Compiler Theory and Practice
Course Assignment Part 2

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

Contents

Introduction

This first project describes the implementation and development of the Tea2Lang update for the TeaLang interpreter and so will mostly focus on the changes implemented. The interpreter was coded in C++20 and built using CMake version 3.16. The Tea2Lang language follows the EBNF included on page 2. This document documents the changes done on each part of the interpreter and assignment in detail, providing source code explanation, design details, figures, tables, and test cases. The update features: support for 2 new types, char and auto, arrays, function overloading and structs. Tea2Lanf is a fork of TeaLang and has the Tealang repository set as an upstream upstream master. This configuration allowed bug fixes and even continued development for TeaLang while working on Tea2Lang as pulling updates from TeaLang to Tea2Lang was made simple via the ‘git pull upstream master ’ command. Note, pushing the changes of Tea2Lang to TeaLang was never attempted.

Test Cases

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

Test Case 1 – Auto Declaration

```
let x : auto = true;

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

Test Case 2 – Chars and Arrays

```
let x : char = 'x';
let arr[3] : char = {'a', 'r', 'r'};

let i : auto = 0;
while( i < 3 ){
      print arr[i];
      i = i + 1;
}
```

Test Case 3 – Recursion Bug

```
float Factorial( x : float) {
    if (x >= 1.0){
            return x * Factorial(x - 1.0);
    }else{
            return 1.0;
    }
}

print Factorial(5.0);

float Factorial(x : float, toMakeAnExample : float) {
    if (x >= 1.0){
            return x * Factorial(x - 1.0);
    }else{
            return 1.0;
    }
}


print Factorial(5.0, 1.0);
```

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 With Structs

```
auto XGreaterY(toCompare[] : float){ // 1
    let ans : auto = false;
    if(toCompare[0] > toCompare[1])   {
        ans=true;
    }
    return ans;
}

auto XGreaterY(x : int, y : int){
    if(x>y){ //10
        return true;
    }else{
        return false;
    }
}

float Average(toAverage[] : float , count : float){
    let total : float = 0.0;
    for(let i : float = 0.0; i < count; i=i+1.0){
        total = total + toAverage[i]; //20
    }
    return total / count;
    }

let arr1[2] : float;
let arr2[4] : float = {2.4, 2.8, 10.4, 12.1};

arr1[0] = 22.4;
arr1[1] = 6.25;
print arr1[1]; //6.25 //30
print XGreaterY(arr1); //true
print XGreaterY(2,3); //false
print Average(arr2, 4.0); //6.92

tlstruct Vector{
    let self : Vector;
    let x : float = 0.0;
    let y : float = 0.0;
    let z : float = 0.0;

    Vector Scale(s : float){ //40
        x=x*s;
        y=y*s;
        z=z*s;
        return self;//Because functions always return something
    }
    Vector Translate(tx : float,ty : float,tz : float){
        x=x+tx;
        y=y+ty;
        z=z+tz;
        return self;//Language does not support void //50
    }
}

Vector Add(v1 : Vector, v2 : Vector){
```

```
    let v3 : Vector;
    v3.x = v1.x + v2.x;
    v3.y = v1.y + v2.y;
    v3.z = v1.z + v2.z;
    return v3;
} //60

let v1 : Vector;
v1.x=1.0;
v1.y=2.0;
v1.z=2.0;

let v2 : Vector;
v2.x=2.0;
v2.y=1.2;
v2.z=0.0; //70

let v3 : Vector = Add(v1,v2);
print v3.x;//3.0
print v3.y;//3.2
print v3.z;//2.0
```

Test Case 7 – Known Execution Struct-Function Bug

```
auto XGreaterY(toCompare[] : float){ // 1
    let ans : auto = false;
    if(toCompare[0] > toCompare[1])   {
        ans=true;
    }
    return ans;
}

auto XGreaterY(x : int, y : int){
    if(x>y){ //10
        return true;
    }else{
        return false;
    }
}

float Average(toAverage[] : float , count : float){
    let total : float = 0.0;
    for(let i : float = 0.0; i < count; i=i+1.0){
        total = total + toAverage[i]; //20
    }
    return total / count;
    }

let arr1[2] : float;
let arr2[4] : float = {2.4, 2.8, 10.4, 12.1};

arr1[0] = 22.4;
arr1[1] = 6.25;
print arr1[1]; //6.25 //30
print XGreaterY(arr1); //true
print XGreaterY(2,3); //false
print Average(arr2, 4.0); //6.92

tlstruct Vector{
    let self : Vector;
    let x : float = 0.0;
    let y : float = 0.0;
    let z : float = 0.0;

    Vector Scale(s : float){ //40
        x=x*s;
        y=y*s;
        z=z*s;
        return self;//Because functions always return something
    }
    Vector Translate(tx : float,ty : float,tz : float){
        x=x+tx;
        y=y+ty;
        z=z+tz;
        return self;//Language does not support void //50
    }
}

