# Compiler Theory and Practice
# Course Assignment

Matthias Bartolo* (0436103L)

*B.Sc. It (Hons) Artificial Intelligence (Second Year)

Study-unit: **Object Oriented Programming**

Code: **CPS2000**

Lecturer: **Dr Sandro Spina**

Video Link:

## Table of Contents

# Implementation

Please note that the required **PixArDis Compiler** was programmed in the **C++ programming language**. Furthermore, the implementation contains the following hierarchy of files:

The following are the **source code files** used:

1. **Lexer.cpp**
   - This file contains the implementation of the Lexer.
2. **Token.cpp**
   - This file contains the implementation of the Token class, which will be used by the Lexer.
3. **Parser.cpp**
   - This file contains the implementation of the Parser.
4. **ASTNodes.cpp**
   - This file contains the respective implementations of the AST class nodes to represent the EBNF structure.
5. **XMLVisitorNodes.cpp**
   - This file contains the implementation of the Visitor Nodes, which perform the XML Pass.
6. **SemanticVisitorNodes.cpp**
   - This file contains the implementation of the Visitor Nodes, which perform the Semantic Pass.
7. **CodeGeneratorVisitorNodes.cpp**
   - This file contains the implementation of the Visitor Nodes, which generate the PixIr code.
8. **SymbolTable.cpp**
   - This file contains the implementation of the Symbol Table, which is utilised in the Semantic and Code Generation Passes.
9. **MainClass.cpp**
   - This file loads the PixArLang code for the compiler to run.

The following are the **header files** used:

1. **ASTNodes.h**
   - This file contains the class definitions of the ASTNodes.cpp file.
2. **HeaderFile.h**
   - This file contains the class definitions of the Lexer.cpp, Token.cpp, Parser.cpp files.
3. **VisitorNodes.h**
   - This file contains the class definitions of the XMLVisitorNodes.cpp, SemanticVisitorNode.cpp and CodeGeneratorVisitorNodes.cpp files.
4. **SymbolTable.h**
   - This file contains the class definitions of the SymbolTable.cpp file.

The following are the **csv table files** used:

1. **IdentifierTable.csv**
   - This file contains the identifier (keyword) Table used in Lexical Analysis.
2. **CAT.csv**
   - This file contains the category Table used in Lexical Analysis.
3. **TokenTable.csv**
   - This file contains the token Table used in Lexical Analysis.
4. **TransTable.csv**
   - This file contains the transition Table used in Lexical Analysis.

Moreover, the compiler utilises the **PixArLang.txt file** which holds the PixArLang code to compile.

```cpp
//Inclusion of relevant header File
#include "HeaderFile.h"

int main() {
    unique_ptr<Parser> parser= make_unique<Parser>();

    string filename="PixArLang.txt";
    //Loading File
    fstream readFilePointer;
    readFilePointer.open("../"+filename, ios::in);
    parser->Compile(readFilePointer);
    parser.reset();

    return 0;
}
```

**Figure 1: Main Class Method**

As can be seen in Figure 1, the PixArLang.txt file is loaded, and an object of the Parser class is created. Furthermore, utilising the created object, the Compile function is called, which essentially tokenises the contents of the text file, performs syntactic analysis, and initiates the XML, Semantic and Code Generation Passes. The Compile function can be seen in Figure 2.

```cpp
void Parser::Compile(fstream &readFilePointer) {
    //Parsing Program
    this->program =ParseProgram(readFilePointer);
    //Calling the Respective Passes
    system("Color 0A");

    cout<<"\n<Parsing Complete>"<<endl;
    system("Color 0F");

    cout<<"\nXML Pass: \n"<<endl;
    system("Color 0B");

    XMLPass();
    system("Color 0F");

    cout<<"\nSemantic Pass: \n"<<endl;
    SemanticPass();
    system("Color 0A");

    cout<<"<No Semantic Errors>"<<endl;
    system("Color 0F");

    cout<<"\nCode Generation Pass: \n"<<endl;
    system("Color 0E");

    CodeGenerationPass();
    system("Color 0F");
}
```

**Figure 2: Compile Function in Parser Class**

4

# Task 1 - Table-driven Lexer

A crucial step in the Compiling process, is the **Lexical Analysis**, which ultimately is responsible for tokenising the program into tokens, whilst identifying erroneous tokens and characters. Implementation of the Lexer class can be seen in Figure 3.

```
class Lexer{
private:
    stack<string> stack;//Stack
    map<string,string> catTable;
    map<string,string> tokenTable;
    map<string,string> identifierTable;
    map<string,map<string,string>> transitionTable;
    void ClearStack();
    void LoadTables();
public:
    shared_ptr<Token> GetNextToken(fstream &readFilePointer);
    Lexer();
    ~Lexer()=default;
};
```

Figure 3: Lexer Implementation

## Deterministic Finite State Automata (DFSA)

The Lexer implementation utilises regular expressions as a transition system, in order to tokenise the input program to valid tokens, which will later be passed on to the Parser. Furthermore, to facilitate this, a deterministic finite state automata (DFSA) was used in order implement such regular expressions. The DFSA utilised can be seen pictorially in Figure 4. In addition, programming such DFSA was facilitated through the use of multiple tables, whereby mappings of the different transitions were modelled as lookups in the tables mentioned above.
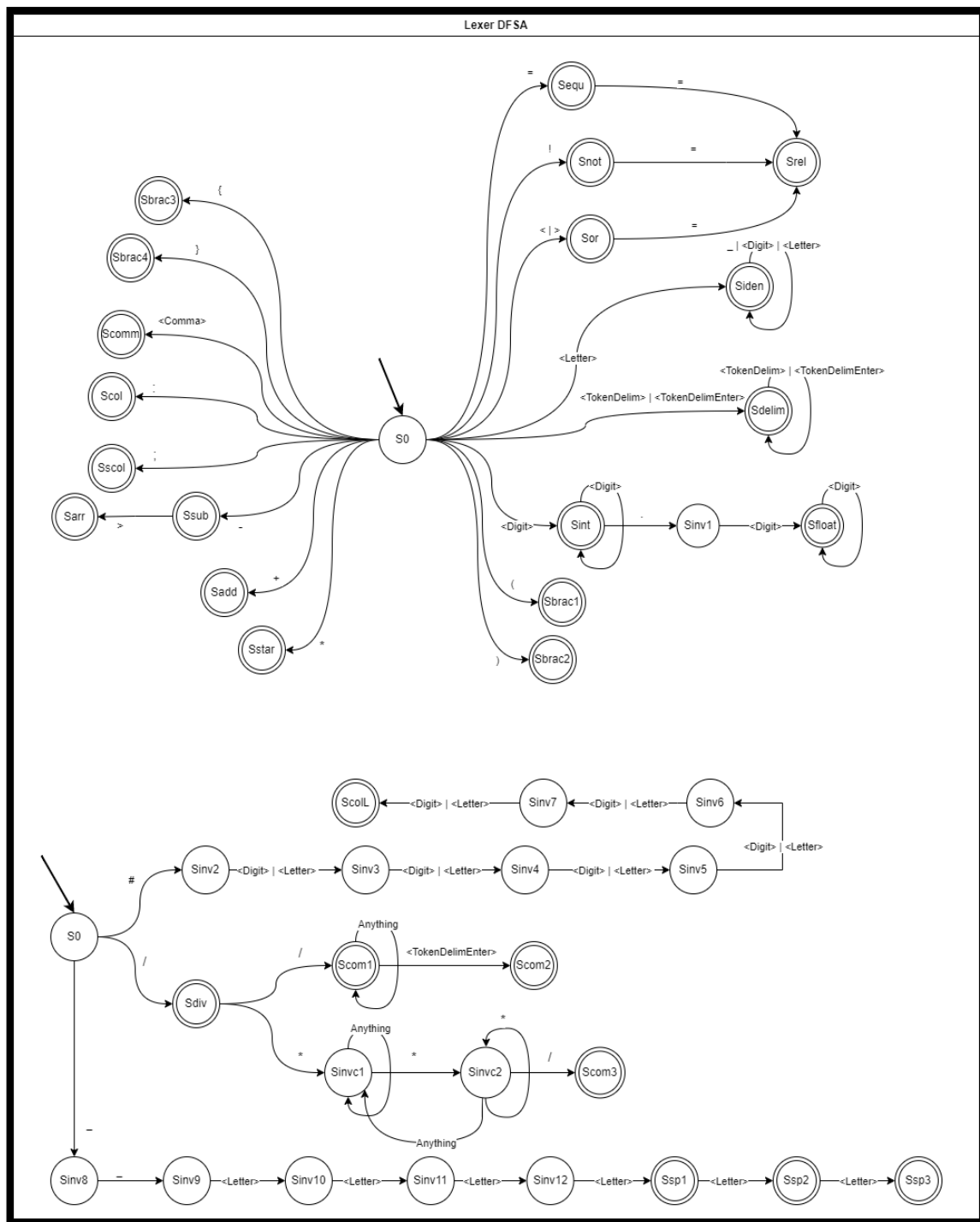
**Figure 4: Lexer DFSA**

## CAT Table

The CAT table is used to map the current character to its token type, for example, the character A would be mapped to a <Letter> token, whilst the character of 9 would be mapped to the <Digit> token. Additionally, these tokens will be utilised as the DFSA's transition labels. This table can be seen in Figure 5.

