



L-Università
ta' Malta

CPS2000 - Assignment

TinyLang Compiler

Nathan Bonavia Zammit

Bachelor of Science in

Information Technology (Honours)

(Artificial Intelligence)

June 2022

Plagiarism Form

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as “the unacknowledged use, as one’s own work, of work of another person, whether or not such work has been published” (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Nathan Bonavia Zammit

Student Name



Signature

Student Name

Signature

Student Name

Signature

Student Name

Signature

CPS2000

Course Code

TinyLang Compiler

Title of work submitted

18/06/2022

Date

Contents

Plagiarism Form	1
Introduction	3
Task 1: Table-Driven Lexer	4
Explanation	4
Implementation	5
Example.....	9
Task 2: Hand-crafted LL(k) Parser	11
Explanation	11
Implementation	14
Example.....	17
Task 3: AST XML Generation Pass	18
Explanation	18
Example.....	19
Task 4: Semantic Analysis Pass	21
Explanation	21
Implementation	21
Semantic Analysis Features.....	24
Example.....	25
Task 5: Interpreter Execution Pass.....	27
Explanation	27
Implementation	27
Interpreter Note.....	31
Example.....	31
Conclusion.....	32
References	33
Appendix A – List of Figures	34

Introduction

This is the report for the assignment of unit CPS2000 – Compiler Theory and Practice. A Compiler and Interpreter were created for the given programming language TinyLang, alongside a Lexer, a Parser, an XML generator, and a Semantic Analyser. The programming language that was used was C++ in conjunction with the VSCode IDE. Testing was performed on the Linux Subsystem for Windows and C++9 was used. The provided CMakeLists.txt was also used to compile and run the code.

No known errors were identified, and any compiler warnings were dealt with. Several input files can be found which showcase different aspects of the TinyLang programming language. Each task of the assignment was tackled and shown to be functioning properly.

The code is divided into separate folders to categorize them by functionality. Any errors are outputted properly, even showcasing at which line the error is occurring. The only limitation being, only one error at a time may be displayed.

Task 1: Table-Driven Lexer

Explanation

The Lexer will tokenize the input stream by taking in a character by character and performing tokenisation and grouping said characters into tokens. A token is a tuple, a lexeme or attribute. The line number is also added for debugging purposes, whether the debugging is done by the developer of the TinyLang language or anyone who wishes to use said language.

When it came to implementing the code, the entire file string is traversed and a vector of tokens will be created. This is done so the file can be closed and the string can then be discarded as soon as the Lexer is done with the file. As mentioned before, any error will be reported via the line number.

The Classifier Table is represented as follows:

1.	Singular Punctuation	+ - () { } ; : ,
2.	Exclamation Point	!
3.	Fullstop	.
4.	Digit	1-9
5.	Alpha	A-Z a-z _
6.	Forward Slash	/
7.	New-line	\n
8.	Asterisk	*
9.	Equals	=
10.	Whitespace/Tab or \r	' '\t \r
11.	Greater/Smaller than	< >
12.	Other	Anything else

The Following diagram is the state transition diagram DFA. 'Not X' refers to any other character which isn't X.