Vector Add(v1 : Vector, v2 : Vector){
```

```
    let v3 : Vector;
    v3.x = v1.x + v2.x;
    v3.y = v1.y + v2.y;
    v3.z = v1.z + v2.z;
    return v3;
} //60

let v1 : Vector;
v1.x=1.0;
v1.y=2.0;
v1.z=2.0;

let v2 : Vector;
v2.x=2.0;
v2.y=1.2;
v2.z=0.0; //70

let v3 : Vector = Add(v1,v2);
print v3.x;//3.0
print v3.y;//3.2
print v3.z;//2.0
print v3.z;//2.0
v3.Translate(1.0, 1.0, 1.0);
print v3.x;//4.0
print v3.y;//4.2
print v3.z;//3.0
// Begin of bug
let v4 : Vector = v3;
print v4.x;//4.0
print v4.y;//4.2
print v4.z;//3.0
let arrV[2] : Vector = { v1, v2 };
print arrV[1].x;
```

Test Case 8 – TeaLang Large Program

```
// Large program - Successful

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' | `char' | `auto'
<BooleanLiteral> ::= `true' | `false
<IntegerLiteral> ::= <Digit> \{ <Digit> \}
<FloatLiteral> ::= <Digit> \{ <Digit> \} `.' <Digit> \{ <Digit> \}
<StringLiteral> ::= `"' \{ <Printable> \} `"'
<CharLiteral> ::= `’' \{ <Printable> \} `’'
<Literal> ::= <BooleanLiteral> |
              <IntegerLiteral> |
              <FloatLiteral>   |
              <StringLiteral>  |
              <CharLiteral>

