOK_BOOL_TYPE:
      case lexer::TOK_STRING_TYPE:
      return currentToken.value;
default:
      throw std::runtime_error();
}
}
```

As mentioned earlier the parser starts by constructing the ASTProgramNode, which is simply a collection of statement nodes. In parseProgram the program is parsed until an TOK_END is found, while comments are ignored. Parsing consists of adding a parsedStatement return to the collection of statements. The parseProgram is also used for parsing block cases, were if the block flag is true, the parsing also stops with the discovery of the TOK_CLOSING_CURLY token.

```
std::shared_ptr<ASTProgramNode>
Parser::parseProgram(bool block) {
      auto statements = std::vector<std::shared_ptr<ASTStatementNode>>();
// Loop over each token and stop with an END token
while (currentToken.type != lexer::TOK_END && !(block &&
       currentToken.type == lexer::TOK_CLOSING_CURLY)) {
// Ignore comments and skip '}' if parsing a block
if (currentToken.type != lexer::TOK_SINGLE_LINE_COMMENT
          && currentToken.type != lexer::TOK_MULTI_LINE_COMMENT
          && (!block || currentToken.type !=lexer::TOK_CLOSING_CURLY))
         statements.push_back(parseStatement());
      // Get next Token
// There is a case when a scope/block is empty where were need
      // to check before moving the token window
      if (  currentToken.type != lexer::TOK_END
            || nextToken.type != lexer::TOK_END)
            moveTokenWindow();
}
return std::make_shared<ASTProgramNode>(statements);
}
```

At this stage, the parseStatement function is called. This function features a switch statement, considering the currentToken variable and depending on the token type an action is taken. A snippet of this function is provided below.

```
std::shared_ptr<ASTStatementNode> Parser::parseStatement() {
// Parse a singular statement
// The current token type determines what can be parsed
switch (currentToken.type) {
// Variable declaration case
case lexer::TOK_LET:
            return parseDeclaration();
//An identifier can either be a Function call or an assignment
case lexer::TOK_IDENTIFIER:
// If next token is '(' then we found a Function call
            if (nextToken.type == lexer::TOK_OPENING_CURVY){
                  return std::make_shared<ASTSFunctionCallNode>
                        (parseFunctionCall(true));
            } else {
            // if not, its should be an Assignment
                  return parseAssignment();
            }
            // Print case
            case lexer::TOK_PRINT:
            return parsePrint();
            . . .
            // Any other case is an error case
            default:
           throw std::runtime_error();
      }
 }
```

All the parse functions called within the parseStatement return an AST class that inherits from the ASTStatement node, for example the parsePrint function.

```
std::shared_ptr<ASTPrintNode> Parser::parsePrint() {
// Determine line number
      unsigned int lineNumber = currentToken.lineNumber;
      // Current token is PRINT
      // Get next token
      moveTokenWindow();
      // Get expression after print
      auto expr = parseExpression();
      // Get next token
      moveTokenWindow();
      // Ensure proper syntax
      if (currentToken.type != lexer::TOK_SEMICOLON)
            throw std::runtime_error();
// Create ASTPrintNode to return
return std::make_shared<ASTPrintNode>(expr, lineNumber);
}
```

It is now in these ASTStatement inheriting AST class parse functions that the parseExpression is called. As defined in the EBNF the most basic form of an expression is the Factor. For this reason, once entering the parseExpression, the scope travels to the parseFactor function recursively via the parseSimpleExpression and parseTerm functions.

```
std::shared_ptr<ASTExprNode> Parser::parseExpression() {
auto simple_expr =
      std::shared_ptr<ASTExprNode>(parseSimpleExpression());
      . . .
}
std::shared_ptr<ASTExprNode> Parser::parseSimpleExpression() {
auto term = std::shared_ptr<ASTExprNode>(parseTerm());
      . . .
}
std::shared_ptr<ASTExprNode> Parser::parseTerm() {
auto factor = std::shared_ptr<ASTExprNode>(parseFactor());
      . . .
}
```

The parseFactor function, functions as the parseStatement does, utilizing a switch statement taking an action based on the currentToken token type.