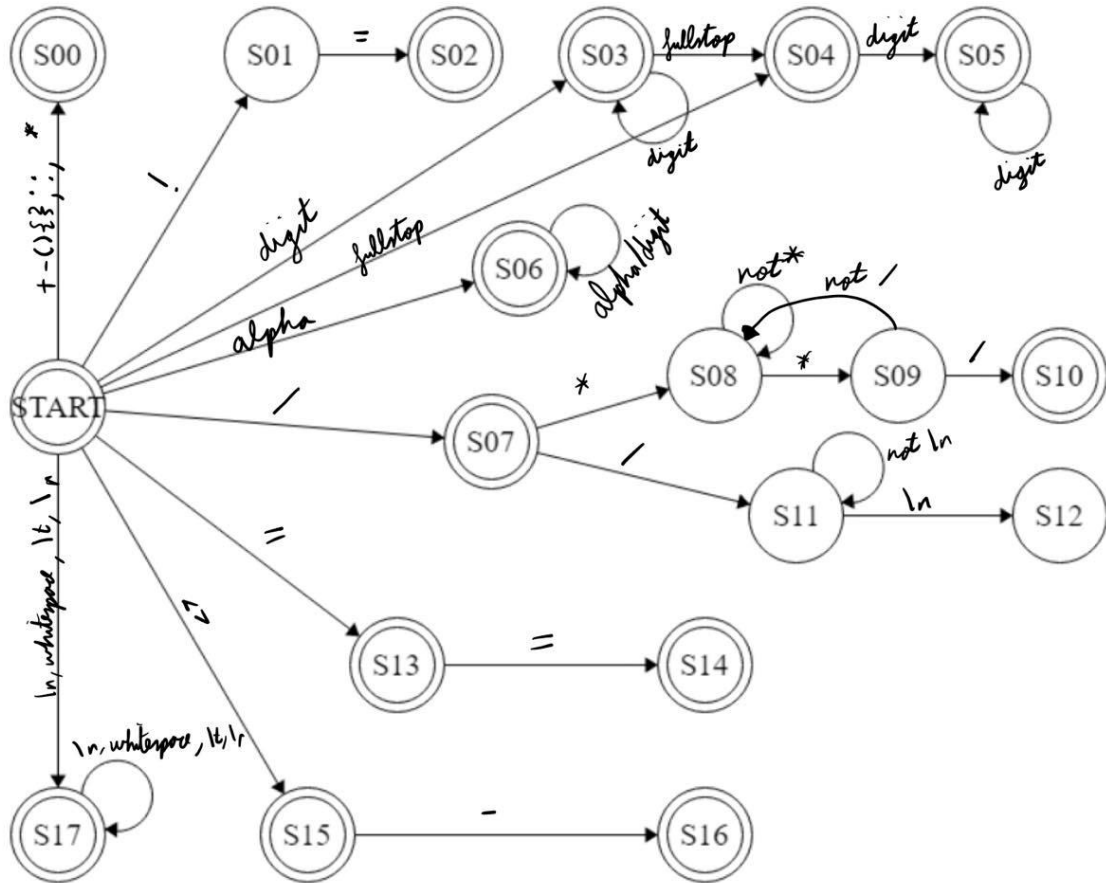


Fig.1. DFA for Lexer

Implementation

As a table it can be found in the code as seen below:

The comments around the table are used to help the user understand a bit better

```

State transitionTable[19][12] = {
  /*      0      1      2      3      4      5      6      7      8      9      10      11
  |  |  |  |  |  +-(){};:, !      .      digit  alpha  /      \n  *      =      space/tab\r  <>  other*/
  /*00 STA*/ {S00,    S01, S04, S03,    S06,    S07, S17, S00, S13, S17,    S15, ERR},
  /*01 S00*/ {ERR,    ERR, ERR, ERR,    ERR,    ERR, ERR, ERR, ERR, ERR,    ERR, ERR},
  /*02 S01*/ {ERR,    ERR, ERR, ERR,    ERR,    ERR, ERR, ERR, S02, ERR,    ERR, ERR},
  /*03 S02*/ {ERR,    ERR, ERR, ERR,    ERR,    ERR, ERR, ERR, ERR, ERR,    ERR, ERR},
  /*04 S03*/ {ERR,    ERR, S04, S03,    ERR,    ERR, ERR, ERR, ERR, ERR,    ERR, ERR},
  /*05 S04*/ {ERR,    ERR, ERR, S05,    ERR,    ERR, ERR, ERR, ERR, ERR,    ERR, ERR},
  /*06 S05*/ {ERR,    ERR, ERR, S05,    ERR,    ERR, ERR, ERR, ERR, ERR,    ERR, ERR},
  /*07 S06*/ {ERR,    ERR, ERR, S06,    S06,    ERR, ERR, ERR, ERR, ERR,    ERR, ERR},
  /*08 S07*/ {ERR,    ERR, ERR, ERR,    ERR,    S11, ERR, S08, ERR, ERR,    ERR, ERR},
  /*09 S08*/ {S08,    S08, S08, S08,    S08,    S08, S08, S09, S08, S08,    S08, S08},
  /*10 S09*/ {S08,    S08, S08, S08,    S08,    S10, S08, S08, S08, S08,    S08, S08},
  /*11 S10*/ {ERR,    ERR, ERR, ERR,    ERR,    ERR, ERR, ERR, ERR, ERR,    ERR, ERR},
  /*12 S11*/ {S11,    S11, S11, S11,    S11,    S11, S12, S11, S11, S11,    S11, S11},
  /*13 S12*/ {ERR,    ERR, ERR, ERR,    ERR,    ERR, ERR, ERR, ERR, ERR,    ERR, ERR},
  /*14 S13*/ {ERR,    ERR, ERR, ERR,    ERR,    ERR, ERR, ERR, S14, ERR,    ERR, ERR},
  /*15 S14*/ {ERR,    ERR, ERR, ERR,    ERR,    ERR, ERR, ERR, ERR, ERR,    ERR, ERR},
  /*16 S15*/ {ERR,    ERR, ERR, ERR,    ERR,    ERR, ERR, ERR, S16, ERR,    ERR, ERR},
  /*17 S16*/ {ERR,    ERR, ERR, ERR,    ERR,    ERR, ERR, ERR, ERR, ERR,    ERR, ERR},
  /*18 S17*/ {ERR,    ERR, ERR, ERR,    ERR,    ERR, S17, ERR, ERR, S17,    ERR, ERR}
};

```

Fig.2. Transition Table for Lexer

If at any point the program enters the ERR state, then the lexeme is over. If the current state is not a final state, then the error found will be reported, and Lexical Analysis shall stop at this point. However, if the program enters an ERR state, and said state is a final state, then the lexeme is kept and tokenization occurs. The token is as follows:

```

class Token{
public:
    static string TokenString[];
    //the Token Parameters
    TokenType type;
    string lexeme = "";
    float number;
    int lineNumber;

    // Constructor
    Token (TokenType _type, string _lexeme, float _number, int _lineNumber){
        type = _type;
        lexeme = _lexeme;
        number = _number;
        lineNumber = _lineNumber;
    }

    // helper method to neatly print the current token
    void printToken();
};

```

Fig.3. Token Class

The final state will always result in a token. Based on the lexeme, the following is a list of which states are related to which token:

1.	S0	PLUS, MINUS, OPEN_BRACKET, CLOSED_BRACKET, OPEN_BRACE, CLOSED_BRACE, COLON, SEMI_COLON, COMMA, TIMES
2.	S2	NE
3.	S3	INT
4.	S5	FLOAT
5.	S6	ID or one of the mykeywords[]
6.	S7	DIVISION
7.	S11	Discard '/*' and '*/' and return COMMENT
8.	S13	Discard '/' and return COMMENT
9.	S14	EQ
10.	S15	EQQ
11.	S16	GT, ST
12.	S17	GE, SE
13.	S18	Discard since it is white spaces or new lines or tabs or carriage returns only

An array of keywords is implemented in Token.cpp:


```

keyword_token mykeywords[] = {
    {"and", AND},
    {"or", OR},
    {"not", NOT},
    {"if", IF},
    {"else", ELSE},
    {"for", FOR},
    {"while", WHILE},
    {"fn", FN},
    {"return", RETURN},
    {"bool", TYPE_BOOL},
    {"float", TYPE_FLOAT},
    {"int", TYPE_INT},
    {"char", TYPE_CHAR},
    {"var", VAR},
    {"true", BOOL},
    {"false", BOOL},
    {"print", PRINT}
};

```

Fig.4. Array of Keywords

The lex() function which is the main method which obtains all required tokens from the file works as follows:

1. Initialise the start state.
2. Initialise the lexeme to an empty string.
3. Until the end of file is reached:
 - a. Read the next character from said file
 - b. Check to which column this character points to, then use the current state as the row value to go to the new state
4. If at an error state
 - a. Check if the lexeme is valid with the previous state before said error state
 - i. If Invalid report the error
 - ii. Otherwise, append the new token to the token vector, reset the lexeme and the state and go back to step 3.
5. Otherwise, set the state as the one obtained from 3b as the next state and append the character to the lexeme. Then go back to step 3.

Example

Via `printToken()` in the Token class and `printTokens()` in the Lexer class, we can attempt the following program:

```
1  < <= > >= == != and or not
2  if else for while fn return bool float int var
3  = + - * /
4  someid true false 12.3 .7 44 75
5  : ; ,
6  () {}
7  // hello world
8  /* hello
9  world
10 multiline */
11
12 //$ // comment this to avoid lexing error
```

Fig.5. Code run to test Lexer

This program is saved in the file “trial_Lexer.txt”. And when run showcases the following output:

```

Now Lexing:
Lexer has finished Lexing

Obtained tokens from lexer

<someid,ID,0>
<true,BOOL,1>
<false,BOOL,0>
<12.3,FLOAT,12.3>
<.7,FLOAT,0.7>
<44,INT,44>
<75,INT,75>
<<,ST,0>
<=<,SE,0>
<>,GT,0>
<=>,GE,0>
<==,EQ,0>
<!=,NE,0>
<and,AND,0>
<or,OR,0>
<not,NOT,0>
<if,IF,0>
<else,ELSE,0>
<for,FOR,0>
<while,WHILE,0>
<fn,FN,0>
<return,RETURN,0>
<bool,TYPE_BOOL,0>
<float,TYPE_FLOAT,0>
<int,TYPE_INT,0>
<var,COLON,0>
<=,EQ,0>
<+,PLUS,0>
<-,MINUS,0>
<*,TIMES,0>
</,DIVISION,0>
<:,SEMI_COLON,0>
<;,COMMA,0>
<,,OPEN_BRACKET,0>
<(),CLOSED_BRACKET,0>
<),OPEN_BRACE,0>
<{,CLOSED_BRACE,0>
<},COMMENT,0>

```

```

< hello world
,PRINT,0>
< hello
world
multiline ,PRINT,0>

```

Fig.6. "trial_Lexer.txt" output

After building the entire TinyLang language, the error "Error, could not match EQ at line 1" will be outputted when running "trial_Lexer.txt". This is because there is nothing to parse for the Parser which will be looked into later on.

Task 2: Hand-crafted LL(k) Parser

Explanation

The parser will take the tokens which were generated from the Lexer and then puts them into a parse tree based on the grammar. Below is the grammar re-written in order to make terminals and non-terminals more visible. To differentiate between them, terminals are underlined whilst non-terminals are enclosed in brackets. There are also square brackets which represent optional parts whilst curly brackets represent parts which can then be repeated. The peeking method is used whenever there is the use of options (sub-bullet points which represent |), [] or { }. This is done in order to avoid backtracking.

The parser uses something close to the first set in order to peek in order to determine the next production to choose.

An identifier non-terminal was created due to 'Factor' as it accepts a node, therefore, ID was enclosed with a non-terminal.