## Token Table

The Token table is used to map the current final state to its token type, for example, the state Siden, will be mapped to the <Identifier> token. This table can be seen in Figure 6.

## Identifier/Keyword Table

The identifier/Keyword table is used to map special keywords which have been mapped as identifiers. This was done as special keywords are composed of <Letter> tokens, which through the DFSA, are mapped as <Identifier> tokens. An example of such token would be the "let" keyword, which is used specifically for Variable Declarations. This table can be seen in Figure 7.

## Transition Table

The transition table is used to map the transition between two states in the DFSA, i.e., given the current state and the transition label (current token), one would be able to determine the new state. For example, given the current state of S0 and the token of <Comma>, one can easily use such table to lookup the next state in the table, and transition to the new state of Scomm. This table can be seen in Figure 8.

| 1 | float | <Type> |
|---|---|---|
| 2 | int | <Type> |
| 3 | bool | <Type> |
| 4 | colour | <Type> |
| 5 | true | <BooleanLiteral> |
| 6 | false | <BooleanLiteral> |
| 7 | and | <MultiplicativeOp> |
| 8 | or | <AdditiveOp> |
| 9 | not | <not> |
| 10 | return | <return> |
| 11 | if | <if> |
| 12 | else | <else> |
| 13 | for | <for> |
| 14 | let | <let> |
| 15 | while | <while> |
| 16 | fun | <fun> |

**Figure 6: Identifier/Keyword Table**

| 1 | S0 | <Invalid> |
|---|---|---|
| 2 | Siden | <Identifier> |
| 3 | Sdelim | <TokenDelim> |
| 4 | Sint | <IntegerLiteral> |
| 5 | Sinv1 | <Invalid> |
| 6 | Sfloat | <FloatLiteral> |
| 7 | Sbrac1 | <(> |
| 8 | Sbrac2 | <)> |
| 9 | Sbrac3 | <{> |
| 10 | Sbrac4 | <}> |
| 11 | Sstar | <MultiplicativeOp> |
| 12 | Sadd | <AdditiveOp> |
| 13 | Ssub | <AdditiveOp> |
| 14 | Sarr | <->> |
| 15 | Sscol | <;> |
| 16 | Scol | <:> |

**Figure 5: Token Table**

| 1 | A | <Letter> |
|---|---|---|
| 2 | B | <Letter> |
| 3 | C | <Letter> |
| 4 | D | <Letter> |
| 5 | E | <Letter> |
| 6 | F | <Letter> |
| 7 | G | <Letter> |
| 8 | H | <Letter> |
| 9 | I | <Letter> |
| 10 | J | <Letter> |
| 11 | K | <Letter> |
| 12 | L | <Letter> |
| 13 | M | <Letter> |
| 14 | N | <Letter> |
| 15 | O | <Letter> |
| 16 | P | <Letter> |

**Figure 7: CAT Table**

| 1 | | <Letter> | <Digit> | <TokenDelim> | <(> | <)> | <{> | <}> | <*> | <+> | <-> | <>> | <;> | <:> | <Comma> | <=> | <!> | <<> | <#> | </> | <TokenDelimEnter> |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | S0 | Siden | Sint | Sdelim | Sbrac1 | Sbrac2 | Sbrac3 | Sbrac4 | Sstar | Sadd | Ssub | Sor | Sscol | Scol | Scomm | Sequ | Snot | Sor | Sinv2 | Sdiv | Sdelim |
| 3 | Siden | Siden | Siden | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |
| 4 | Sdelim | Serr | Serr | Sdelim | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Sdelim |
| 5 | Sint | Serr | Sint | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |
| 6 | Sinv1 | Serr | Sfloat | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |
| 7 | Sfloat | Serr | Sfloat | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |
| 8 | Sbrac1 | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |
| 9 | Sbrac2 | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |
| 10 | Sbrac3 | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |
| 11 | Sbrac4 | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |
| 12 | Sstar | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |
| 13 | Sadd | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |
| 14 | Ssub | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Sarr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |
| 15 | Sarr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |
| 16 | Sscol | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr | Serr |

**Figure 8: Transition Table**

Impelementation of such tables was done through csv files, in order to make the the process of changing the DFSA quite easy and dynamic, i.e., the program would not require any major changes, in case of changes to the DFSA. Furthermore, such tables were loaded in the std::map data structure in C++, in order to gurantee fast lookup. Moreover, loading such tables can be facilitated through the LoadTables() function, and an example of the loading process for a single table can be seen in Figure 9.

```cpp
//Opening Transition Table File
fstream transitionTableFilePointer;
transitionTableFilePointer.open("../TransTable.csv", ios::in);
//Checking whether file is open
if(transitionTableFilePointer.is_open()){
    string secondaryKeys,data;
    //Retrieving the secondary Keys
    getline(transitionTableFilePointer,secondaryKeys);

    string key;
    //Retrieving the dictionary key
    while(getline(transitionTableFilePointer,key,',')){
        //Removing first , from secondary Keys
        stringstream str2(secondaryKeys);
        string value,secondaryKey;
        getline(transitionTableFilePointer,value);
        stringstream str(value);
        getline(str2,secondaryKey,',');
        //Retrieving the dictionary secondary key and value
        while(getline(str2,secondaryKey,',')) {
            getline(str,value,',');
            //Storing in table dictionary data structure whereby (first index is state, secondary index is the token)
            transitionTable[key][secondaryKey]=value;
        }
    }
}else{
    //Printing respective error message
    cerr<<"\nError: Transition Table not found"<<endl;
    exit(1);
}
//Closing file
transitionTableFilePointer.close();
```

**Figure 9: Loading Process for Transition Table**

## Overview of Lexical Analysis

The main component of the Lexical Analysis can be attributed to the GetNextToken() function, which is responsible for delivering a valid token to the Parser, given a current file stream. Moreover, such function is composed of three components:

1. **Scanning Loop**
2. **Rollback Loop**
3. **Printing Results**

Scanning Loop

This process entails looping until the current state is not an Error State or the end of file is reached. Furthermore, for every iteration in the Scanning Loop, the read file character is appended to the lexeme. Afterwards, the Lexer appends the current state to the Lexer Stack which holds a stack of states and proceeds to look up the current transition label from the cat Table, in order to perform the relevant transition to the new state, through look up from the transition table. In case that the current state is not invalid, the Lexer will proceed to Clear the Stack, as the old valid state is not required any longer. Please note that some special case checks where hardcoded for example the "\n" and "," characters as it was not possible to implement them in the csv file format. This process can be seen in Figure 10.

Rollback Loop

This process entails looping until the current state is Invalid and the Lexer stack is not empty. Moreover, for every iteration in the Rollback Loop, a state is popped from the stack, and the lexeme is truncated. This process aims to extract a valid token with the largest size. Moreover, this process can be seen in Figure 11.

Printing Results

This process entails returning the relevant token, through the state look up in the token table. Furthermore, in this step, the Lexer also checks whether Identifier tokens map to special keywords, through lookup in identifier table. Conclusively, a token data structure is returned, which holds the current token type and the lexeme, in case that the token was not invalid. This process can be seen in Figure 12.