```
std::shared_ptr<ASTExprNode> Parser::parseFactor() {
// Determine line number
unsigned int lineNumber = currentToken.lineNumber;
      // Define operator for Unary
      std::string op;
      // check current token type
switch (currentToken.type) {
// Literal Cases
case lexer::TOK_INT:
      return std::make_shared<ASTLiteralNode<int>>(
      std::stoi(currentToken.value), lineNumber);
case lexer::TOK_FLOAT:
      return std::make_shared<ASTLiteralNode<float>>(
      std::stof(currentToken.value), lineNumber);
      . . .
case lexer::TOK_NOT:
            // Current token is either not or -
            // store the operator
            op = currentToken.value;
            // Move over it
            moveTokenWindow();
            // return an ASTUnaryNode
            return std::make_shared<ASTUnaryNode>(
                   parseExpression(), op, currentToken.lineNumber);
            default:
                throw std::runtime_error();
}
```

Testing Returns

Test 1

```
\cmake-build-debug\TeaLang.exe" -p -p ../Test1.tlng

Process finished with exit code 0
```

Test 2

~omitted as failure is already proven

Test 3

```
\cmake-build-debug\TeaLang.exe" -p -p ../Test3.tlng

Process finished with exit code 0
```

Test 4

```
\cmake-build-debug\TeaLang.exe" -p -p ../Test4.tlng
terminate called after throwing an instance of 'std::runtime_error'
  what():  Expected '{' after ')' on line 4.

Process finished with exit code 3
```

Test 5

```
\cmake-build-debug\TeaLang.exe" -p -p ../Test5.tlng

Process finished with exit code 0
```

Test 6

```
\cmake-build-debug\TeaLang.exe" -p -p ../Test6.tlng

Process finished with exit code 0
```

XML Generation

While not necessary for the development of the TeaLang interpreter Task 3 requires the implementation of an xml generation artefact that would, generate an xml file for the AST produced by the parser. This artefact is the first task that uses the visitor design pattern.

Visitor Design Pattern

The visitor design pattern is a behavioral design pattern and is used when a number of different operations on a group of element classes need to be performed. With the help of the visitor pattern, the operational logic is moved from these classes to the class performing the operation. A visitor class is used which changes the executing algorithm of an element class. This approach encourages designing lightweight element classes as processing functionality is removed from their list of responsibilities. New functionality can easily be added to the original inheritance hierarchy by creating a new Visitor subclass. The design stipulates that these element classes implement visit and accept functions. In the TeaLang case these elements would be the AST nodes, and sub nodes within said nodes. As has already been shown in with the AST implementation where the virtual visit function is implemented by calling the accept function. It is then in the hands of the various Visitor classes such as the XML Generator, Semantic Analyser, and Interpreter to implement their own visit functionality that operates on the AST nodes. The implementation of these classes will be gone over in their respective sub-headings.

Design

The XMLVisitor class inherits from the Visitor class, and so follows the visitor design pattern as described above. A visit function for all the AST nodes is implemented. In this visit set of functions each visit handles its node’s attributes and sub-AST nodes to create a valid xml file.

Implementation

On initialization of the XMLVisitor class, an ofstream object is initialized and over the process of visiting each AST node, is streamed in indentations and string representations of the AST nodes. This implementation of the visitor class is very simple with little to no logic required. Apart from the visit set of functions, 2 other functions are implemented. These are the indentation function that along with the indentationLevel variable manages the indentation of text in the file and the xmlSafeOp function, that converts ‘<’ into ‘gt’ and ‘>’ into lt, to preserve a valid xml file format. Below the visit functions for the ASTProgramNode and ASTPrintNode are included.

```
void XMLVisitor::visit(parser::ASTProgramNode* programNode) {
// Add initial <program> tag
      xmlfile << indentation() << "<program>" << std::endl;
      // Add indentation level
      indentationLevel++;
      // For each statement, generate XML for it
      for (auto &statement : programNode->statements)
            statement->accept(this);
// Remove indentation level
      indentationLevel--;
      // Add closing tag
      xmlfile << indentation() << "</program>" << std::endl;
}

void XMLVisitor::visit(parser::ASTPrintNode *printNode) {
// Add initial <print> tag
      xmlfile << indentation() << "<print>" << std::endl;
      // Add indentation level
      indentationLevel++;
      // Expression tags
      printNode->exprNode->accept(this);
      // Remove indentation level
      indentationLevel--;
      // Add closing tag
      xmlfile << indentation() << "</print>" << std::endl;
}
```

Testing

Test 1

```
<program>
    <declaration>
        <id type = "bool">x</id>
        <bool>true</bool>
    </declaration>
    <if>
        <condition>
            <id>x</id>
        </condition>
        <ifBlock>
            <block>
                <declaration>
                    <id type = "float">y</id>
                    <float>10.000000</float>
                </declaration>
                <print>
                    <id>y</id>
                </print>
            </block>
        </ifBlock>
    </if>
    <print>
        <id>y</id>
    </print>
</program>
```

Test 2

~omitted as failure is already proven

Test 3