- Type
 - TYPE_FLOAT
 - TYPE_INT
 - TYPE_BOOL
- Literal
 - FLOAT
 - INT
 - BOOL
- Identifier
 - ID
- MultiplicativeOp
 - TIMES
 - DIVISION
 - AND
- AdditiveOp
 - PLUS
 - MINUS
 - OR
- RelationalOp

- ST
- GT
- EQQ
- NE
- SE
- GE
- ActualParams
 - (Expression) { COMMA (Expression) }
- FunctionCall
 - (Identifier) OPEN_BRACKET [(ActualParams)] CLOSED_BRACKET
- SubExpression
 - OPEN_BRACKET (Expression) CLOSED_BRACKET
- Unary
 - MINUS (Expression)
 - NOT (Expression)
- Factor
 - (Literal)
 - (Identifier)
 - (FunctionCall)
 - (SubExpression)
 - (Unary)
- Term
 - (Factor) { (MultiplicativeOP) (Factor) }
- SimpleExpression
 - (Term) { (AdditiveOp)(Term) }
- Expression
 - { (SimpleExpression) { (RelationalOp) (SimpleExpression) }
- Assignment
 - (Identifier) EQUALS (Expression)
- VariableDecl
 - VAR (Identifier) COLON (Type) EQ (Expression)
- PrintStatement
 - PRINT (Expression)
- ReturnStatement

- RETURN (Expression)
- IfStatement
 - IF OPEN BRACKET (Expression) CLOSED BRACKET (Block) [ELSE (Block)]
- ForStatement
 - FOR OPEN BRACKET [(VariableDecl)] SEMI COLON (Expression) SEMI COLON [(Assignment)] CLOSED BRACKET (Block)
- FormalParam
 - (Identifier) COLON (Type)
- FormalParams
 - FormalParam { COMMA (FormalParam) }
- FunctionDecl
 - FN (Identifier) OPEN BRACKET [(FormalParams)] CLOSED BRACKET COLON (Type) (Block)
- Statement
 - (VariableDecl) SEMI COLON
 - (Assignment) SEMI COLON
 - (PrintStatement) SEMI COLON
 - (IfStatement)
 - (ForStatement)
 - (ReturnStatement) SEMI COLON
 - (FunctionDecl)
 - (Block)
- Block
 - OPEN BRACE { Statement } CLOSED BRACE
- Program
 - { (Statement) }

Since it is an Abstract Syntax Tree, the following tokens are matched and then discarded by the Parser:

OPEN_BRACKET, CLOSED_BRACKET, OPEN_BRACE, CLOSED_BRACE, SEMI_COLON, COLON, COMMA, FN, VAR, RETURN, IF, ELSE, FOR, and PRINT.

As can be seen there are 26 different possible types of nodes. This is because the tree isn't fully abstract. The advantage of this is that operator precedence for expressions is handled automatically via the grammar. On the other hand, the disadvantage would be that it leads to more complexity as each type of node needs to be catered for.

Implementation

The 'Parser' class will receive the tokens and then store them in 'TokenManager' to go through the functions in said manager. The 'Parser' class also holds the root node for the recursive AST which will then be generated via the 'parse()' method.

```
class Parser{
private:
    TokenManager* tokenManager;
    ASTNode* tree;
public:
    Parser(vector<Token> *tokens);
    ~Parser();
    bool parse();
    ASTNode *getTree() {return tree;}
    int getTokenManagerIndex() {return tokenManager->index();}
};
```

Fig.7. The Parser Class

```
class TokenManager{
private:
    //The index for the token queue
    unsigned int tokenIndex = 0;

    //Token queue
    vector<Token> *tokens;

public:
    TokenManager(vector<Token> *_tokens) {tokens = _tokens; };
    Token *currentToken();
    Token *nextToken();
    Token *peekToken(int steps);
    Token *peekToken();
    Token *peekTokenUnsafe(int steps);
    int index() {return tokenIndex;}
};
```

Fig.8. The TokenManager class

The 26 ASTNode classes (27 if we're counting the abstract class which is used for polymorphism) all have a parse method to parse their own form accordingly, and then store what is parsed in their own way. An example would be the 'Type' ASTNode which will store token of type TYPE_FLOAT, TYPE_INT, TYPE_BOOL, and TYPE_CHAR. Meanwhile a program will contain a vector of statement nodes.

Each parse method for the ASTNodes can either return true or false in order to notify the callback recursion whether it has succeeded or not. However, since a predictive parser with follow sets is used, then they should all return true, otherwise parsing will simply fail and the error and token line are reported back to the user. The function 'match(TokenType)' is used to check if a token is matched, specifically when a bracket needs to be matched. If in any case the token type is not found, then an error is reported that the required token was not found and the parser will simply exit.

```
class ASTNodeIdentifier : virtual public ASTNode{
public:
    // constructor is the same as the parent
    ASTNodeIdentifier(TokenManager *tokenManager) : ASTNode(tokenManager) {};
    virtual ~ASTNodeIdentifier(){};
    virtual bool parse(); // will return true if parse was successful
    virtual void *accept(Visitor *v);

    Token* token;
};
```

Fig.9. Parse Method for ASTNodeIdentifier in the AST.h file

```
bool ASTNodeIdentifier::parse(){
    token = match(ID);
    return true;
}
```

Fig.10. Parse Method for ASTNodeIdentifier in the AST.cpp file

The process is as entails. The token is simply stored within the variable 'token', and it becomes a leaf of the tree. All leaves of the tree are tokens, whilst the internal nodes are ASTNodes.