**Note:** A Token class was created in order to act as a data structure to store the relevant string token type and string token attribute. Furthermore, such class was instrumental in such analysis, as to allow the interface between the Lexer and the Parser.

```
//Scanning Loop (Looping until state is not error state, and file pointer has not reached end of file)
while(state!="Serr"&&!readFilePointer.eof()){
    //Reading next Character from the file stream
    char nextChar;
    readFilePointer >> noskipws >> nextChar;
    //ErrorChecking: Checking whether end of file is reached
    if(readFilePointer.eof()){
        break;
    }
    //Appending nextChar to invalidToken and lexeme
    invalidToken+=nextChar;
    lexeme+=nextChar;
    //Checking whether state is not Invalid, through tokenTable look-up, and if so clearing stack
    if(tokenTable[state]!="<Invalid>"){
        ClearStack();
    }
    //Pushing state on the stack, and retrieving tokenType through catTable look-up
    stack.push(state);
    string token;
    token=nextChar;
    string tokenType=catTable[token];

    //Static Checks for Token Delimiters and Commas, as was proven to being difficult through the use of csv file
    //Updating the states respectively through transitionTable look-up, based on the state and tokenType
    if(token ==","){
        state=transitionTable[state]["<Comma>"];
    }
    else if(token ==" "){
        state=transitionTable[state]["<TokenDelim>"];
    }
    else if(token == "\n"){
        state=transitionTable[state]["<TokenDelimEnter>"];
    }
    else{
        state=transitionTable[state][tokenType];
    }
}
```

**Figure 10: Scanning Loop**

```
//RollBack Loop (Looping until state is Invalid and state is not Bad)
while(tokenTable[state]=="<Invalid>"&&state!="Bad"){
    //Popping state from the stack
    state=stack.top();
    stack.pop();
    //Truncating Lexeme
    readFilePointer.putback(lexeme[lexeme.length()-1]);
    lexeme = lexeme.substr(0, lexeme.length() - 1);
}
```

**Figure 12: Rollback Loop**

```
//Report Result
//Checking whether state is not Bad, and state is not Invalid
if(state!="Bad"&&tokenTable[state]!="<Invalid>"){
    string tokenName=tokenTable[state];
    //Checking for specific keywords, in the case the token is an identifier
    if(tokenTable[state]=="<Identifier>"){
        //Checking whether lexeme is one of the keys in the identifierTable
        if(identifierTable.find(lexeme)!=identifierTable.end()){
            tokenName=identifierTable[lexeme];
        }
    }
    //Checking whether tokenNam is empty, if so setting tokenName to Invalid
    if(tokenName.empty()){
        tokenName="<Invalid>";
    }
    //Returning Token
    shared_ptr<Token> token = make_shared<Token>(tokenName,lexeme);
    return token;
}
else{
    //Printing respective error message
    cerr<<"\nError: Invalid Token: "<<invalidToken<<endl;
    exit(3);
}
```

**Figure 11: Printing Results**

# Task 2 - Hand-crafted LL(k) parser

**Syntax Analysis** can also be considered as a key important step in the process of constructing a compiler. Moreover, this analysis aims to interpret the sequence of produced tokens from the Lexer, and report any errors, in case that the tokens don't follow a correct order, i.e., the sequence of tokens, does not satisfy the PixArLang programming language. Additionally, in this step the Parser derives a Parse Tree, and constructs a **Syntax Tree**, which will be used later on through the respective passes.

## Lookahead Tokens

The implementation of the required LL(k) parser, was facilitated through the utilisation of two lookahead tokens, named **lookaheadtoken1** and **lookaheadtoken2** respectively. Furthermore, updating such tokens, was facilitated through the **GetNextToken()** function in the Parser. In addition, this function works by first updating lookaheadtoken1 with the contents of lookaheadtoken2. Afterwards, the function proceeds to check whether the end of file is reached, and if so, setting the EOFFlag, in order to halt execution later on. Next, the function proceeds to retrieve the next token from the Lexer and stores this token in the lookaheadtoken2 token. Additionally, such token is fed to **CheckValidToken()** function which is responsible to display the relevant error in case the received token has an invalid format. What is more, in case the current token is a white space token (has a token type of **TokenDelim** or **TokenDelimEnter**, i.e., is a space of enter), or a comment token, the Parser ignores such tokens, and continues to retrieve tokens from the Lexer, until the received token does not constitute to the mentioned token cases. This function can be seen in Figure 13.

Additionally, it was decided that the special language tokens, i.e., those which start with "__", as well as Colour Literals would be verified for correct structure in the Parser. In continuation, depending on the size of the "__" special language token, each special token was assigned to a different special case type. For example, since "__pixelr" and "__height" have a character length of 8, they were assigned to SpecialCase3, whilst "__read" has a character length of 6, it is assigned to SpecialCase1. Validity of such tokens was also checked through the CheckValidToken() function. Moreover, such tokens were checked through the use of case matching with specific strings and std::vectors depending on the different special case types, in order to maintain a dynamic approach and enable easy modifications. Furthermore, such function is also responsible for the respective error presentation, in case of Invalid tokens, and Invalid comments. This function can be seen in Figure 14.

```
//Method which communicates with the Lexer, and updates the two lookahead tokens
void Parser::GetNextToken() {
    //Setting lookaheadtoken1 to lookaheadtoken2
    lookaheadToken1=lookaheadToken2;
    //Checking whether end of file is reached, if so, setting the relevant flag
    if(!file.eof()){
        //Retrieving token from lexer, and storing in lookaheadtoken2, and checking when token is valid
        lookaheadToken2 = lexer->GetNextToken(file);
        CheckValidToken(lookaheadToken2);
        //Continue to loop is token is one of the special cases, which will be ignored during parsing
        while((lookaheadToken2->GetTokenName()=="<TokenDelimEnter>"||lookaheadToken2->GetTokenName()=="<TokenDelim>"||lookaheadToken2->GetTokenName()=="<Comment>"))
            lookaheadToken2 = lexer->GetNextToken(file);
            CheckValidToken(lookaheadToken2);
        }
    }
    else{
        EOFFlag=true;
    }
    if(lookaheadToken2->GetTokenName()=="<Empty>"){
        EOFFlag=true;
    }
}
```

**Figure 13: Parser Get Next Token function.**

```
//Method which Checks whether Token is Valid, given a token
void Parser::CheckValidToken(const shared_ptr<Token>& token) {
    //Checking Valid Colour Literal (Hex Number)
    if(token->GetTokenName()=="<ColourLiteral>") {
        string colourLiteral=token->GetTokenAttribute();
        for(int i=1;i<colourLiteral.length();i++){
            bool validChar=validHexCharacters.find(colourLiteral[i])!= string::npos;
            if(!validChar){
                token->SetTokenName("<Invalid>");
            }
        }
    }
    //Checking Special Case 1 (tokens which start with __)
    if(token->GetTokenName()=="<SpecialCase1>") {
        if(specialCase1[token->GetTokenAttribute()].empty()){
            token->SetTokenName("<Invalid>");
        }
        else{
            token->SetTokenName(specialCase1[token->GetTokenAttribute()]);
        }
    }
    //Checking Special Case 2 (tokens which start with __)
    if(token->GetTokenName()=="<SpecialCase2>") {
        if(specialCase2[token->GetTokenAttribute()].empty()){
            token->SetTokenName("<Invalid>");
        }
        else{
            token->SetTokenName(specialCase2[token->GetTokenAttribute()]);
        }
    }
    //Checking Special Case 3 (tokens which start with __)
    if(token->GetTokenName()=="<SpecialCase3>") {
        if(specialCase3[token->GetTokenAttribute()].empty()){
            token->SetTokenName("<Invalid>");
        }
        else{
            token->SetTokenName(specialCase3[token->GetTokenAttribute()]);
        }
    }
    //Relevant Error Printing if token is Invalid
    if(token->GetTokenName()=="<Invalid>") {//Invalid Token
        cerr<<"\nError: Invalid Token: \n"<<token->GetTokenAttribute()<<endl;
        exit(3);
    }
    if(token->GetTokenName()=="<InvalidComment>") {//Invalid Comment
        cerr<<"\nError: Invalid Comment: \n"<<token->GetTokenAttribute()<<endl;
        exit(3);
```

**Figure 14: Parser Check Valid Token function.**

## Changes to EBNF

Additionally, please also note that the following modifications were made to the assignment's EBNF, in order to include the Parsing of the Clear Statement, which was not specifically mentioned in the brief, however hinted in Task 5, as one of the PixIR Commands.