```
<program>
    <functionDeclaration type = "float">
        <id>Factorial</id>
        <param type = "float">x</param>
        <block>
            <if>
                <condition>
                    <bin op = "&gt;=">
                        <id>x</id>
                        <int>1</int>
                    </bin>
                </condition>
                <ifBlock>
                    <block>
                        <return>
                            <bin op = "*">
                                <id>x</id>
                                <functionEcall>
                                    <id>Factorial</id>
                                    <param>
                                        <bin op = "-">
                                            <id>x</id>
                                            <int>1</int>
                                        </bin>
                                    </param>
                                </functionEcall>
                            </bin>
                        </return>
                    </block>
                </ifBlock>
            <elseBlock>
                <block>
                    <return>
                        <int>1</int>
                    </return>
                </block>
            </elseBlock>
            </if>
        </block>
    </functionDeclaration>
    <print>
        <functionEcall>
            <id>Factorial</id>
            <param>
                <float>5.000000</float>
            </param>
        </functionEcall>
    </print>
</program>
```

Test 4

~omitted as failure is already proven

Test 5

```
<program>
    <functionDeclaration type = "float">
        <id>Factorial</id>
        <param type = "int">x</param>
        <block>
            <if>
                <condition>
                    <bin op = "&gt;=">
                        <id>x</id>
                        <int>1</int>
                    </bin>
                </condition>
                <ifBlock>
                    <block>
                        <return>
                            <bin op = "*">
                                <id>x</id>
                                <functionEcall>
                                    <id>Factorial</id>
                                    <param>
                                        <bin op = "-">
                                            <id>x</id>
                                            <int>1</int>
                                        </bin>
                                    </param>
                                </functionEcall>
                            </bin>
                        </return>
                    </block>
                </ifBlock>
            <elseBlock>
                <block>
                    <return>
                        <int>1</int>
                    </return>
                </block>
            </elseBlock>
            </if>
        </block>
    </functionDeclaration>
    <print>
        <functionEcall>
            <id>Factorial</id>
            <param>
                <float>5.000000</float>
            </param>
        </functionEcall>
    </print>
</program>
```

Test 6