A more complex example of this would be the ASTNodeIfStatement which is showcased below:

```
class ASTNodeIfStatement : virtual public ASTNode{
public:
    // constructor is the same as the parent
    ASTNodeIfStatement(TokenManager *tokenManager) : ASTNode(tokenManager) {};
    virtual ~ASTNodeIfStatement();
    virtual bool parse(); // will return true if parse was successful
    virtual void *accept(Visitor *v);

    ASTNode* expression = NULL;
    ASTNode* block = NULL;
    ASTNode* elseBlock = NULL;
};
```

Fig.11. Parse Method for ASTNodeIfStatement in the AST.h file

```
bool ASTNodeIfStatement::parse(){
    match(IF);
    match(OPEN_BRACKET);

    ASTNode *n = new ASTNodeExpression(tokenManager);
    if (n->parse() == false) return false;
    expression = n;

    match(CLOSED_BRACKET);

    ASTNode *n2 = new ASTNodeBlock(tokenManager);
    if (n2->parse() == false) return false;
    block = n2;

    if(tokenManager->peekTokenUnsafe(0) != nullptr){
        if(tokenManager->peekToken()->type == ELSE){
            match(ELSE);

            ASTNode *n3 = new ASTNodeBlock(tokenManager);
            if (n3->parse() == false) return false;
            elseBlock = n3;
        }
    }

    return true;
}

ASTNodeIfStatement::~~ASTNodeIfStatement(){
    delete expression;
    delete block;
    delete elseBlock;
}
```

Fig.12. Parse Method for ASTNodeIfStatement in the AST.cpp file

As can be seen an IF statement has 3 children and neither are leaves. Parsing starts by first matching an IF and OPEN_BRACKET and then discards them. Then it attempts to parse an ASTNodeExpression. If this fails then false is returned, otherwise, the expression node within the if node is set to the newly parsed expression node. This indicates how the parse method is recursive until leaves are found. The rest follows suit. The else block is optional, so the parser will first check if it should expect an else block by checking for an ELSE token. If it doesn't exist it is skipped, otherwise, it is set accordingly.

Inspiration for the AST.h and AST.cpp files were taken from a gitrepo for 'cppast' [1].

Example

The parse method will simply return true or false and as such is difficult to visualize the tree. Therefore, a visualisation process can be shown in Task 3: AST XML Generation Pass, for a proper visualisation of the tree.

Task 3: AST XML Generation Pass

Explanation

The visitor design pattern is used for the XML generator. Each ASTNode accepts the visitor class's 'visit' function via the use of the 'accept' function.

Implementation (Sub-Header)

An XMLVisitor class was created which contains a string stream called 'xml', which will contain the generated string after the tree is all visited. The 'numOfTabs' integer holds the number of indentations which should then be applied at each line.

The visit methods are of type 'void*' in order to cater for further visitors requiring to be type-casted to different types. For this visitor however, they will all return NULL or 0.

```
class XMLVisitor : virtual public Visitor{
public:
    XMLVisitor(){};
    virtual ~XMLVisitor(){};
    // All these return null since all is needed is to update the
    virtual void *visit(ASTNode*){ return 0; };
    virtual void *visit(ASTNodeType *n);
    virtual void *visit(ASTNodeLiteral *n);
    virtual void *visit(ASTNodeIdentifier *n);
    virtual void *visit(ASTNodeMultiplicativeOp *n);
    virtual void *visit(ASTNodeAdditiveOp *n);
    virtual void *visit(ASTNodeRelationalOp *n);
    virtual void *visit(ASTNodeActualParams *n);
    virtual void *visit(ASTNodeFunctionCall *n);
    virtual void *visit(ASTNodeSubExpression *n);
    virtual void *visit(ASTNodeUnary *n);
    virtual void *visit(ASTNodeFactor *n);
    virtual void *visit(ASTNodeTerm *n);
    virtual void *visit(ASTNodeSimpleExpression *n);
    virtual void *visit(ASTNodeExpression *n);
    virtual void *visit(ASTNodeAssignment *n);
    virtual void *visit(ASTNodeVariableDecl *n);
    virtual void *visit(ASTNodePrintStatement *n);
    virtual void *visit(ASTNodeReturnStatement *n);
    virtual void *visit(ASTNodeIfStatement *n);
    virtual void *visit(ASTNodeForStatement *n);
    virtual void *visit(ASTNodeFormalParam *n);
    virtual void *visit(ASTNodeFormalParams *n);
    virtual void *visit(ASTNodeFunctionDecl *n);
    virtual void *visit(ASTNodeStatement *n);
    virtual void *visit(ASTNodeBlock *n);
    virtual void *visit(ASTNodeProgram *n);
    void *trimXMLNewLines(); // remove empty lines from xml
    string getXML(){ return xml.str(); }
```

Fig.13 XMLVisitor class in Visitor.h file

```

    void *trimXMLNewLines(); // remove empty lines from xml
    string getXML(){ return xml.str(); }
private:
    stringstream xml;
    unsigned int numOfTabs = 0;
    string tabsString();
};

```

Fig.14 XMLVisitor class in Visitor.h file (second part)

```

void *XMLVisitor::visit(ASTNodeMultiplicativeOp *n){
    xml << "OP=\"\" << n->token->lexeme << "\"\"";
    return 0;
}

```

Fig.15. An XML visit for a leaf node accept

The above is for a multiplicative operator, therefore, '*' would be shown as 'OP = "*" '.