**EBNF Modifications:**

⟨Statement⟩  ::=  ⟨VariableDecl⟩  ';'
                  | ⟨Assignment⟩  ';'
                  | ⟨PrintStatement⟩  ';'
                  | ⟨DelayStatement⟩  ';'
                  | ⟨PixelStatement⟩  ';'
                  | **⟨ClearStatement⟩**  ';'
                  | ⟨IfStatement⟩
                  | ⟨ForStatement⟩
                  | ⟨WhileStatement⟩
                  | ⟨RtrnStatement⟩  ';'
                  | ⟨FunctionDecl⟩
                  | ⟨Block⟩

**⟨ClearStatement⟩  ::= '__clear' <ColourLiteral>**

## Parse Functions

Furthermore, syntax analysis in the Parser was implemented through the implementation of various Parse functions, which attempt to match the valid case for that specific EBNF case. Furthermore, the lookahead tokens mentioned above, are used as a tool to differentiate between the different Parse functions to call. Moreover, these tokens also enable the verification of the correct sequence of tokens which abide by the rules of the PixArLang programming language. The numerous Parsing functions can be seen in Figure 15. Moreover, an example of the inner workings of one of these functions can be seen in Figure 16.

```cpp
shared_ptr<ASTProgram> ParseProgram(fstream &readFilePointer);
shared_ptr<ASTStatement> ParseStatement();
shared_ptr<ASTAssignment> ParseAssignment();
shared_ptr<ASTIdentifier> ParseIdentifier();
shared_ptr<ASTExpr> ParseExpression();
shared_ptr<ASTExpr> ParseFactor();
shared_ptr<ASTLiteral> ParseLiteral();
shared_ptr<ASTBlock> ParseBlock();
shared_ptr<ASTExpr> ParseSimpleExpression();
shared_ptr<ASTExpr> ParseTerm();
shared_ptr<ASTActualParams> ParseActualParams();
shared_ptr<ASTFunctionCall> ParseFunctionCall();
shared_ptr<ASTVariableDecl> ParseVariableDecl();
shared_ptr<ASTExpr> ParsePadRandi();
shared_ptr<ASTExpr> ParseSubExpression();
shared_ptr<ASTExpr> ParseUnary();
shared_ptr<ASTPadRead> ParsePadRead();
shared_ptr<ASTPrintStatement> ParsePrintStatement();
shared_ptr<ASTClearStatement> ParseClearStatement();
shared_ptr<ASTDelayStatement> ParseDelayStatement();
shared_ptr<ASTRtrnStatement> ParseReturnStatement();
shared_ptr<ASTPixelStatement> ParsePixelStatement();
shared_ptr<ASTIfStatement> ParseIfStatement();
shared_ptr<ASTForStatement> ParseForStatement();
shared_ptr<ASTWhileStatement> ParseWhileStatement();
shared_ptr<ASTFormalParam> ParseFormalParam();
shared_ptr<ASTFormalParams> ParseFormalParams();
shared_ptr<ASTFunctionDecl> ParseFunctionDecl();
```

**Figure 15: Parsing Functions**

```cpp
//Method which Parses a Return Statement, and returns pointer of ASTRtrnStatement
shared_ptr<ASTRtrnStatement> Parser::ParseReturnStatement() {
    /*Matching Case: ⟨RtrnStatement⟩ ::= 'return' ⟨Expr ⟩ */
    shared_ptr<ASTExpr> expression;
    //Error Checking when return is not matched
    if(lookaheadToken1->GetTokenName()!="<return>"){
        cerr<<"\nReturn Statement Error: Expected return, received:"<<lookaheadToken1->GetTokenAttribute()<<endl;
        exit(4);
    }
    GetNextToken();
    expression=ParseExpression();
    return make_shared<ASTRtrnStatement>(expression);
}
```

**Figure 16: Parse Return Statement function**

## Syntax Tree Construction

As previously mentioned, the Parser is also responsible for the creation of the Syntax Tree. This was achieved, through the creation of various types of **Abstract Syntax Tree (AST) Nodes**, thus forming a hierarchical structure, which was used in conjunction with the Parse function mentioned above. Moreover, this can also be seen in Figure 16, whereby the ParseReturnStatement() function returns a pointer to the "ASTRtrnStatement" node. Additionally, creation of such Hierarchical structure utilised the amalgamation of different Object-Oriented concepts, such as Polymorphism and Inheritance. The aforementioned structure can be seen in its entirety in Figure 19.

In continuation, each AST Node was designed to have different properties, for example, the ASTWhileStatement Node contains references to ASTBlock and ASTExpr Nodes. On the other hand, the ASTFormalParams Node only contains references to a vector of ASTFormalParam Nodes. This difference can be seen in Figure 17 and 18.

```cpp
class ASTWhileStatement: public ASTStatement{
public:
    ASTWhileStatement()=default;
    ASTWhileStatement( shared_ptr<ASTExpr> expression, shared_ptr<ASTBlock> block);
    void accept(VisitorNode * visitor) override;
    shared_ptr<ASTBlock> block;
    shared_ptr<ASTExpr> expression;
    ~ASTWhileStatement();
};
```

**Figure 17: ASTWhileStatement Node**

```cpp
class ASTFormalParams: public ASTStatement{
public:
    ASTFormalParams()=default;
    explicit ASTFormalParams( vector<shared_ptr<ASTFormalParam>> formalParams);
    void accept(VisitorNode * visitor) override;
    vector<shared_ptr<ASTFormalParam>> formalParams;
    ~ASTFormalParams();
};
```

**Figure 18: ASTFormalParams Node**
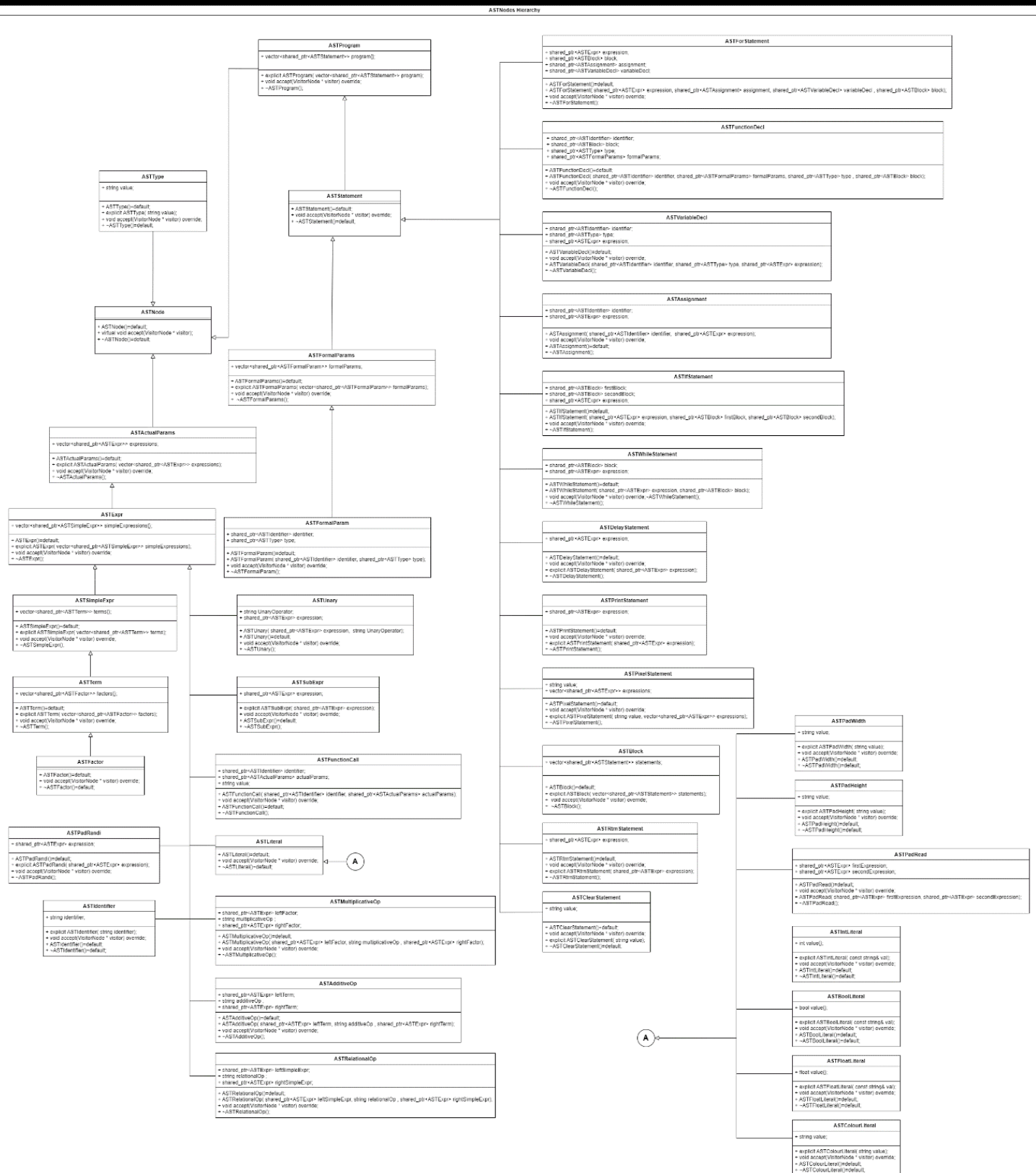
**Figure 19: Abstract Syntax Tree Node Hierarchy**