<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> \}
<AccessOperator> ::= `[' { Expression } ‘]’
<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> \} `}'
<Struct> :: ‘lsStruct’ <Block>
<Program> ::= \{ <Statement> \}
```

Lexer

The first update was done to the 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.

New Design or Changes

A required update was to design the lexer in a DFSA – table driven way. This was achieved by designing a new complete DFSA for the Tea2Lang language. Figure 1 shows a subset of this DFSA, which describes the possible transitions from one state to another in order to create a char literal at state 3. Transitions between states that are not defined here default to state 24.

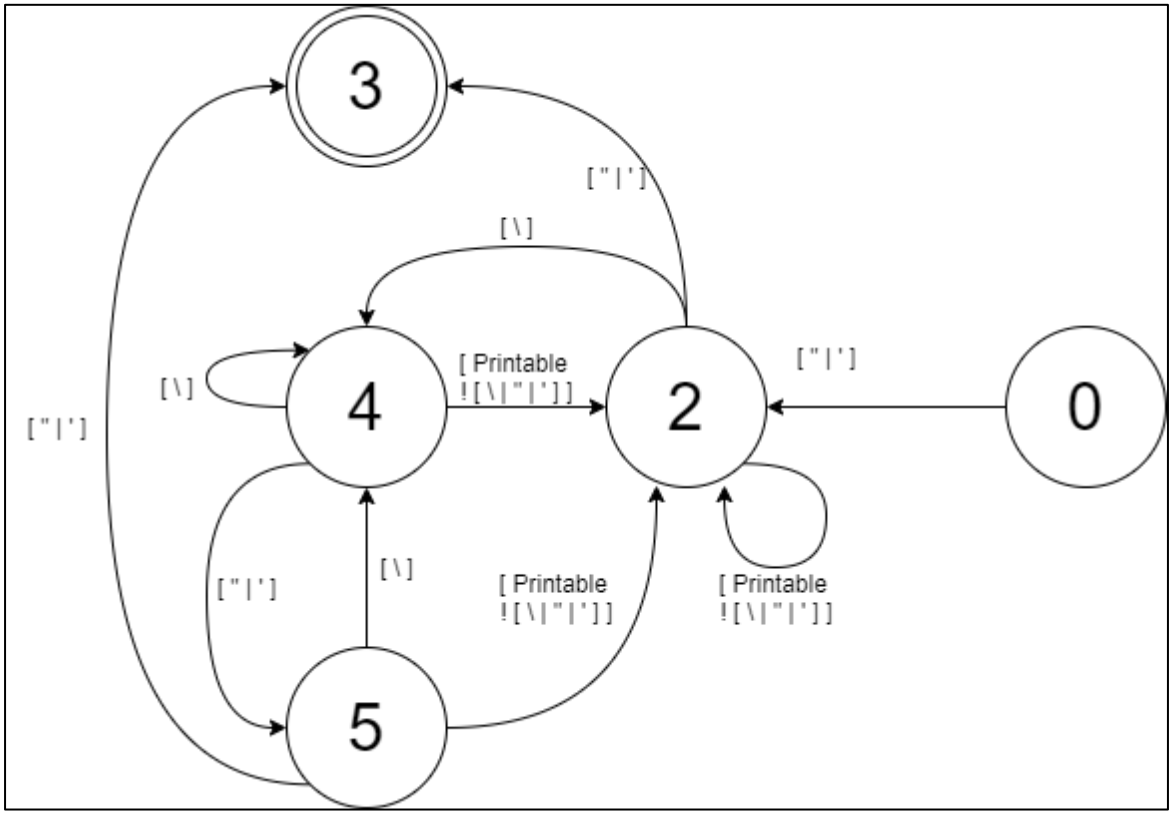

Figure 1 DFSA That Handles String and Char literals

A complete DFSA can be seen in Figure 2 which shows the complete DFSA for the entire Tea2Lang language. For the sake of neatness transition to state 24 are omitted visually.

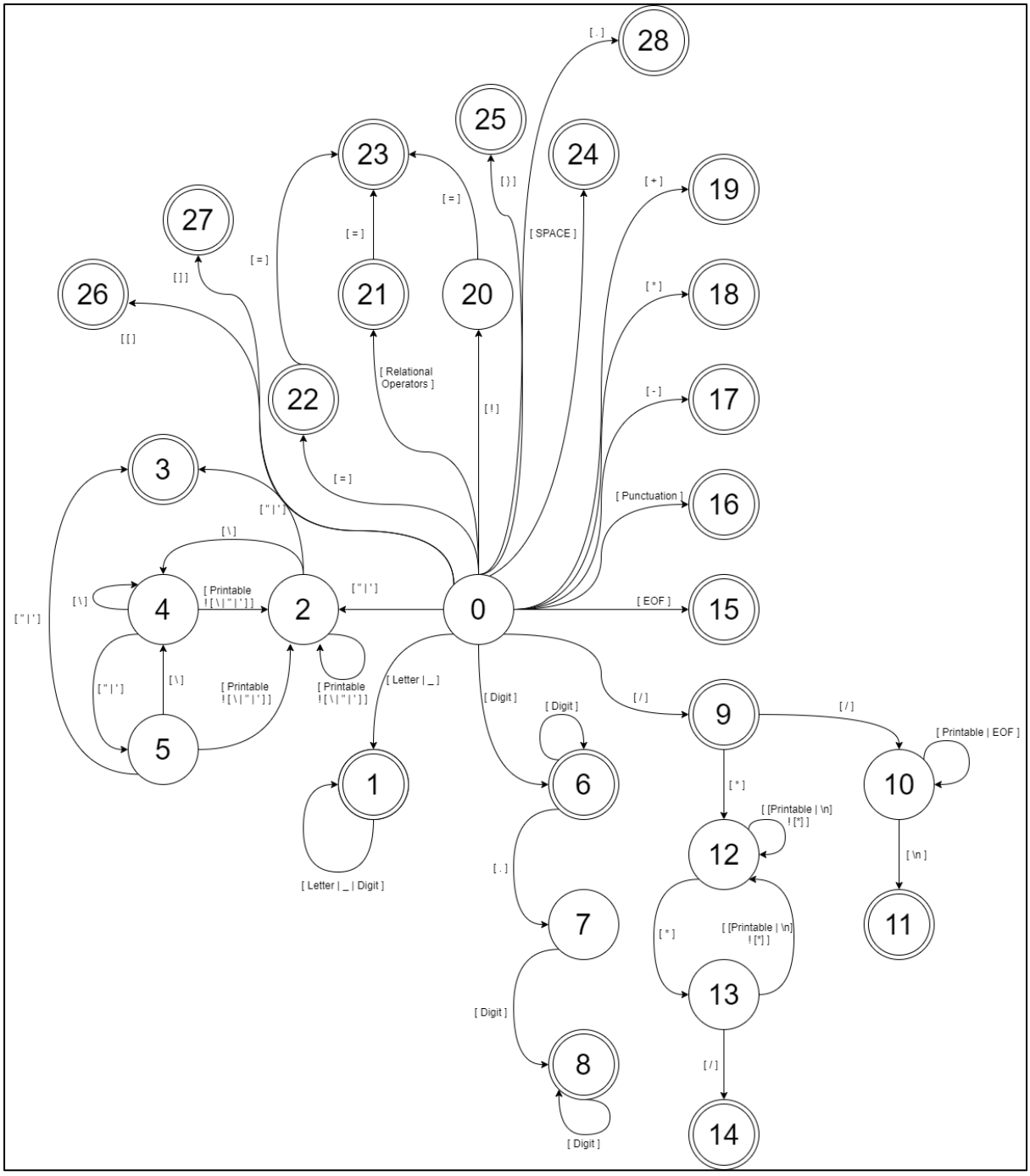

Figure 2 DFSA for the Tea2Lang Programming Language

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. A detailed definition of this quintuple can be found in the report for TeaLang.

Updates for the 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.

Table 1 New Characters to New Transition Types

Label	Condition	Transition Type	Assigned ID
Opening Square	‘]’	OPENING_SQUARE	TT19
Closing Square	‘[’	CLOSING_SQUARE	TT20
Apostrophe	‘ ‘ ’	APOSTROPHE	TT21

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. The added 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. Transitions of the pre-defined TeaLang alphabet results in entering the S24 state.

Table 2 DFSA Transition Table Added Columns

SID	TT19	TT20	TT21
S0	S26	S27	S28
S1	S24	S24	S24
S2	S2	S2	S2
S3	S24	S24	S24
S4	S2	S2	S2
S5	S2	S2	S2
S6	S24	S24	S24
S7	S24	S24	S24
S8	S24	S24	S24
S9	S24	S24	S24
S10	S10	S10	S10
S11	S24	S24	S24
S12	S12	S12	S12
S13	S12	S12	S12
S14	S24	S24	S24
S15	S24	S24	S24
S16	S24	S24	S24
S17	S24	S24	S24
S18	S24	S24	S24
S19	S24	S24	S24
S20	S24	S24	S24
S21	S24	S24	S24
S22	S24	S24	S24
S23	S24	S24	S24
S24	S24	S24	S24
S25	S24	S24	S24
S26	S24	S24	S24
S27	S24	S24	S24
S28	S24	S24	S24

Implementation Updates

Due to the changes in the above tables and DFSA diagrams small implementation updates were necessary. Some new validation functions were created like isClosingSquare or isCharLiteral. For the latter function the same regex expression was used for validation, but the ‘ “ ’ character was swapped out for the ‘ ‘ ’ character. This is so that Tea2Lang can support multi-character chars such as ‘\n’ and ‘\b’.