This method is also recursive and as such the program node iteratively goes through the statements and calls the accept statement for the visitor on them as well which in turn will create recursion.

```

void *XMLVisitor::visit(ASTNodeProgram *n){
    for(unsigned int i = 0; i < n->statements.size(); ++i){
        n->statements.at(i)->accept(this);
    }
    return 0;
}

```

Fig.16. Aforementioned program which is recursive

Example

For this example, the following was inputted as a program node:

```

for(var x:int = 0; x<10;x = x + 1){ // change minus to plus
    print(x);
}

```

Fig.17. XML infinite loop test

The output by the XML generator is as follows:

```
XML Representation:

<FOR>
  <VarDecl>
    <Var Type = "int">x</ID>
    <IntConst>0</IntConst>
  </VarDecl>
  <BinExprNode OP="<">
    x</ID>
    <IntConst>10</IntConst>
  </BinExprNode>
  <Assign>
    x</ID>
    BinExprNode OP="+">
      x</ID>
      <IntConst>1</IntConst>
    </BinExprNode>
  </Assign>
<DO>
  <Print>
    x</ID>
  </Print>
<ENDFOR>
```

Fig.18. XML Generation output

Task 4: Semantic Analysis Pass

Explanation

The main goal of semantic analysis is to perform type checking and scope checking. This is done by traversing the AST and then making the required checks at each node.

Implementation

This was done through a new visitor inherited class which was named 'SAVisitor'. It contains the following code in order to perform its job (alongside the visit methods).

```
vector<map<string, TokenType>> scope;
//add scope as the 0 index of the vector
void newScope();
void insert(string, TokenType); //in the current scope
void removeScope(); //remove the scope at position 0
//set as pointer to TokenType due to the need to make it return null
TokenType* lookup(string); //lookup starting from vector 0 and going down
//a map, mapping function names to their parameter types
map <string, vector<TokenType> > functions;
TokenType *currentFunctionType = nullptr;
int lineNumber = 0;

//Methods to help if function has a proper return statement in all paths
bool insideFor = false; //Do nothing while inside for
bool insideFunction = false; //Only applies if inside function
vector<bool> ifsReturn;
int ifsReturnIndex = -1;
bool goodReturn;
```

Fig.19. Code in 'Visitor.h' file to perform Semantic Analysis

The scope vector is treated as a stack. However, it was created as a vector in order to be iterated over easily. New entries are inserted at the front of the vector and the same is done with the deletions. Iteration starts from the front in order to let new entries cover old ones of the same name. This is known as scope shadowing. The scope, alongside the functions map, make the symbol table.

Regarding the visit classes, each one (although described as 'void*') has some form of concrete return value, and not all of them are the same. The functionality of each method and their return type is discussed as follows for each type of ASTNode visit method:

- Type : Type
 - Returns FLOAT, INT, CHAR, or BOOL depending on its token
- Literal : Type
 - Returns the type of its literal value
- Identifier : string (The lexeme of the identifier token)
 - Returns the name of the identifier
- MultiplicativeOp : MultOp Type (Example: TIMES, AND, etc.)
 - Returns the operator itself so that the expression node it belongs to can check that the operator supports the types it is operating upon. (Example: "5*3" is valid whilst "5 and 3" is not)
- AdditiveOp : AddOp Type (Example PLUS, OR, etc.)
 - Similar to MultiplicativeOp but with different operators
- RelationalOp : RelOp Type (Example EQ, ST, etc.)
 - Similar to MultiplicativeOp but with different operators
- ActualParams : Vector of Type
 - Iterates through all the parameters and returns a vector of all their types in order for them to be matched with the formal parameters by the FunctionDecl node.
- FunctionCall : Type
 - Validates that the function actually exists
 - Validates that the parameters provided are of the required type
 - If any of these are false, errors are reported and the program exits
- SubExpression : Type
 - Returns the type of expression it has
- Unary : Type
 - Will validate that the unary operator is applied to the proper type and returns the type of expression
- Factor : Type
 - Returns the type of node
- Term : Type
 - Validates that any operators are used properly and of the correct type
- SimpleExpression : Type
 - Validates that any operators are being used properly and are of the correct type
 - For example "5*8.9" returns float, whilst "4*false" will return an error
- Expression : Type

- Very similar to term but with relational operators instead
- Assignment : void
 - Validates the identifier exists
 - Validates the identifier's type and type of expression match
- VariableDecl : void
 - Makes sure that the types of the given type are not conflicting
 - Adds the identifier to the symbol table, with the given type.
 - If this already exists an error will occur
- PrintStatement : void
 - Verify that the given expression is correct
- ReturnStatement : void
 - Makes sure that the type of expression also matches the type of current function it is within by using the 'currentFunctionType' variable.
- IfStatement : void
 - Validates that the expression is of type bool
 - Starts the new scope
 - Validates the block
 - Pop the scope
 - Validates else block if the else block exists
- ForStatement : void
 - Creates the new scope
 - Performs and validates the variable declaration
 - Validates that the expression is of type bool
 - Validates the assignment
 - Validates the block
 - Pops the scope
- FormalParam : Type
 - Put each parameter in the current symbol table
 - Return the type of parameter
- FormalParams: Vector of Type
 - Iterate through FormalParam declaration and adds them to the vector
- FunctionDecl : void
 - Add a function to the functionParams map after starting the new scope and validating the FormalParams

- They are added in the scope within their own methods
- Set the currentFunctionType to the type of the function so ReturnStatement can know
- Add the function to the symbol table after validating that the size of the symbol is one
- Validate the block
- Close the scope
- Statement : void
 - Validate the statement
 - If it is a block
 - Start the new scope
- Block : void
 - Validate the list of statements
- Program : void
 - Validate the list of statements
 - Starts with function declarations

Scopes were not started and ended in the 'Block' node because of function declarations and the need to initialise the variables within the block.