# Visitor Node Design Pattern

Following the construction of the Abstract Syntax Tree in the Syntax Analysis phase, one can easily traverse the tree in order to perform different passes, such as the **XML Generation Pass**. Furthermore, traversing such tree, required the creation of a **Visitor Node Design Pattern**, which allowed different visitor nodes to read the contents and perform the necessary computation depending on the type of the current AST node in the AST tree.

Moreover, the latter was achieved through the construction of the **accept** method in all of the different AST Node data structures, whereby such method, would be given a Visitor Node, and depending on the type of the Visitor Node, the accept method would invoke the appropriate **visit** method for that particular Visitor Node. Additionally, the following implementation was constructed in such a way, as to facilitate simple expansion, and to employ proper coding design principles, utilising; **Inheritance**, **Method Overriding** and **Method Overloading**. In this case Inheritance is used to allow different Visitor Nodes, to be able to visit the different AST Nodes. In addition, Method Overriding is being used, to ensure that all the different Visitor Nodes would perform different computation with respect to the current AST node. Moreover, Method Overloading is used to allow the Visitor Nodes to be able to visit all the different AST Nodes, whilst only needing to invoke the same visit method. Additionally, some of the AST Node accept methods, can be seen in Figure 20. Furthermore, Figure 21 illustrates the Visitor Node class, which was inherited by other types of Visitor Nodes.

```
//accept method for the relevant AST Nodes
void ASTNode::accept(VisitorNode * visitor) {
    visitor->visit(this);
}
void ASTProgram::accept(VisitorNode * visitor) {
    visitor->visit(this);
}
void ASTStatement::accept(VisitorNode * visitor) {
    visitor->visit(this);
}
void ASTType::accept(VisitorNode * visitor) {
    visitor->visit(this);
}
```

Figure 20: AST Node accept methods.

```cpp
//Visitor Node class with its relevant variables and methods
class  VisitorNode{
public:
    int tabCounter=0;
    VisitorNode()=default;
    virtual void visit( ASTNode * pointer)=0;
    virtual void visit( ASTProgram * pointer)=0;
    virtual void visit( ASTStatement *pointer)=0;
    virtual void visit( ASTPrintStatement *pointer)=0;
    virtual void visit( ASTDelayStatement *pointer)=0;
    virtual void visit( ASTForStatement *pointer)=0;
    virtual void visit( ASTWhileStatement *pointer)=0;
    virtual void visit( ASTIfStatement *pointer)=0;
    virtual void visit( ASTVariableDecl *pointer)=0;
    virtual void visit( ASTAssignment *pointer)=0;
    virtual void visit( ASTBlock *pointer)=0;
    virtual void visit( ASTFunctionDecl *pointer)=0;
    virtual void visit( ASTFunctionCall *pointer)=0;
    virtual void visit( ASTFormalParams *pointer)=0;
    virtual void visit( ASTFormalParam *pointer)=0;
    virtual void visit( ASTType *pointer)=0;
    virtual void visit( ASTIdentifier *pointer)=0;
    virtual void visit( ASTMultiplicativeOp *pointer)=0;
    virtual void visit( ASTAdditiveOp *pointer)=0;
    virtual void visit( ASTRelationalOp *pointer)=0;
    virtual void visit( ASTRtrnStatement *pointer)=0;
    virtual void visit( ASTFactor *pointer)=0;
```

Figure 21: Visitor Node class

19

As previously mentioned, the overloaded visit methods for the different Visitor Node Designs, each provide different functionality. Such difference can be seen in Figures 22 and 23, whereby calling the same method given an ASTProgram Node pointer, would render the XML Visitor Node to display the relevant XML tags, whilst the Semantic Visitor Node to perform the necessary Semantic checks.

```cpp
void XMLVisitorNode::visit(ASTProgram * pointer) {
    printIndent();
    cout<<"<Program>"<<endl;
    tabCounter++;
    for(auto iter = pointer->program.begin(); iter < pointer->program.end(); iter++)
    {
        ((*iter))->accept(this);
    }
    tabCounter--;
    printIndent();
    cout<<"</Program>"<<endl;
}
```

**Figure 23: XML Visitor Node visit function for ASTProgram node.**

```cpp
void SemanticVisitorNode::visit(ASTProgram * pointer) {
    //Creating a new Scope and pushing it onto the symbol Table
    Scope initialScope;
    symbolTable->push(initialScope);
    string currentFunctionName=currentStoredFunctionName;
    //Iterating through all the statements in the program, and resetting the currentStoredFunctionName, for every iteration
    for(auto iter = pointer->program.begin(); iter < pointer->program.end(); iter++)
    {
        currentStoredFunctionName=currentFunctionName;
        ((*iter))->accept(this);
    }
    //Popping initialScope from the symbol Table
    symbolTable->pop();
}
```

**Figure 22: Semantic Visitor Node visit function for ASTProgram node.**

# Task 3 - AST XML Generation Pass

In this pass the Compiler proceeds to utilise the constructed AST Node Hierarchical structure to generate XML code which matches the given PixArLang program code, to allow easy visualisation of the passed code. Furthermore, such pass is first initiated from the Parser as can be seen in Figure 24, through the XMLPass method, whereby a new XMLVisitorNode is created and fed to the accept method for the generated ASTProgram node.

```cpp
//Method which Initiates an XML Pass
void Parser::XMLPass() {
    //Creating new XML Visitor Node, and initiating accept method from node
    auto* node= new XMLVisitorNode();
    program->accept(node);
}
```

**Figure 24: XML Pass method in the Parser**

Additionally, the XMLVisitorNode class can also be seen in Figure 25.

```cpp
//XML Visitor Node class with its relevant variables and methods
class XMLVisitorNode : public VisitorNode{
public:
    XMLVisitorNode()=default;
    void printIndent();
    void visit( ASTNode *pointer)override;
    void visit( ASTProgram *pointer)override;
    void visit( ASTStatement *pointer)override;
    void visit( ASTPrintStatement *pointer)override;
    void visit( ASTDelayStatement *pointer)override;
    void visit( ASTForStatement *pointer)override;
    void visit( ASTWhileStatement *pointer)override;
    void visit( ASTIfStatement *pointer)override;
    void visit( ASTVariableDecl *pointer)override;
    void visit( ASTAssignment *pointer)override;
    void visit( ASTBlock *pointer)override;
    void visit( ASTFunctionDecl *pointer)override;
    void visit( ASTFunctionCall *pointer)override;
    void visit( ASTFormalParams *pointer)override;
    void visit( ASTFormalParam *pointer)override;
    void visit( ASTType *pointer)override;
    void visit( ASTIdentifier *pointer)override;
    void visit( ASTMultiplicativeOp *pointer)override;
    void visit( ASTAdditiveOp *pointer)override;
    void visit( ASTRelationalOp *pointer)override;
    void visit( ASTRtrnStatement *pointer)override;
    void visit( ASTFactor *pointer)override;
```