```
bool isClosingSquare(char c) {
return c == ']';
}

std::regex charLiteral(R"(\'(\\.|[^"\\])*\')");
bool isChar(const std::string &s) {
return (std::regex_match(s, charLiteral));
}
```

Token Class

The Token class is updated in the determineTokenType function to include transitions from the 26, 27 and 28 states.

```
TOKEN_TYPE fromState26(const std::string &s) {
// Can only be Closing Square
      if (isOpeningSquare(s)) return TOK_OPENING_SQUARE;
      return TOK_INVALID;
}

TOKEN_TYPE fromState27(const std::string &s) {
// Can only be Closing Square
      if (isClosingSquare(s)) return TOK_CLOSING_SQUARE;
      return TOK_INVALID;
}

TOKEN_TYPE fromState28(const std::string &s) {
// Can only be a fullstop
      if (isFullstop(s)) return TOK_FULLSTOP;
      return TOK_INVALID;
}
```

Lexer class

The Lexer class has its finalStates vector updated to include the new final states shown in Figure 2. The δ function is also included to accommodate for the new transitions given by determineTokenType. For example, the FULLSTOP case is updated and the APOSTROPHE case is joined with the QUOTATION_MARK case. For the square brackets (required for arrays) their case is handled individually.

```
case OPENING_SQUARE:
(fromState == 0)
      ? current_state = 26 : current_state = 24;
      break;

case CLOSING_SQUARE:
(fromState == 0)
      ? current_state = 27 : current_state = 24;
      break;

case FULLSTOP:
      if (fromState == 6){
            current_state = 7;
}else if(fromState == 0){
            current_state = 28;
}else{
            current_state = 24;
}
      break;

case QUOTATION_MARK:
case APOSTROPHE:
if (fromState == 0) {
            current_state = 2;
} else if (fromState == 2 || fromState == 5) {
            current_state = 3;
} else if (fromState == 4) {
            current_state = 5;
} else {
            current_state = 24;
}
      break;
```

Lastly a new function isStruct is used throughout the interpreter.

```
bool isStruct(const std::string& type){
        return
            !(isFloatType(type) || isIntType(type) || isBoolType(type) ||
            isStringType(type) || isCharType(type) || isAutoType(type));
}
```

Testing

Test Case 1

```
\cmake-build-debug\TeaLang.exe" -l -p ../Test1.tl2ng

Process finished with exit code 0
```

Test Case 2

```
\cmake-build-debug\TeaLang.exe" -l -p ../Test2.tl2ng

Process finished with exit code 0
```

Test Case 3

```
\cmake-build-debug\TeaLang.exe" -l -p ../Test3.tl2ng

Process finished with exit code 0
```

Test Case 4

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

Process finished with exit code 0
```

Test Case 5

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

Process finished with exit code 0
```

Test Case 6

```
\cmake-build-debug\TeaLang.exe" -l -p ../Test6.tl2ng

Process finished with exit code 0
```

Test Case 7

```
\cmake-build-debug\TeaLang.exe" -l -p ../Test7.tl2ng

Process finished with exit code 0
```

Test Case 8

```
\cmake-build-debug\TeaLang.exe" -l -p ../Test8.tlng

Process finished with exit code 0
```

Parser

The second update regards 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.

New Design or Changes

The existing AST structure was kept, and no new design choices were made. New AST nodes were however implemented and also added to the visitor namespace visit function set. These include the ASTLiteralNode for that handles chars, the ASTArrayLiteralNode and the ASTStructNode. The ASTIdentifierNode class was also modified to be more robust.

Implementation Update

AST Classes

Below are the new AST nodes: ASTLiteralNode<char>, ASTArrayLiteralNode and ASTStructNode.

```
template <typename T>
class ASTLiteralNode : public ASTExprNode {
public:
ASTLiteralNode(T val, unsigned int lineNumber) :
                        val(val),
                        lineNumber(lineNumber)
                        {};
~ASTLiteralNode() = default;
      T val;
      unsigned int lineNumber;
      void accept(visitor::Visitor* v) override;
};

class ASTArrayLiteralNode : public ASTExprNode {
public:
ASTArrayLiteralNode(
std::vector<std::shared_ptr<ASTExprNode>> expressions,
unsigned int lineNumber) :
            expressions(std::move(expressions)),
            lineNumber(lineNumber)
{};
      ~ASTArrayLiteralNode() = default;
      std::vector<std::shared_ptr<ASTExprNode>> expressions;
      unsigned int lineNumber;
      void accept(visitor::Visitor* v) override;
};

class ASTStructNode : public ASTStatementNode {
public:
        ASTStructNode(
            std::shared_ptr<ASTIdentifierNode> identifier,
            std::shared_ptr<ASTBlockNode> structBlock,
            unsigned int lineNumber) :
identifier(std::move(identifier)),
      structBlock(std::move(structBlock)),
      lineNumber(lineNumber)
      {};
      ~ASTStructNode() = default;

std::shared_ptr<ASTIdentifierNode> identifier;
      std::shared_ptr<ASTBlockNode> structBlock;
      unsigned int lineNumber;
      void accept(visitor::Visitor* v) override;
};
```

Also included is the ASTIdentifierNode update. This update features new functionality in the form of getID, getChild, isEmpty and a new constructor that takes in a shared pointer. This added functionality is needed for the semantic and interpreter to differentiate between array and non-array identifiers. The rule used is that if ilocExprNode is not defined than the identifier is not referring to an array. The added ASTIdentifierNode child would represent a child of the identifier being referenced. For example, v1.x or if a struct is defined inside a struct, v1.v2.v.x, since child itself is an Identifier node.