```xml
<program>
    <functionDeclaration type = "float">
        <id>Square</id>
        <param type = "float">x</param>
        <block>
            <return>
                <bin op = "*">
                    <id>x</id>
                    <id>x</id>
                </bin>
            </return>
        </block>
    </functionDeclaration>
    <functionDeclaration type = "bool">
        <id>XGreaterThanY</id>
        <param type = "float">x</param>
        <param type = "float">y</param>
        <block>
            <declaration>
                <id type = "bool">ans</id>
                <bool>true</bool>
            </declaration>
            <if>
                <condition>
                    <bin op = "&gt;">
                        <id>y</id>
                        <id>x</id>
                    </bin>
                </condition>
                <ifBlock>
                    <block>
                        <assign>
                            <id>ans</id>
                            <bool>false</bool>
                        </assign>
                    </block>
                </ifBlock>
            </if>
            <return>
                <id>ans</id>
            </return>
        </block>
    </functionDeclaration>
    <functionDeclaration type = "bool">
        <id>XGreaterThanYv2</id>
        <param type = "float">x</param>
        <param type = "float">y</param>
        <block>
            <return>
                <bin op = "&gt;">
                    <id>x</id>
                    <id>y</id>
                </bin>
            </return>
        </block>
    </functionDeclaration>
    <functionDeclaration type = "float">
        <id>AverageOfThree</id>
        <param type = "float">x</param>
        <param type = "float">y</param>
        <param type = "float">z</param>
        <block>
            <declaration>
                <id type = "float">total</id>
                <bin op = "+">
                    <id>x</id>
                    <bin op = "+">
                        <id>y</id>
                        <id>z</id>
                    </bin>
                </bin>
            </declaration>
            <return>
                <bin op = "/">
                    <id>total</id>
                    <float>3.000000</float>
                </bin>
            </return>
        </block>
    </functionDeclaration>
    <functionDeclaration type = "string">
        <id>JoinStr</id>
        <param type = "string">s1</param>
        <param type = "string">s2</param>
        <block>
            <declaration>
                <id type = "string">s3</id>
                <bin op = "+">
                    <id>s1</id>
                    <id>s2</id>
                </bin>
            </declaration>
            <return>
                <id>s3</id>
            </return>
        </block>
    </functionDeclaration>
    <declaration>
        <id type = "float">x</id>
        <float>2.400000</float>
    </declaration>
    <declaration>
        <id type = "float">y</id>
        <functionEcall>
            <id>Square</id>
            <param>
                <float>2.500000</float>
            </param>
        </functionEcall>
    </declaration>
    <print>
        <id>y</id>
    </print>
    <print>
        <functionEcall>
            <id>XGreaterThanY</id>
            <param>
                <id>x</id>
            </param>
            <param>
                <float>2.300000</float>
            </param>
        </functionEcall>
    </print>
    <print>
        <functionEcall>
            <id>XGreaterThanYv2</id>
            <param>
                <functionEcall>
                    <id>Square</id>
                    <param>
                        <float>1.500000</float>
                    </param>
                </functionEcall>
            </param>
            <param>
                <id>y</id>
            </param>
        </functionEcall>
    </print>
    <print>
        <functionEcall>
            <id>AverageOfThree</id>
            <param>
                <id>x</id>
            </param>
            <param>
                <id>y</id>
            </param>
            <param>
                <float>1.200000</float>
            </param>
        </functionEcall>
    </print>
    <print>
        <functionEcall>
            <id>JoinStr</id>
            <param>
                <string>Hello</string>
            </param>
            <param>
                <string>World</string>
            </param>
        </functionEcall>
    </print>
    <for>
        <declaration>
            <declaration>
                <id type = "int">ii</id>
                <int>0</int>
            </declaration>
        </declaration>
        <condition>
            <bin op = "&lt;">
                <id>ii</id>
                <int>10</int>
            </bin>
        </condition>
        <assignment>
            <assign>
                <id>ii</id>
                <bin op = "+">
                    <id>ii</id>
                    <int>1</int>
                </bin>
            </assign>
        </assignment>
        <block>
            <print>
                <id>ii</id>
            </print>
        </block>
    </for>
    <declaration>
        <id type = "int">ii</id>
        <int>0</int>
    </declaration>
    <while>
        <condition>
            <bin op = "&lt;">
                <id>ii</id>
                <int>10</int>
            </bin>
        </condition>
        <block>
            <print>
                <id>ii</id>
            </print>
            <assign>
                <id>ii</id>
                <bin op = "+">
                    <id>ii</id>
                    <int>1</int>
                </bin>
            </assign>
        </block>
    </while>
</program>
```

Semantic Pass

To continue the development of the TeaLang Interpreter a semantic analyzer is required to check the semantical validity of a program. Semantic validity handles anything that requires a type, this means type matching in binary expressions, the declaration of functions and variables and ensuring no double declarations happen in the same scope. Variable shadowing is supported in sub scopes, while function overloading is implemented only for Tea2Lang. Implicit and automatic typecasting however are not supported by TeaLang.

Design

The semantic analyzer is designed on the visitor design pattern, this however has been described already and so only the salient points in the implementation process will be gone over. Apart from the visitor design, a new namespace, semantic, is defined to include 3 classes. These classes: Variable, Function and Scope include the basic operations that the SemanticVisitor class will use. The variable class defined with 3 attributes: a string type, a string identifier, and an integer lineNumber. The Function class is defined with the same 3 attributes and an extra attribute of a vector of strings which represent the parameter types.

The Scope class defines the functionality for these classes. 2 private attributes: variableTable and functionTable, are maps with a string key representing the object's identifier and a value of the intended type (either Variable or Function). A public Boolean isGlobal variable is also defined. This variable is only ever turned on for the global scopes and is what allows or stops function declarations in the scope. Finally, an insert, find and found functions are defined for both tables. The insert function adds a new Variable or Function to its respective table, the find function returns an iterator from the map, and the found function, confirms the whether the find function was successful or not.

Implementation

The implementation of this task can be split into the 2 namespaces, semantic and visitor.

Semantic

In the semantic scope the implementation of the Variable, Function and Scope classes can be found. As explained in the design heading, these classes are very simple, so below I will include the code snippets that handle the Variable class. The Function class handling functions are written in the same way.