**Figure 25: XMLVisitorNode class**

## Printing Indented Tags

The printed XML code needed to be indented in order to provide easier readability. This was achieved through the creation of a variable in the XMLVisitorNode class, which was used as a tabCounter, and would occasionally be incremented or decrementing accordingly, when printing the different XML tags. Additionally, a method was also created which prints the number of tabs in accordance with the tabCounter, which was invoked before the printed XML tag. This method is illustrated in Figure 26. An example showing one of the XMLVisitorNode accept methods, can be seen in Figure 23, showing the full use of the tabCounter, and printIndent method.

```cpp
//Method which prints relevant indent based on tabCounter
void XMLVisitorNode::printIndent() {
    for(int v=0;v<tabCounter;v++){
        cout<<"  ";
    }
}
```

**Figure 26: Print Indent Method**

Figure 27 illustrates the output given from the XML Pass given the following input:

"let x:int = (7*7) +7;".

```xml
XML Pass:

<Program>
  <Decl>
    <Var Type= "int">
      <Id>x</Id>
    </Var>
    <BinExprNode Op="+">
      <SubExpr>
        <BinExprNode Op="*">
          <IntConst>7</IntConst>
          <IntConst>7</IntConst>
        </BinExprNode>
      </SubExpr>
      <IntConst>7</IntConst>
    </BinExprNode>
  </Decl>
</Program>
```

**Figure 27: XML Pass Output**

# Symbol Table

A Symbol table is a data structure which is utilised by compilers, which stores the relevant information pertaining to special symbols such as variables and functions. Moreover, the symbol table has the structure of a stack of scopes, whereby each scope represents the nested block in the program. Additionally, such table will be extensively used in the Semantic and Code Generation Passes. Figure 28 illustrates the creation of such table data structure.

```cpp
//Creating Struct Scope, which will be used by Symbol Table
struct Scope{
    map<string,map<string,string>> scope;
};


//Symbol Table class with its relevant variables and methods
class SymbolTable{
public:
    SymbolTable()=default;
    vector<Scope> scopeStack;
    void push(const Scope& scope);
    Scope pop();
    bool CheckIdentifierExists(const string& identifier, bool functionFlag);
    string ReturnIdentifierType(const string& identifier);
    string ReturnFunctionParameters(const string& identifier);
    string ReturnIdentifierAddress(const string& identifier);
    ~SymbolTable()=default;
};
```

**Figure 28: Symbol Table class and Scope data structure**

As previously mentioned, the implementation of the Symbol table required the implementation of a stack of scopes, this was achieved through the creation of the **Scope** data structure, which stored an std:map of strings. Additionally, the Symbol table class stores a std::vector of the Scope data structure which acts as a stack of scopes, with the added functionality of allowing direct access to all the scopes. In addition, the Symbol table class also contains the relevant **Push** and **Pop** methods which mimic the stack operations to add or remove scopes from the stack of scopes data structure in such class.

## Symbol table functions

The Symbol table class also contains the following methods, which will be used later on to retrieve the relevant information, with respect to the passed symbol:

**CheckIdentifierExists:**

This method takes a string identifier, and a flag which determines whether the passed identifier is a function or not. Furthermore, such method then proceed to iterate through the stack of scopes backwards, in the order of the last pushed scope, and checks whether the passed identifier is inside the current scope. If the identifier is not a function and is found, then the function returns true. If the identifier is a function, then the function proceeds to check whether the current identifier has any Formal parameters. Consequently, if the latter does apply, then true is returned, else false is returned as the identifier is being invoked as a function whilst, not being stored as a function. In case that no identifier is found, the function returns false. Figure 29 illustrates the aforementioned function, and such function is used extensively in the Semantic Pass.

```cpp
//Method which given an Identifier and a FunctionFlag (to determine whether identifier is a function), checks whether Identifier exists
bool SymbolTable::CheckIdentifierExists(const string& identifier, bool functionFlag) {
    //Iterating through all the Scopes in the scopeStack, in the order of the last pushed scope
    auto iter = scopeStack.end();
    for(iter--; iter >= scopeStack.begin(); iter--){
        //If Identifier is found, returning true
        if((*iter).scope.find(identifier) != (*iter).scope.end()&&!functionFlag){
            return true;
        }
        //Checking whether identifier is a function or not, if it is checking whether it has FormalParam and returning true
        //If not returning False, as identifier is not a function, but FunctionFlag is set to true
        if((*iter).scope.find(identifier) != (*iter).scope.end()&&functionFlag){
            if((*iter).scope[identifier].find("FormalParam")!= (*iter).scope[identifier].end()){
                return true;
            }
            else{
                return false;
            }
        }
    }
    //Returning false, as Identifier was not found
    return false;
}
```

**Figure 29: CheckIdentifierExists function**

 **ReturnIdentifierTypes:**

This method takes a string identifier as a parameter and proceeds to iterate through the stack of scopes backwards, in the order of the last pushed scope, checking whether the passed identifier is inside the current scope. If the identifier exists in the current scope, the function proceeds to return the identifier's type, if the identifier does not exist in the stack of scopes, then "No Type" is returned. Figure 30 illustrates the aforementioned function, and such function is used extensively in the Semantic Pass.

```cpp
//Method which given an Identifier, returns the Identifier Type
string SymbolTable::ReturnIdentifierType(const string &identifier) {
    //Iterating through all the Scopes in the scopeStack, in the order of the last pushed scope
    auto iter= scopeStack.end();
    for(iter--; iter >= scopeStack.begin(); iter--){
        //If Identifier is found, returning its Type
        if((*iter).scope.find(identifier) != (*iter).scope.end()){
            return (*iter).scope[identifier]["type"];
        }
    }
    //Returning "No Type", as Identifier was not found
    return "No Type";
}
```

**Figure 30: ReturnIdentifierType function**

**ReturnFunctionParameters:**

This method works similarly to previous method function, however instead of returning the type, the function parameters (FormalParam) are returned. Figure 31 illustrates the aforementioned function, and such function is used extensively in the Semantic Pass.

```cpp
//Method which given an Identifier, returns the Identifier Formal Parameters
string SymbolTable::ReturnFunctionParameters(const string &identifier) {
    auto iter= scopeStack.end();
    for(iter--; iter >= scopeStack.begin(); iter--){
        //If Identifier is found, returning its Formal Parameters
        if((*iter).scope.find(identifier) != (*iter).scope.end()){
            return (*iter).scope[identifier]["FormalParam"];
        }
    }
    //Returning "No Parameters", as Identifier was not found
    return "No Parameters";
}
```

**Figure 31: ReturnFunctionParameters**

25

**ReturnIdentifierAddress:**

This method works similarly to previously method functions, however instead of returning the type or function parameters, the function returns the Identifier Address. Consequently, at each iteration through the stack of scopes, the frame Counter is being incremented, and is then used to recalculate the memory address before returning it, in case that the identifier is found. This was done as to accommodate the PixIR language specification, whereby creation of new frames, would change the access location to variables stored in the old frames. Figure 32 illustrates the aforementioned function, and such function is used extensively in the Code Generation Pass.

```cpp
//Method which given an Identifier, returns the Identifier Address
string SymbolTable::ReturnIdentifierAddress(const string &identifier) {
    int frameCounter=0;
    auto iter= scopeStack.end();
    for(iter--; iter >= scopeStack.begin(); iter--){
        //If Identifier is found, returning its Address
        if((*iter).scope.find(identifier) != (*iter).scope.end()){
            //Parsing through address to retrieve the scopeIndex, and frame Index
            string address=(*iter).scope[identifier]["Address"];
            string delimiter=":";
            string scopeIndex=address.substr(1, address.find(delimiter)-1);
            int frameIndex=stoi(address.substr(address.find(delimiter) + 1,(address.length() - (address.find(delimiter) + 1))))+frameCounter;
            //Returning Instruction with updated frame Index
            return "push ["+scopeIndex+":"+ to_string(frameIndex)+"]\n";
        }
        //Incrementing frame Counter
        frameCounter++;
    }
    //Returning empty string
    return "";
}
```