```
class ASTIdentifierNode : public ASTExprNode {
private:
std::shared_ptr<ASTIdentifierNode> child;
public:
ASTIdentifierNode(
      std::string identifier,
      std::shared_ptr<ASTIdentifierNode> child,
      std::shared_ptr<ASTExprNode> ilocExprNode,
      unsigned int lineNumber) :
            identifier(std::move(identifier)),
            child(std::move(child)),
            ilocExprNode(std::move(ilocExprNode)),
            lineNumber(lineNumber)
{};

      explicit ASTIdentifierNode(
      const std::shared_ptr<ASTIdentifierNode>& identifier) :
            identifier(identifier->identifier),
            child(identifier->child),
            ilocExprNode(identifier->ilocExprNode),
            lineNumber(identifier->lineNumber)
{};

      ~ASTIdentifierNode() = default;

      std::shared_ptr<ASTExprNode> ilocExprNode;
      std::string identifier;
      unsigned int lineNumber;

      std::string getID(){
            if(child != nullptr)
                  if(!child->isEmpty())
                        return identifier + "." + child->getID();
            return identifier;
}
std::shared_ptr<ASTIdentifierNode> getChild(){
            return child;
}
      bool isEmpty(){
            return identifier.empty();
}
void accept(visitor::Visitor* v) override;
};
```

Parse Function Set

Additions to the parse function set were made to parse new tokens and generate new AST nodes. The parseIdentifier function is also updated to support child identifiers and the access operator. The parse factor was also updated to expect the tokens TOK_CHAR and TOK_OPENING_CURLY for char and array literals. The parseType was also updated to accommodate the char, auto and user defined struct types. The changes to parseTerm and parse Identifer and the new parseArrayLiteral function are included below.

```
In parseTerm

. . .
case lexer::TOK_CHAR:
return
      std::make_shared<ASTLiteralNode<char>>(char(currentToken.value.at(1))
      , lineNumber);
. . .
case lexer::TOK_OPENING_CURLY:
return parseArrayLiteral();
. . .
```

```
std::shared_ptr<ASTArrayLiteralNode> Parser::parseArrayLiteral(){
// Determine line number
      unsigned int lineNumber = currentToken.lineNumber;
      // current token is the curly ({) bracket
      // move over first curly bracket
      moveTokenWindow();
      // Now we should be able to get an expression
      auto expressions = std::vector<std::shared_ptr<ASTExprNode>>();
      // Add first param
      expressions.emplace_back(parseExpression());
      // If next token is a comma there are more
      while (nextToken.type == lexer::TOK_COMMA) {
            // Move current token, to token after comma
            moveTokenWindow(2);
            // Add this token
            expressions.emplace_back(parseExpression());
}
// Current token is on the last param, we need to move beyond that
// to get the closing }
      moveTokenWindow();
      if(currentToken.type != lexer::TOK_CLOSING_CURLY){
            throw std::runtime_error();
}
      return
std::make_shared<ASTArrayLiteralNode>(expressions, lineNumber);
}
```

```
std::shared_ptr<ASTIdentifierNode> Parser::parseIdentifier() {
// Determine line number
      unsigned int lineNumber = currentToken.lineNumber;
      // current value is identifier
      std::string identifier = currentToken.value;

      // now we check if the variable is an array
      auto ilocExprNode = std::shared_ptr<ASTExprNode>();
      if (nextToken.type == lexer::TOK_OPENING_SQUARE) {
            // Get next token (after [)
            moveTokenWindow(2);
      // get expression if the current token isn't a closing square
//(which it can)
            if(currentToken.type != lexer::TOK_CLOSING_SQUARE){
            ilocExprNode = parseExpression();
                  // Get next token
                  moveTokenWindow();
                  // ensure proper syntax
                  if(currentToken.type != lexer::TOK_CLOSING_SQUARE){
                        throw std::runtime_error();
                  }
            }
}

      auto child = std::shared_ptr<ASTIdentifierNode>();
      // Check if next token is '.'
      if (nextToken.type == lexer::TOK_FULLSTOP) {
            moveTokenWindow(2);
            child = parseIdentifier();
}

      return std::make_shared<ASTIdentifierNode>(
                  identifier, child, ilocExprNode, lineNumber);
}
```

Testing

Test Case 1

```
\cmake-build-debug\TeaLang.exe" -p -p ../Test1.tl2ng

Process finished with exit code 0
```

Test Case 2

```
\cmake-build-debug\TeaLang.exe" -p -p ../Test2.tl2ng

Process finished with exit code 0
```

Test Case 3

```
\cmake-build-debug\TeaLang.exe" -p -p ../Test3.tl2ng

Process finished with exit code 0
```

Test Case 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 Case 5

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

Process finished with exit code 0
```

Test Case 6

```
\cmake-build-debug\TeaLang.exe" -p -p ../Test6.tl2ng

Process finished with exit code 0
```

Test Case 7

```
\cmake-build-debug\TeaLang.exe" -p -p ../Test7.tl2ng

Process finished with exit code 0
```

Test Case 8

```
\cmake-build-debug\TeaLang.exe" -p -p ../Test8.tlng

Process finished with exit code 0
```

XML Generation

The update to XML generation was minimal only needed for the new AST nodes and the updated ASTIdentifierNode.