Semantic Analysis Features

Error reports should be outputted if the code is not semantically correct. However, after trials, the code will simply stop after parsing. This is a feature which, due to time constraints had to be scrapped.

1. Type conversion from integers to floats is made available.
 - a. Float types can accept integers however integers cannot accept floats. This was possible through type checking.
2. The program is allowed to exit before it finishes all the lines. Therefore, return statements can be accepted from outside a function.
3. A function declaration is checked to make sure all code paths return a proper value.
 - a. If a return exists in an if but no corresponding return in an else or outside the if exists, then the function is declared semantically incorrect.
 - b. Returns in for loops are also not considered to be enough for a function to be valid for the semantic analyser.
 - c. This was possible through the use of a stack and a few global variables, in conjunction with checks when returns are made inside of functions.

4. Functions can only be defined in the outer scope
 5. Function can be used before they are declared, because their declaration is performed first.
- Therefore, no other variable anywhere in the program may use the function's name.

Example

The following code is semantically correct, therefore, when running it in our TinyLang language, the following is outputted:

```
var x:int = 9; // should be int(change 9.8 to 8)
var y:float = 9.8; // already defined (change x to y)
var z:bool = false and true; // + should be and/or

fn AddAndIncrement(x:float, y:float):float{
|   return (x+y)+1;
|
| }

fn something():int{
|   return -1;
|
| }

print AddAndIncrement(3, 5.3); // true should be some float
print something(); // incorrect parameters
```

Fig.20. Semantically correct code used

```
Now running Semantic Analysis:

Semantic Analysis was performed Successfully
```

Fig.21. Proof that the code was semantically correct

The following code is not semantically correct:

```
fn u():bool{
    if(4>6) {
        return true;
    }else{
        if(5>6){
            return true;
        }else{
            //return true;
        }
    }
}
```

Fig.22. Non-Semantically correct code

The expected output is for the compiler to complain since “//return true” will end up returning a null value, which is not a supported type. However, semantic analysis will simply not occur and the program stops after the parser as seen below:

```
<true,BOL,1>
<;,COMMA,0>
<},COMMENT,0>
<else,ELSE,0>
<{,CLOSED_BRACE,0>
<return true;
,PRINT,0>
<},COMMENT,0>
<},COMMENT,0>

Now Parsing
Reached end of file while Parsing
```

Fig.23. Semantic Analysis not even being attempted.

Task 5: Interpreter Execution Pass

Explanation

The goal of the interpreter is to execute the program line by line and report any run time errors if any are present. The symbol table is regenerated and this time it will contain identifiers and their contents besides just their types. Said content will also be changed as the program is executed.

Implementation

The visitor class “IVisitor” was created and contains the following:

```
private:
    vector<map<string, ValueType>> scope;
    //add scope as the 0 index of the vector
    void newScope();
    void insert(string, ValueType); //in the current scope
    void removeScope(); //remove the scope at position 0
    ValueType* lookup(string); //lookup starting from vector 0 and going down
    //a map, mapping function names to their parameter types
    map <string, ASTNode*> functions;
    int lineNumber = 0;

    bool performFunction = false;
    vector<ValueType> parameters;
    bool returnFromFunction = false;
    ValueType *returnValue = NULL;
};
```

Fig.24. Code for interpreter in the “Visitor.h” class

```
struct ValueType{
    void* value;
    //Bool, Int, Float, or Char (needed for conversion)
    TokenType type;
    ValueType(){};
    ValueType(void* _value, TokenType _type){
        value = _value;
        type = _type;
    };
};
```

Fig.25. ValueType is a struct which was created for this visitor.

'performFunction' is used to determine whether a function would be either declared or performed. 'parameters' is used to store the values of the parameters while they are being sent to a function. 'returnFromFunction' is set to be true when a return is found and not set back to false until a function call is ended. While this is set to true, no statement or block are executed. 'returnValue' is set too something by the return statement and then is set back to null after a function is returned.

The functionality of each node is as follows:

- Type : TokenType
 - Returns FLOAT, INT, CHAR, or BOOL depending on its token
- Literal : Value
 - Returns the value of the literal
- Identifier : string (The lexeme of the identifier token)
 - Returns the name of the identifier
- MultiplicativeOp : MultOp Type (Example: TIMES, AND, etc.)
 - Returns the operator itself so that the expression node it belongs can use it to operate on the values
- AdditiveOp : AddOp Type (Example PLUS, OR, etc.)
 - Similar to MultiplicatievOp but with Additive operators
- RelationalOp : Type (Example EQQ, ST, etc.)
 - Similar to MultiplicatievOp but with Relational operators
- ActualParams : Vector of Values
 - Gets value from each parameter and returns their vector
- FunctionCall : Value
 - Sets 'performFunction' to true
 - Sets the 'parameters' value to the values returned from the actual parameters
 - Gets function from 'functions' map and traverses it until a return is found
 - Resets the parameters value
 - Resets the 'returnValue' variable and returns it's previous value
- SubExpression : Value
 - Returns value of the containing expression
- Unary : value
 - Applies the unary operator
 - Returns the value after application
- Factor : Value
 - Returns value of containing factor