```
bool Scope::insert(const Variable& v){
if (v.type.empty()){
            throw VariableTypeException();
}
      auto ret = variableTable.insert(std::pair<std::string,
                                    Variable>(v.identifier, v) );
return ret.second;
}

auto Scope::find(const Variable& v) {
return variableTable.find(v.identifier);
}

bool Scope::found
(
      std::_Rb_tree_iterator<std::pair<const std::basic_string<char,
      std::char_traits<char>, std::allocator<char>>, Variable>> result
){
return result != variableTable.end();
}
```

Visitor

Apart from the visit set of functions the SemanticVisitor also makes use of 3 attributes: a vector of Scopes scopes, a string currentType and a boolean returns. These variables are used to store important information throughout the semantic pass. The visit functions in the semantic analyzer handle the semantic analyses operation of the AST. This implementation consists of various visit functions that in turn, accept their sub-nodes, like how the XMLVisitor class' visit functions are implemented. Each visit is implemented such that it can detect semantic errors for the given AST node. Most of the heavy lifting is done in the AST expression nodes. This is because variable declaration and assignment, function declaration and calling, and any other operation requiring typing, their AST nodes inherit from the ASTExpression class. As an example, let us follow a complete visitation to the ASTWhileNode and its sub-nodes:

```
void SemanticAnalyser::visit(parser::ASTWhileNode *whileNode) {
// Get the condition type
      whileNode -> condition -> accept(this);
      // Make sure it is boolean
      if(currentType != "bool")
            throw std::runtime_error();
// Check the while block
      whileNode -> loopBlock -> accept(this);
}
```

The first action taken is accepting a visit to the condition, which as defined in the ASTWhileNode class is a ASTExpressionNode. Let us say for the sake of the example, that the condition is not false. This would be defined as an ASTUnaryNode, which visit is shown below.

```
void SemanticAnalyser::visit(parser::ASTUnaryNode *unaryNode) {
// Go over exprNode
      unaryNode -> exprNode -> accept(this);
      // Handle different cases
      if (currentType == "string") {
            throw std::runtime_error();
      }else if (currentType == "int" || currentType == "float") {
            if(unaryNode -> op != "-")
                  throw std::runtime_error();
      }else if (currentType == "bool"){
            if(unaryNode -> op != "not")
                  throw std::runtime_error();
      }
}
```

An ASTUnaryNode defines its exprNode, as another ASTExpressionNode. Since ‘false’ is a Boolean literal, the next visit goes to the respective ASTLiteralNode.

```
void SemanticAnalyser::visit(parser::ASTLiteralNode<bool> *literalNode) {
currentType = "bool";
}
```

The ASTLiteralNode visits change the current type. As the Literal visit returns the currentType is also checked in the Unary visit, as not all variable types upport a unary operation. In the while visit, this type is also checked for to confirm that the condition ASTExpressionNode is in fact a Boolean condition. Lastly the ASTBlockNode is visited, which in turn visits each statement within the while loop.Other visits are more complex, such as function and variable declaration. Function declaration also handles scopes so this function will be explained in detail below.

Function Declaration

On visiting, the function checks whether the latest scope in scopes is the global scope or not. This is achieved by getting the last item in the vector via the back() function.

```
void SemanticAnalyser::visit(parser::ASTFunctionDeclarationNode
                              *functionDeclarationNode) {
// If current scope is not global then do not allow declaration
 auto scope = scopes.back();
 if (!scope->isGlobal()){
      throw std::runtime_error();

```

Next a new scope is added to scopes. Note the scope variable stays untouched. This new scope is created as the function scope.Each ASTFunctionDeclarationNode features a vector of strings, representing the parameter types. These are looped over and inserted to function scope.

```
scopes.emplace_back(std::make_shared<semantic::Scope>());
      std::vector<std::string> paramTypes;
      for (const auto& param : functionDeclarationNode->parameters){
            paramTypes.emplace_back(param.second);
            // While going over the types add these to the new scope
            scopes.back()->insert(semantic::Variable(param.second,
                        param.first, functionDeclarationNode->lineNumber));
}
```

Now all the required data is in hand to declare a new function in the global scope. The process is as follows. First initialize a new Function f. Then by using the scope.find and scope.found functions check in the global scope if the function has already been declared. If it hasn't, f can be inserted to the global scope.

```
        // NOTE: The scope variable is still viewing the global scope
        // now generate the function object
        semantic::Function f(functionDeclarationNode->type,
                        functionDeclarationNode->identifier->identifier,
                        paramTypes, functionDeclarationNode->lineNumber);
        // Try to insert f
        auto result = scope->find(f);

if(scope->found(result)){
            // The variable has already been declared in the current scope
            // Overloading is a problem for task 2
            throw std::runtime_error();
}
scope->insert(f);
```

The final step is to check the function block and see that it is semantically correct. This also includes checking for a return. It is at this stage that the returns attribute is set to false. After the block visit the returns variable is expected to be true. Finally, the return type is matched with the declared type and the function scope is popped, i.e. removed.