No new design choices or changes were made

Implementation Update

The implementation of the new AST nodes is standard, the visit to the ASTStructNode is included below. Due to the changes in the ASTIdentifeirNode an update was required, this is also included below with new cases added that handles the new attributes.

```
void XMLVisitor::visit(parser::ASTStructNode *structNode) {
// Add initial <struct> tag
      xmlfile << indentation() << "<struct>" << std::endl;
      // Add indentation level
      indentationLevel++;
      // Add identifier
xmlfile << indentation() << "<id>"
<< structNode->identifier->getID() << "</id>" << std::endl;
      // Add <structBlock> tag
      xmlfile << indentation() << "<structBlock>" << std::endl;
      // Add indentation level
      indentationLevel++;
      // structBlock
      structNode->structBlock->accept(this);
      // Remove indentation level
      indentationLevel--;
      // Add closing tag
      xmlfile << indentation() << "</structBlock>" << std::endl;
      // Remove indentation level
      indentationLevel--;
      // Add closing tag
      xmlfile << indentation() << "</struct>" << std::endl;
    }
```

```
void XMLVisitor::visit(parser::ASTIdentifierNode *identifierNode) {
// Add initial <id> tag
      xmlfile << indentation() << "<id>" << std::endl;
      // Add value
      indentationLevel++;
      xmlfile << indentation() << identifierNode->getID() << std::endl;
      if(identifierNode->ilocExprNode != nullptr){
            identifierNode->ilocExprNode->accept(this);
}
      indentationLevel--;
      // Add closing tag
xmlfile << indentation() << "</id>" << std::endl;
}
```

Testing

Test Case 1

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

Test Case 2

```
<program>
    <declaration>
        <id type = "char">x</id>
        <char>121</char>
    </declaration>
    <declaration>
        <id type = "char">arr</id>
        <Array>
            <item>
                <char>97</char>
            </item>
            <item>
                <char>114</char>
            </item>
            <item>
                <char>114</char>
            </item>
        </Array>
    </declaration>
    <declaration>
        <id type = "auto">i</id>
        <int>0</int>
    </declaration>
    <while>
        <condition>
            <bin op = "&lt;">
                <id>
                    i
                </id>
                <int>3</int>
            </bin>
        </condition>
        <block>
            <print>
                <id>
                    arr
                    <id>
                        i
                    </id>
                </id>
            </print>
            <assign>
                <id>i</id>
                <bin op = "+">
                    <id>
                        i
                    </id>
                    <int>1</int>
                </bin>
            </assign>
        </block>
    </while>
</program>
```

Test Case 3

```
<program>
    <functionDeclaration type = "float">
        <id>Factorial</id>
        <param type = "float">x</param>
        <block>
            <if>
                <condition>
                    <bin op = "&gt;=">
                        <id>
                            x
                        </id>
                        <float>1.000000</float>
                    </bin>
                </condition>
                <ifBlock>
                    <block>
                        <return>
                            <bin op = "*">
                                <id>
                                    x
                                </id>
                                <functionEcall>
                                    <id>Factorial</id>
                                    <param>
                                        <bin op = "-">
                                            <id>
                                                x
                                            </id>
                                            <float>1.000000</float>
                                        </bin>
                                    </param>
                                </functionEcall>
                            </bin>
                        </return>
                    </block>
                </ifBlock>
            <elseBlock>
                <block>
                    <return>
                        <float>1.000000</float>
                    </return>
                </block>
            </elseBlock>
            </if>
        </block>
    </functionDeclaration>
    <print>
        <functionEcall>
            <id>Factorial</id>
            <param>
                <float>5.000000</float>
            </param>
        </functionEcall>
    </print>
    <functionDeclaration type = "float">
```

```
        <id>Factorial</id>
        <param type = "float">x</param>
        <param type = "float">toMakeAnExample</param>
        <block>
            <if>
                <condition>
                    <bin op = "&gt;=">
                        <id>
                            x
                        </id>
                        <float>1.000000</float>
                    </bin>
                </condition>
                <ifBlock>
                    <block>
                        <return>
                            <bin op = "*">
                                <id>
                                    x
                                </id>
                                <functionEcall>
                                    <id>Factorial</id>
                                    <param>
                                        <bin op = "-">
                                            <id>
                                                x
                                            </id>
                                            <float>1.000000</float>
                                        </bin>
                                    </param>
                                </functionEcall>
                            </bin>
                        </return>
                    </block>
                </ifBlock>
            <elseBlock>
                <block>
                    <return>
                        <float>1.000000</float>
                    </return>
                </block>
            </elseBlock>
            </if>
        </block>
    </functionDeclaration>
    <print>
        <functionEcall>
            <id>Factorial</id>
            <param>
                <float>5.000000</float>
            </param>
            <param>
                <float>1.000000</float>
            </param>
        </functionEcall>
    </print>
</program>
```