- If it is an ID, it will look up in the scope table
- Term : Value
 - Applies the operator
 - Returns the value after application
- SimpleExpression : Value
 - Applies the operator
 - Returns the value after application
- Expression : Value
 - Applies the operator
 - Returns the value after application
- Assignment : void
 - Change value of item in the symbol table
- VariableDecl : void
 - Add an item to the symbol table
- PrintStatement : void
 - Print the given expression to the screen
- ReturnStatement : void
 - Sets the 'returnValue' variable to the expression
 - Sets the 'returnFromFunction' variable to true in order for nothing to be computed until the program exits the function
- IfStatement : void
 - Computes the given expression
 - If it is true
 - Opens new scope
 - Visits the if-block
 - Closes scope
 - If it is false
 - Check if an else block exists
 - Opens new scope
 - Else block is visited
 - Closes scope
- ForStatement : void
 - Performs variable declaration
 - Start new scope

- Visit the block
 - End scope
- Performs variable assignment
 - Checks expression
 - Exit if false
 - Otherwise repeat block
- FormalParam : string
 - Returns name of the parameter
- FormalParams: Vector of strings
 - Returns all the names of the parameters
- FunctionDecl : void
 - If 'performFunction' is false
 - Add the function to the 'functions' map
 - If 'performFunction' is true
 - Open new scope
 - Assign the variables to the given parameters
 - Traverse through function block
 - Close block
 - Set 'returnFromFunction' to false
 - Return the returnValue
- Statement : void
 - Visits the statement
 - If it is a block
 - Start new scope
 - Visit block
 - End scope
- Block
 - Unless 'returnFromFunction' is set to true, iterate through the statements until 'returnfromFunction' is set to true.
- Program
 - Iterates through all statements until the end
 - Or until 'returnFromFunction' is set to true

Interpreter Note

1. The only runtime error that can occur is if division by zero is attempted. It is reported to the user if found.
2. The program itself can return a value. If a statement is found outside a function declaration it will return Null instead.
3. $3/2$ is 1.5 not 1 because everything, even Booleans, are treated as a float.
 - a. However if the statement **var x: int = 3/2** is found
 - i. It is valid, but x is set to .1
 - b. This was done to allow as much functionality as possible.

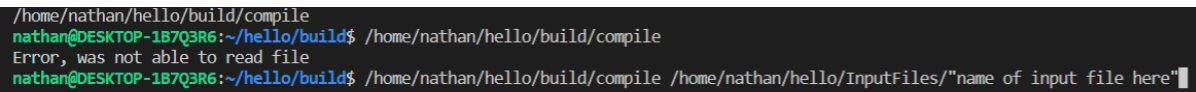
Example

To properly test out the interpreter, a file is included with the assignment containing a number of input files to test out for the user. Said files were even used for this documentation.

Conclusion

This wraps up the report for the TinyLang language. For testing the program was run on an ubuntu server, and to run the program one simply needs to click on the run button in VSC and then insert the file path for the input file the user would want to run.

An example can be seen below ("name of input file would obviously need to be changed"):



```
/home/nathan/hello/build/compile
nathan@DESKTOP-1B7Q3R6:~/hello/build$ /home/nathan/hello/build/compile
Error, was not able to read file
nathan@DESKTOP-1B7Q3R6:~/hello/build$ /home/nathan/hello/build/compile /home/nathan/hello/InputFiles/"name of input file here"
```

Fig.26. Input file being run

The zip file will contain all the required files, and any input files which were used for testing in the sub folder InputFiles, and the CMakeLists which is required to build and run the program.

References

[1] - Git Repo for 'cppast' - <https://github.com/foonathan/cppast/tree/main/src> (15/06/2022)

Appendix A – List of Figures

Fig.1. DFA for Lexer

Fig.2. Transition Table for Lexer

Fig.3. Token Class

Fig.4. Array of Keywords

Fig.5. Code run to test Lexer

Fig.6. “trial_Lexer.txt” output

Fig.7. The Parser Class

Fig.8. The TokenManager class

Fig.9. Parse Method for ASTNodeIdentifier in the AST.h file

Fig.10. Parse Method for ASTNodeIdentifier in the AST.cpp file

Fig.11. Parse Method for ASTNodeIfStatement in the AST.h file

Fig.12. Parse Method for ASTNodeIfStatement in the AST.cpp file

Fig.13 XMLVisitor class in Visitor.h file

Fig.14 XMLVisitor class in Visitor.h file (second part)

Fig.15. An XML visit for a leaf node accept

Fig.16. Aforementioned program which is recursive

Fig.17. XML infinite loop test

Fig.18. XML Generation output

Fig.19. Code in ‘Visitor.h’ file to perform Semantic Analysis

Fig.20. Semantically correct code used

Fig.21. Proof that the code was semantically correct

Fig.22. Non-Semantically correct code

Fig.23. Semantic Analysis not even being attempted

Fig.24. Code for interpreter in the “Visitor.h” class

Fig.25. ValueType is a struct which was created for this visitor

Fig.26. Input file being run