**Figure 32: ReturnIdentifierAddress**

# Task 4 - Semantic Analysis Pass

In this pass the Compiler proceeds to utilise the constructed AST Node Hierarchical structure to perform a Semantic Pass on the given PixArLang program code, in order to ensure that the passed code is semantically correct (makes sense). Additionally, this pass relied on the use of the Symbol table, in order to determine whether identifiers or functions existed or not. Furthermore, such pass is first initiated from the Parser as can be seen in Figure 33, through the SemanticPass method, whereby a new SemanticVisitorNode is created and fed to the accept method for the generated ASTProgram node.

```cpp
//Method which Initiates a Semantic Pass
void Parser::SemanticPass() {
    //Creating new Semantic Visitor Node, and initiating accept method from node
    auto* node= new SemanticVisitorNode();
    program->accept(node);

}
```

<p align="center"><strong>Figure 33: Semantic Pass method in Parser</strong></p>

Additionally, the SemanticVisitorNode class can also be seen in Figure 34.

```cpp
//Semantic Visitor Node class with its relevant variables and methods
class SemanticVisitorNode: public VisitorNode{
public:
    unique_ptr<SymbolTable> symbolTable= make_unique<SymbolTable>();
    //Variable to store the current Type, to determine the current expression type
    string currentStoredType;
    //Variable to store the current Function Name, to determine the current function being processed
    string currentStoredFunctionName;
    //Variable will act as a flag, and fire when there is a division by zero
    bool zeroFlag=false;
    //Variable will act as a flag, and fire when a function does not have a return statement
    bool returnFlag=false;
    void RemoveFunctionParameters(const shared_ptr<ASTFormalParams>& pointer) const;
    SemanticVisitorNode()=default;
    void visit( ASTNode *pointer)override;
    void visit( ASTProgram *pointer)override;
    void visit( ASTStatement *pointer)override;
    void visit( ASTPrintStatement *pointer)override;
    void visit( ASTDelayStatement *pointer)override;
    void visit( ASTForStatement *pointer)override;
    void visit( ASTWhileStatement *pointer)override;
    void visit( ASTIfStatement *pointer)override;
    void visit( ASTVariableDecl *pointer)override;
    void visit( ASTAssignment *pointer)override;
    void visit( ASTBlock *pointer)override;
    void visit( ASTFunctionDecl *pointer)override;
    void visit( ASTFunctionCall *pointer)override;
    void visit( ASTFormalParams *pointer)override;
    void visit( ASTFormalParam *pointer)override;
    void visit( ASTType *pointer)override;
    void visit( ASTIdentifier *pointer)override;
    void visit( ASTMultiplicativeOp *pointer)override;
    void visit( ASTAdditiveOp *pointer)override;
    void visit( ASTRelationalOp *pointer)override;
    void visit( ASTRtrnStatement *pointer)override;
    void visit( ASTFactor *pointer)override;
```

<p align="center"><strong>Figure 34: SemanticVisitorNode class</strong></p>

## Types of Errors

Aside from Semantic Errors, the constructed PixArDis Compiler provides a proper interface for different errors. The following table portrays the different Error codes which can aid PixArDis developers and clients to identify the correct type of error:

| Exit Code | Interpretation | Occurrence |
|---|---|---|
| 0 | No Errors | N/A |
| 1 | Missing File (csv file) | Lexical Analysis |
| 2 | Missing File (txt file) | Lexical Analysis |
| 3 | Invalid Token | Syntax Analysis/Lexical Analysis |
| 4 | Syntax Analysis Error (EBNF Case not matched) | Syntax Analysis |
| 5 | Semantic Error | Semantic Analysis |
| 6 | Type Mismatch Error | Semantic Analysis |
| 7 | Semantic Error (Division by 0) | Semantic Analysis |

## Semantic Pass Errors

The following are the cases which constitute to a **Semantic Error**:

1. Cannot create identifier which already exists.
2. Program is not expected to have a return.
3. Cannot assign a value to an identifier which does not exist.
4. Cannot have nested function declarations.
5. Function declaration has not return type.
6. Cannot call a function which does not exist.
7. Function call has missing parameters.
8. Function call has unwanted parameters.
9. Cannot divide by zero.

The following are the cases which constitute to a **Type Mismatch Error**:

1. Identifier assigned a conflicting type, which does not match its declaration.
2. Function returns a conflicting type, which does not match its return type declaration.
3. Delay statement expected to have integer type parameters.
4. Pixel statement expected to have integer type parameters, and last parameter to be of type colour.
5. Cannot apply Multiplicative/Additive/Relational operators between conflicting types of left and right expressions.
6. PadRead expected to have integer type parameters.
7. Unary operator expected to have "not" before Boolean expression, and "-" before integer and float expressions.
8. Function call assigned a conflicting type, which does not match its function declaration.

In continuation, the Compiler is responsible to display the relevant message, in case of encountering one of the previously aforementioned Semantic and Type Mismatch Error cases. Illustration of such error checking method can be seen in Figure 35, through one of the SemanticVisitorNode visit method.

```cpp
void SemanticVisitorNode::visit(ASTVariableDecl * pointer) {
    //Calling the accept method on the expression
    pointer->expression->accept(this);
    //Retrieving and storing the currentStoredType
    string storedType=currentStoredType;
    //Retrieving access to last scope pushed in the symbol table
    auto iter = symbolTable->scopeStack.end();iter--;
    string identifier=pointer->identifier->identifier;
    string type=pointer->type->value;
    //Calling CheckIdentifierExists function, to check whether identifier exists
    bool exists=symbolTable->CheckIdentifierExists(identifier,false);
    //Displaying Error, since Identifier exists
    if(exists){
        cerr<<"\nSemantic Error: Identifier already exists, cannot create Identifier with name: \""<<identifier<<"\""<<endl;
        exit(5);
    }
    //Checking for Type Mismatch Error
    if(type!=storedType){
        cerr<<"\nType Mismatch Error: Identifier "<<identifier<<" of type \""<<type<<"\" cannot be assigned \""<<storedType<<"\" type"<<endl;
        exit(6);
    }
    //Saving identifier in the last scope pushed in the symbol table with its type
    (*iter).scope[identifier]["type"]=type;
}
```

**Figure 35: Semantic Visitor Node visit function for ASTVariableDecl node.**

Figure 36 illustrates the relevant print, in case no Semantic or Type Mismatch Errors were encountered.



**Figure 36: No Semantic Errors print**

# Task 5 - PixIR Code Generation Pass

In this pass the Compiler proceeds to utilise the constructed AST Node Hierarchical structure to generate the relevant PixIR code, which will be later fed to the Virtual Machine to be run. Additionally, this pass relied on the use of the Symbol table, in retrieve the memory address of a given identifier. Furthermore, such pass is first initiated from the Parser as can be seen in Figure 37, through the CodeGenerationPass method, whereby a new CodeGeneratorVisitorNode is created and fed to the accept method for the generated ASTProgram node.

```
//Method which Initiates a Code Generation Pass
void Parser::CodeGenerationPass() {
    //Creating new Code Generator Visitor Node, and initiating accept method from node
    auto* node= new CodeGeneratorVisitorNode();
    program->accept(node);

}
```

**Figure 37: Code Generation Pass method in Parser**

Additionally, the CodeGeneratorVisitorNode class can also be seen in Figure 38.

```
//Code Generator Visitor Node class with its relevant variables and methods
class CodeGeneratorVisitorNode: public VisitorNode{
public:
    unique_ptr<SymbolTable> symbolTable= make_unique<SymbolTable>();
    //Vector which will store the order of th created functions, and will be used to hold the keys of the printList
    vector<string> functionNames;
    //Map which will act as a buffer, which will store the code to be printed
    map<string,string> printList;
    //Variable will act as a counter for the current Frame Index
    int frameIndex=0;
    //Variable will act as a counter for nested Conditional Statements i.e., if/for and while statements
    int conditionalCounter=0;
    //Variable will act as a flag, and fire when a - unary operator is found
    bool negativeFlag=false;
    //Variable will act as a flag, and fire when a not unary operator is found
    bool notFlag=false;
    //Variable to store the current Function Name, to determine the current function being processed
    string currentStoredFunctionName;
    CodeGeneratorVisitorNode()=default;
    void PrintProgram();
    static int CalculateJumpAddress(const string& print);
    void visit( ASTNode *pointer)override;
    void visit( ASTProgram *pointer)override;
    void visit( ASTStatement *pointer)override;
    void visit( ASTPrintStatement *pointer)override;
    void visit( ASTDelayStatement *pointer)override;
    void visit( ASTForStatement *pointer)override;
    void visit( ASTWhileStatement *pointer)override;
    void visit( ASTIfStatement *pointer)override;
    void visit( ASTVariableDecl *pointer)override;
    void visit( ASTAssignment *pointer)override;
    void visit( ASTBlock *pointer)override;
    void visit( ASTFunctionDecl *pointer)override;
    void visit( ASTFunctionCall *pointer)override;
    void visit( ASTFormalParams *pointer)override;
    void visit( ASTFormalParam *pointer)override;
    void visit( ASTType *pointer)override;
    void visit( ASTIdentifier *pointer)override;
```