Test Case 4

~Omitted as failure has already been proven

Test Case 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 Case 6

~ Omitted because of output size

Test Case 7

~ Omitted because of output size

Test Case 8

```
<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

A substantial update to the TeaLang Semantic Analyzer was required to check the semantical validity of the new programming features. The update includes struct, array, and char declaration and assignment, function overloading and the handling of the auto type. Implicit and automatic typecasting however are also not supported by Tea2Lang.

The TeaLang structure and usage of the semantic and visitor namespaces is retained. The update mostly concerns the implementation.

Implementation Update

Apart from adding to the visit set of functions the Variable class is updated by having the bool array attribute added. When true, the Tea2Lang interpreter recognizes the specified Variable as an array object. A new class, Struct is also defined here. A struct holds 4 attributes, a string identifier, a vector of child variables, a vector of child functions and the lineNumber. Functionality of Struct like Variable and Function is handled in the Scope class. A new function unique to Struct is the insertTo function. This function handles registering a Variable or Function with a Struct via inserting the Variable or Function to their respective vector.

```
class Struct{
public:
Struct(std::string identifier) :
            identifier(std::move(identifier)),
            variables(std::vector<Variable>()),
            functions(std::vector<Function>()),
            lineNumber()
{};
      ~Struct() = default;

      std::string identifier;
      std::vector<Variable> variables;
      std::vector<Function> functions;
      unsigned int lineNumber;

      void insert(const Variable& v);
      void insert(const Function& f);
      void defineLineNumber(unsigned int lineNumber);
};
```

```
void Scope::insertTo(const Struct& s, const Variable& v){
auto result = find(s);
      if(!found(result)){
            throw StructInsertionException();
}
      auto cpy(result -> second);
      // add the new value
      cpy.insert(v);
      // remove the result
      structTable.erase(result);
      // insert the copy
      insert(cpy);
}
```

The update to the ASTIdentifierNode had unintended consequences that complicated the semantic analyses step. The main source of these issues generated originate from the child ASTIdentifierNode. Handling this recursive attribute was solved via the below algorithm.

```
auto parent = parser::ASTIdentifierNode(  identifierNode->identifier,
                                          identifierNode->getChild(),
                                          identifierNode->ilocExprNode,
                                          identifierNode->lineNumber);
auto child = identifierNode->getChild();
bool found = false;
while(child != nullptr){
// we have found a child
      // this means that the identifier of identifierNode must be a struct
      for(const auto& scope : scopes) {
            // First find the variable (remember identifierNode->identifier
            // is a variable of a type)
            auto result = scope->
                  find(semantic::Variable(parent.identifier));
            if(scope->found(result)) {
                  // we found it, now does it have a struct type?
                  if(!lexer::isStruct(result->second.type)){
                        throw std::runtime_error();
                  }
                  // get the struct
                  for(const auto& _scope : scopes) {
                        auto struct_result = _scope
                        ->find(semantic::Struct(result->second.type));
                        if(_scope->found(struct_result)) {
                              // found the struct
                              // go over its variables and verify
                              //child.identifier is there
                              for
                              (
                              auto var : struct_result->second.variables
                              )
                              {
                                    if(var.identifier == child->identifier)
                                    {
                                          //found
                                          found = true;
                                          currentType = var.type;
                                          break;
                                    }
                              }
                                    if(found) break;
                        }
                  }
                  if(found) {
                        found = false;
                  }else{
                        throw std::runtime_error();
            }
                        break;
            }
}
      parent = parser::ASTIdentifierNode(child);
      child = child->getChild();
      if(child == nullptr)
            return;
}
```

This iterative algorithm builds a struct -> struct -> variable/function path so that the actual type of the found end node is found.

The visits to ASTStructNode and ASTArrayLiteralNode are quite standard and do not deviate from the design of TeaLang.

Auto handling is done by checking the return type of an expression before finally inserting the Variable or Function. For the function declaration case, since the function would have already been declared, the existing Function is erased and re inserted with the proper type after the ASTBlockNode is visited.

```
if(functionDeclarationNode->type == "auto"){
// remove function and re insert it with the new type
      scope->erase(scope->find(f));
      scope->insert(semantic::Function(currentType,
            functionDeclarationNode->identifier->getID(), paramTypes,
            functionDeclarationNode->lineNumber));
// add this to the struct as well (if we are in a struct)
      if(!structID.empty()){
      structScope->insertTo(semantic::Struct(structID), f);
}
```

Finally function overloading is handled very simply by a small update to the functionTable object. Instead of using the Function identifier only as a key for the map, the identifier is coupled with the vector of parameter types to form a pair-key. The find found and insert function are updated to accommodate for this change.