```
      returns = false;
      functionDeclarationNode->functionBlock->accept(this);
      if(!returns){
           throw std::runtime_error();
      }
      // Check that the return type matches with the function type
if(functionDeclarationNode->type != currentType) {
            throw std::runtime_error();
      }
      // Close function scope
      // This discards any declared variable in the foo() section
scopes.pop_back();
}
```

Testing

Test 1

```
\cmake-build-debug\TeaLang.exe" -s -p ../Test1.tlng
terminate called after throwing an instance of 'std::runtime_error'
  what():  Object with identifier y called on line 10 has not been
declared.

Process finished with exit code 3
```

Test 2

~omitted as failure is already proven

Test 3

```
\cmake-build-debug\TeaLang.exe" -s -p ../Test3.tlng

Process finished with exit code 0
```

Test 4

~omitted as failure is already proven

Test 5

```
\cmake-build-debug\TeaLang.exe" -s -p ../Test5.tlng
terminate called after throwing an instance of 'std::runtime_error'
  what():  Expression with left type int does not share a common right type
float on line 3.
Implicit and Automatic Typecasting is not supported by TeaLang.

Process finished with exit code 3
```

Test 6

```
\cmake-build-debug\TeaLang.exe" -s -p ../Test6.tlng

Process finished with exit code 0
```

Execution Pass

The final requirement is developing the interpreter execution pass for the TeaLang language. This final task consists of executing the lines of code by managing the memory of the variables and functions and by operating on these variables correctly.

Design

As a visitor class the same pattern as has been used before will be re-used. And like the Semantic Analyzer design, the Interpreter will also have an interpreter namespace occupied by interpreter specific classes. Similar to the Semantic Analyzer 3 classes are created: Variable, Function and Table. The Variable and Function classes inherit and add attributes to the semantic namespace classes. The Table class completely replaces the scope class. This class makes use of templating and is a wrapper for the std::map class.

The Interpreter class is designed to use these classes to store the required information. Each Variable has a vector of values, treated as a stack. When a new value is assigned to the variable, be it in a function call, or expression, a new value is pushed. If this push happens in a function call, this pushed value is popped back once the function finishes, otherwise in the global scope the value simply replaces the last value by popping then pushing.

Implementation

In this heading I will go over the implementation of the 3 interpreter namespace classes and the visitor Interpreter class.

Variable

The interpreter::Variable class inherits from semantic::Variable and adds 3 new attributes: values, a vector of values, latestValue which is a copy of the values.back(), and size which follows the size of values. Functionality is not given to Variable, it is the Table class that handles operations on Variable.

```
template <typename T>
class Variable : public semantic::Variable{
public:
Variable(const std::string& type, const std::string& identifier, T
            value, unsigned int lineNumber)
                  :
                  semantic::Variable(type, identifier, lineNumber),
                  latestValue(value),
                  size(0)
                  {
                        values.emplace_back(value);
                  };
...
}
```

Function

Like Variable, the interpreter::Function class inherits from its counterpart in the semantic namespace. 2 new attributes are added a vector of strings paramIDs and a pointer to the function's functionBlock ASTBlockNode. Unlike Variable, functionality is left in the hand of the Interpreter class, to implement the find, found and insert functions. These functions will be gone over in the Table heading, as they are implemented very similarly.

```
class Function : public semantic::Function{
public:
Function(   const std::string& type, const std::string& identifier,
            const std::vector<std::string>& paramTypes,
            std::vector<std::string>  paramIDs,
            std::shared_ptr<parser::ASTBlockNode> blockNode,
            unsigned int lineNumber)
            :
            semantic::Function(type, identifier, paramTypes, lineNumber),
            paramIDs(std::move(paramIDs)),
            blockNode(std::move(blockNode))
            {};
. . .
}
```

Table

The Table class replaces the Scope class from the semantic namespace. As mentioned earlier it is a wrapper for the std::map class with 5 functions: find, insert, found, pop_back, and get. These functions handle the creation of the mentioned, stack style variables. Each of these functions will be gone over below.

Firstly, the class definition is included below as a reference.

```
template <typename Key, typename Value>
class Table {
public:
Table(){
            self = std::map<Key, Value>();
};

~Table() = default;
std::map<Key, Value> self;


auto find(Value v);
bool insert(Value v);
bool found  (
            std::_Rb_tree_iterator<std::pair<const std::basic_string<char,
            std::char_traits<char>, std::allocator<char>>, Value>> result
                  );
      void pop_back(const std::string& identifier);
      Value get(const std::string& identifier = "0CurrentVariable");
};
```

Starting with the find and found functions, both functions are re-implemented from the scope class.

```
template<typename Key, typename Value>
bool Table<Key, Value>::found(
std::_Rb_tree_iterator<std::pair<const std::basic_string<char,
      std::char_traits<char>, std::allocator<char>>, Value>> result) {
            return result != self.end();
}

template<typename Key, typename Value>
bool Table<Key, Value>::found(
std::_Rb_tree_iterator<std::pair<const std::basic_string<char,
      std::char_traits<char>, std::allocator<char>>, Value>> result) {
      return result != self.end();
}
```

The insert function handles creating the insertion of both new variables and new values into existing variables. This depends on whether a fin search is successful or not. A small but very important task insert is handled with is updating the size variable. As we will see in the pop_back function this variable is crucial to successful operation handling.

```
template<typename Key, typename Value>
bool Table<Key, Value>::insert(Value v) {
if(v.type.empty()){
            throw semantic::VariableTypeException();
}
      auto result = find(v);
      if(found(result)){
            // Copy the result variable
            Value cpy(result -> second);
            // add the new value
            cpy.values.emplace_back(v.latestValue);
            ++cpy.size;
            cpy.latestValue = v.latestValue;
            // remove the result
            self.erase(result);
            // insert the copy
            insert(cpy);
            return false;
}else{
            // The variable doesnt exist so we add a new one
            v.size = v.values.size();
            auto ret = self.insert(std::pair<Key, Value>(v.identifier, v));
            return ret.second;
}
}
```

The pop_back handles popping the latestValue from values as well as updating latestValue.

```
 template<typename Key, typename Value>
 void Table<Key, Value>::pop_back(const std::string &identifier) {
      auto result = find(Value(identifier));
      if(!found(result)){
            throw std::runtime_error();
}
      // Copy the result variable
      Value cpy (result -> second);
      // pop the value from the copy
      --cpy.size;
      if(cpy.size == -1){
            cpy.size = 0;
}else{
            cpy.values.erase(cpy.values.begin() + cpy.size);
}
      if(cpy.size != 0){
            cpy.latestValue = cpy.values.back();
      }
      // remove the result
      self.erase(result);
      // insert the copy
      insert(cpy);
}
```

Lastly the get functions is a getter function that retrieves a variable with the passed identifier.

```
template<typename Key, typename Value>
Value Table<Key, Value>::get(const std::string &identifier) {
auto result = find(Value(identifier));
      if(!found(result)){
            throw std::runtime_error();
}
      auto ret = result -> second;
      // pop_back case
      if(identifier == "0CurrentVariable") {
            pop_back("0CurrentVariable");
}
      // return the popped value
      return ret;
}
```

Interpreter

The Intepreter class inherits from Visitor and so follows the visitor design pattern. Each visit then uses the described functionality above, and new functionality, to complete the execution pass. Apart from the interpreter namespace functionality the Interpreter class also makes use of 2 strings, currentType and currentID, a Boolean function variable and a vector of a string tuple toPop. A table for each type is also intitialised, not that operations on the functionTable are handled by the Interpreter class and not the Table class. On initialisation 2 variables of each type are inserted into their respective table, with id “0Literal” and “0CurrentVariable”. These variables are initialised with a 0 infront, because it is impossible for a TeaLang user to initialise in the same way. These variables will be used as variables within the implementation of the interpreter.

4 visits will be gone over in this section: Function Declaration, Function Calling, Variable Declaration and Assignment.

Function Declaration

Function declaration is handled by the ASTFunctionDeclarationNode visit shown below. Since at this stage it is certain there should be no errors, the only requirement here is to build the Function object. This is done by extracting the parameter types and ids and other data from the ASTNode. Then via the insert function a new Function is inserted.