```
std::map<std::pair<std::string, std::vector<std::string>>, Function>
functionTable;
```

Testing

Test Case 1

```
\cmake-build-debug\TeaLang.exe" -s -p ../Test1.tl2ng

Process finished with exit code 0
```

Test Case 2

```
\cmake-build-debug\TeaLang.exe" -s -p ../Test2.tl2ng

Process finished with exit code 0
```

Test Case 3

```
\cmake-build-debug\TeaLang.exe" -s -p ../Test3.tl2ng

Process finished with exit code 0
```

Test Case 4

~Omitted as failure has already been proven

Test Case 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 Case 6

```
\cmake-build-debug\TeaLang.exe" -s -p ../Test6.tl2ng

Process finished with exit code 0
```

Test Case 7

```
\cmake-build-debug\TeaLang.exe" -s -p ../Test7.tl2ng

Process finished with exit code 0
```

Test Case 8

```
\cmake-build-debug\TeaLang.exe" -s -p ../Test8.tlng

Process finished with exit code 0
```

Execution Pass

The final update required was to the final interpreter execution step. A new Popable class was created to be used in the toPop vector. Also, several new tables were included for the new char and array types. Note, the auto type is wholly handled by the Semantic Analyser and never reaches the execution step.

Implementation Update

The main headache source for this update came with Structs. Their dynamic nature goes against the static design of the TeaLang interpreter. Instead of re-writing the whole interpreter with a new design in mind, various small change throughout is made. An unorthodox decision was made on how to handle structs. Instead of creating a Struct class like Variable, each element within a struct would be declared as if it were a normal variable. This is best illustrated with an example.

```
// Take the vector Struct:

tlstruct Vector{
let self : Vector;
let x : float = 0.0;
      let y : float = 0.0
      let z : float = 0.0;

Vector Scale(s : float){
            x=x*s;
            y=y*s;
            z=z*s;
            return self;//Because functions always return something
}

Vector Translate(tx : float,ty : float,tz : float){
            x=x+tx;\n"
            y=y+ty;\n"
            z=z+tz;\n"
            return self;//Language does not support void
}"
}

// and the declaration line:

let v1 : Vector;
```

First the a visit to the ASTStructNode happens. The visit is very simple.

```
void Interpreter::visit(parser::ASTStructNode *structNode) {
// visit the block to define the functions
      structTable.insert(
                  std::pair<std::string, interpreter::Struct>
                  (
                  std::make_pair(structNode->identifier->getID(),
                  interpreter::Struct(structNode->identifier->getID(),
                  structNode->structBlock))));
}
```

The insert function here registers the Vector struct with structTable. Upon reaching declaration line. the following actions are taken. The Interpreter variable structID is set to "v1". The structID variable is added prior to variable and function declarations, usually it is empty so it has no effect. But now it is referencing v1. The next step is to visit the structNode attached to Vector. This visit is handled as if it is a function call almost, with the exception that the declarations and assignments done within are not popped (unless in an actual function). In the block the x, y, z variables and Scale and Translate Functions will be declared. This declaration is standard, but as mentioned earlier structID is added to their identifier. This results in 3 new float variables added to floatTable and 2 new functions, all starting with "v1.".

This can be exemplified by the ASTFunctionDeclarationVisit.

```
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,
                                    structID + functionDeclarationNode ->
                                    identifier -> getID(),
                                    paramTypes,
                                    paramIDs,
                                    functionDeclarationNode->functionBlock,
                                    functionDeclarationNode -> lineNumber)
);
}
```

Most of this update includes the handling of char and arrays was not an issue, the updated visit to the ASTPrintNode shows this elegant update.

```
void Interpreter::visit(parser::ASTPrintNode *printNode) {
// Visit expression node to get current type
      structID = "";
      printNode -> exprNode -> accept(this);
      if(currentType == "int"){
            std::cout <<
                  (array ?
                  intArrayTable.get(currentID).latestValue.at(iloc) :
                  intTable.get(currentID).latestValue) << std::endl;
}else if(currentType == "float"){
                  std::cout << (array ?
                  floatArrayTable.get(currentID).latestValue.at(iloc) :
                  floatTable.get(currentID).latestValue) << std::endl;
      }else if(currentType == "bool"){
                  std::cout << ((array ?
                  boolArrayTable.get(currentID).latestValue.at(iloc) :
                  boolTable.get(currentID).latestValue) ? "true" : "false")
                  << std::endl;
      }else if(currentType == "string"){
                  std::cout << (array ?
                  stringArrayTable.get(currentID).latestValue.at(iloc) :
                  stringTable.get(currentID).latestValue) << std::endl;
      }else if(currentType == "char"){
                  std::cout << (array ?
                  charArrayTable.get(currentID).latestValue.at(iloc) :
                  charTable.get(currentID).latestValue) << std::endl;
}
array = false;
}
```

Testing

Test Case 1

```
\cmake-build-debug\TeaLang.exe" -i -p ../Test1.tl2ng
10

Process finished with exit code 0
```

Test Case 2

```
\cmake-build-debug\TeaLang.exe" -i -p ../Test2.tl2ng
a
r
r

Process finished with exit code 0
```

Test Case 3

```
\cmake-build-debug\TeaLang.exe" -i -p ../Test3.tl2ng
0
0

Process finished with exit code 0
```

Test Case 4

~Omitted as failure has already been proven

Test Case 5

~Omitted as failure has already been proven

Test Case 6

```
\cmake-build-debug\TeaLang.exe" -i -p ../Test6.tl2ng
6.25
true
false
6.925
3
3.2
2

Process finished with exit code 0
```

Test Case 7

```
"C:\Users\Aiden Williams\CLionProjects\Tea2Lang-Interpeter-CPP20\cmake-
build-debug\TeaLang.exe" -i -p ../Test7.tl2ng
6.25
true
false
6.925
3
3.2
2
2
4
4.2
3

Process finished with exit code 0
```

Test Case 8

```
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, however there are a number of known bugs that affect the relation between structs and functions that appear at the interpreter execution stage. Most importantly, TeaLang was not compromised by this update.