```
void Interpreter::visit(parser::ASTFunctionDeclarationNode
                        *functionDeclarationNode) {

std::vector<std::string> paramTypes;
      std::vector<std::string> paramIDs;
      for (auto & parameter : functionDeclarationNode->parameters){
            paramTypes.emplace_back(parameter.second);
            paramIDs.emplace_back(parameter.first);
      }

      // Insert the new function
      insert (
            interpreter::Function   (
                  functionDeclarationNode->type,
                  functionDeclarationNode -> identifier -> identifier,
                  paramTypes,
                  paramIDs,
                  functionDeclarationNode->functionBlock,
                  functionDeclarationNode -> lineNumber
                                    ));
}
```

Function Calling

Function calling can occur as either a statement or an expression. In either case both are handled in the same manner. The following algorithm is used to find and call a function.

```
visitFunctionCall(ASTFunctionCall *fc)
let f : Function(f->identifier->identifier)
let result : auto = find(f)
if(!found(result)){
      thow "Function not found"

f = result.second //update f to the completely defined function

for(i: int = 0; i< fc->parameters.size(); i+=1)
      fc -> parameters.at(i) -> accept(this)
      // This visit updates the currentID and currentType
      // store current ID so that we dont need to visit the parameters
      // again to pop their values
      toPop.push(pair(currentType, f.paramIDs.at(i)))
      table.insert(
            Variable    (
                                    [Variable constructor parameters]
                        )
            )

function = true
f.blockNode -> accept(this)
function = false
```

This algorithm describes the handling of the parameter variables but leaves execution of the function block to the visit in the f.blockNode. Declaration and assignment would be 2 such statements that may be executed in the function block and are also prime examples to show how the variable system works.

Variable Declaration
Variable declaration in the interpreter works utilises the Variable and Table classes to amend the stack like structures designed. This approach was taken because it was easy to implement and easy to understand. Since the Semantic Analyser handles semantic errors, the interpreter does not need to worry that a variable is being declared twice or in the wrong place, it simply needs to declare a variable. The code is provided below.

```
void Interpreter::visit(parser::ASTDeclarationNode *declarationNode) {
      // Visit the expression to get the current Type and Current Id
      declarationNode -> exprNode -> accept(this);
      // Now we have an updated current type and id
      // Create a variable with this information

      // Insert the new variable
      if(currentType == "int"){
            intTable.insert ( . . . )
            . . .
      }

      if(function){
            toPop.emplace_back(
            std::make_pair(
                  currentType,
                  declarationNode -> identifier -> identifier));
      }
}
```

Here if function is true, the interpreter makes sure to remember that this variable needs to popped once the function finishes.

Variable Assignment
Assignment is similar to declaration in its approach.

```
void Interpreter::visit(parser::ASTAssignmentNode *assignmentNode) {
      assignmentNode -> identifier -> accept(this);
      // We can update the type and identifier local variables
      // These two variables define the found variable
      std::string type = currentType;
      std::string id = currentID;
      // Visit the expression to get the current Type and Current Id
      assignmentNode -> exprNode -> accept(this);
      // Now we have an updated current type and id
      // These two variables define what we will give id
      // Replace the variable value
      // Replacement can be done by popping then inserting
      if(currentType == "int"){
            intTable.insert ( . . . )
      . . .
      }
      if(function){
            toPop.emplace_back(
                  std::make_pair(
                        currentType,
                        assignmentNode -> identifier -> identifier));
      }
}
```

Lastly when a function block visit is concluded the variables in toPop are popped.

```
for (const auto& pair : toPop){
      /*
      * Now we pop the variables
      */
      if(pair.first == "int"){
            intTable.pop_back(pair.second);
      }else if(pair.first == "float"){
            floatTable.pop_back(pair.second);
      }else if(pair.first == "bool"){
            boolTable.pop_back(pair.second);
      }else if(pair.first == "string"){
            stringTable.pop_back(pair.second);
      }
}
toPop = std::vector<std::pair<std::string, std::string>>();
```

Testing

Test 1

~omitted as failure is already proven

Test 2

~omitted as failure is already proven

Test 3

```
\cmake-build-debug\TeaLang.exe" -i -p ../Test3.tlng
50625

Process finished with exit code 0
```

Test 4

~omitted as failure is already proven

Test 5

~omitted as failure is already proven

Test 6

```
\cmake-build-debug\TeaLang.exe" -i -p ../Test6.tlng
6.25
true
false
3.28333
HelloWorld
0
1
2
3
4
5
6
7
8
9
0
1
2
3
4
5
6
7
8
9

Process finished with exit code 0
```

Conclusion

To build and run the complete interpreter build and run sh files are included with the submission.

All the required tasks have been implemented and documented successfully. Groundworks for the Tea2Lang update have been laid out. Overall, this assignment was a very interesting learning experience.