**Figure 38: CodeGeneratorVisitor node**

## Code Generation Process

The required output format of the PixIR code, involved printing the functions in separate blocks. This was achieved through the use of an std::map of strings, which acted as a print buffer. Moreover, depending on the type of AST node, and the current function declaration name, the relevant instruction to be printed would be appended to adequate location in such buffer. Furthermore, a list of function Names was also created in order to determine the order of the function keys of the print buffer to be printed at the end of such pass. Figure 39 illustrates the Print Method, which is used to display all the PixIR instructions in the required format.

```cpp
//Function which Prints the Program by iterating through the different Function Names and printing their respective print
void CodeGeneratorVisitorNode::PrintProgram() {
    //Looping through the list of FunctionNames in order of first stored
    for(auto & functionName : functionNames){
        //Outputting the respective print for the current function Name
        cout << printList[functionName];
    }
}
```

**Figure 39: Function which prints the contents of the print buffer.**

The function illustrated in Figure 40, was created in order to calculate the Jump Address, with respect to Conditional Statements such as If/While and For statements. This was done as the "push #PC" instruction requires an additional integer offset, to indicate the new instruction index to jump to. Continuously, the aforementioned function operates by returning the number of "\n" characters in the passed string parameter.

```cpp
//Function which Calculates the Jump Address, given a string print buffer
int CodeGeneratorVisitorNode::CalculateJumpAddress(const string& print) {
    //Declaring the number of lines
    int noOfLines=0;
    //Looping through all the characters in the print buffer
    for(auto &character : print){
        //Checking whether the current character is a '\n', i.e, indicating a new line, thus incrementing the number of lines
        if(character=='\n'){
            noOfLines++;
        }
    }
    //Returning the number of lines, and adding +1, to cater for the first line
    return noOfLines+1;
}
```

**Figure 40: Function which calculates the Jump Address.**

Additionally, in this pass, the Symbol table was utilised in order to store the memory address of the current Identifier. An example of one of the CodeGeneratorVisitorNode visit functions, which makes use of the Symbol table, can be seen in Figure 41.

```
void CodeGeneratorVisitorNode::visit( ASTAssignment *pointer){
    //Retrieving string Identifier
    string identifier=pointer->identifier->identifier;
    //Accepting expression
    pointer->expression->accept(this);
    //Retrieving the memory address from the symbol table
    string address=symbolTable->ReturnIdentifierAddress(identifier);
    string delimiter=":";
    //Adding the relevant instructions to the printList
    printList[currentStoredFunctionName] += "push "+address.substr(6, address.find(delimiter)-6)+"\n";
    printList[currentStoredFunctionName] += "push "+address.substr(address.find(delimiter)+1, (address.length()-(address.find(delimiter)+3)))+"\n";
    printList[currentStoredFunctionName] += "st\n";
}
```

**Figure 41: CodeGeneratorVisitorNode visit function for ASTAssignment node.**

Figure 42 illustrates the output given from the Code Generation Pass given the following input:

"let x:int = (7*7) +7;".



```
Code Generation Pass:

.main
push 0
oframe
push 1
alloc
push 7
push 7
push 7
mul
add
push 0
push 0
st
cframe
halt
```

**Figure 42: Code Generation Pass Output**

## Code Optimisations

The following two code generation optimisations were implemented, in order to produce more concise code.

**Optimisation 1:** The constructed Compiler proceeds to shorten the code by pre calculating the final value of the Boolean literal, in case that the literal is preceded by multiple not operations. For example, given: "not not true", the Compiler proceeds to only print the "push 1" instruction, instead of the "not \n not \n push 1" sequence of instructions, as the Compiler would have evaluated the expression as true and thus prints "push 1". Additionally, negative expressions, are also evaluated similarly. Figure 43 illustrates the implementation of such optimisation, through the CodeGeneratorVisitorNode visit function for the ASTBoolLiteral node.

```
void CodeGeneratorVisitorNode::visit( ASTBoolLiteral *pointer){
    //Changing value if the notFlag is set
    bool value=pointer->value;
    if(notFlag){
        value=!value;
    }
    //Adding the respective push instruction, based on boolean value to the printList
    if(value){
        printList[currentStoredFunctionName]+="push 1\n";
    }
    else{
        printList[currentStoredFunctionName]+="push 0\n";
    }
}
```

**Figure 43: Optimisation 1: Boolean Literals**

Figure 44 illustrates the output given from the Code Generation Pass given the following input:

"__print not (not (not (false)));".

```
Code Generation Pass:

.main
push 0
oframe
push 1
print
cframe
halt
```

**Figure 44: Code Generation Pass
Optimisation 1**

**Optimisation 2:** The constructed Compiler proceeds to invoke: "push {number of actual parameters} \n oframe" sequence of instructions, instead of invoking multiple "push 1\n alloc" to allocate space for each actual parameter separately, thus reducing the number of lines of the produced PixIR code, with respect to function calls. This was done, as function calls required the creation of a new oframe, in order to allow mapping between the passed parameters in the function call and formal parameters in the matching function declaration. Figure 45 illustrates the implementation of such optimisation, through the CodeGeneratorVisitorNode visit function for the ASTFunctionCall node.

```cpp
void CodeGeneratorVisitorNode::visit( ASTFunctionCall *pointer){
    //Creating new scope to hold the actual Parameters
    Scope funScope;
    symbolTable->push(funScope);
    //Checking whether actualParams is not a nullptr
    if(pointer->actualParams!= nullptr) {
        //Code Optimization: Adding the number of parameters in the beginning of the frame, to avoid multiple push 1 alloc's
        //Adding the respective instructions to the printList
        printList[currentStoredFunctionName] +="push "+ to_string(pointer->actualParams->expressions.size())+"\n";
        printList[currentStoredFunctionName] +="oframe\n";
        pointer->actualParams->accept(this);
    }
    else{
        printList[currentStoredFunctionName] +="push 0\noframe\n";
    }
    //Adding the respective functions to the printList
    printList[currentStoredFunctionName]+= "push ." +pointer->identifier->identifier + "\ncall\n";
    printList[currentStoredFunctionName] +="cframe\n";
    //Popping scope from symbol table
    symbolTable->pop();
}
```

**Figure 45: Optimisation 2: Actual Parameters**

Figure 46 illustrates the output given from the Code Generation Pass given the following input:

"fun test(par1:int,par2:int)->int{return par1;}

let program:int=test(5,7);".

```
Code Generation Pass:

.main
push 0
oframe
push 1
alloc
push 2
oframe
push 7
push 1
push 0
st
push 5
push 0
push 0
st
push .test
call
cframe
push 0
push 0
st
cframe
halt

.test
push 0
oframe
push [0:2]
cframe
ret
```
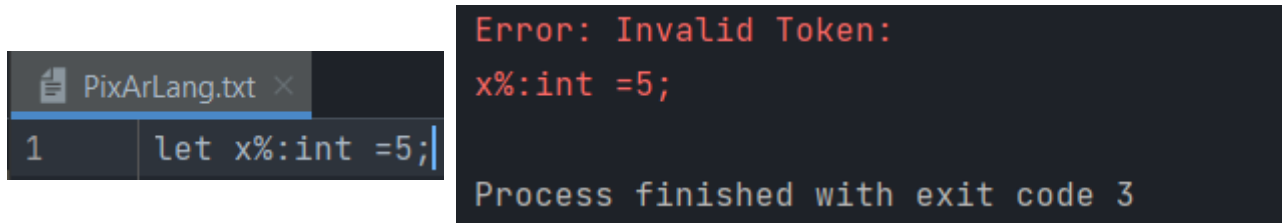
**Figure 46: Code Generation Pass Optimisation 2**

# Testing and Conclusion

## Task 1 Testing:

**Testing for Invalid characters which are not present in the PixArLang programming Language:**
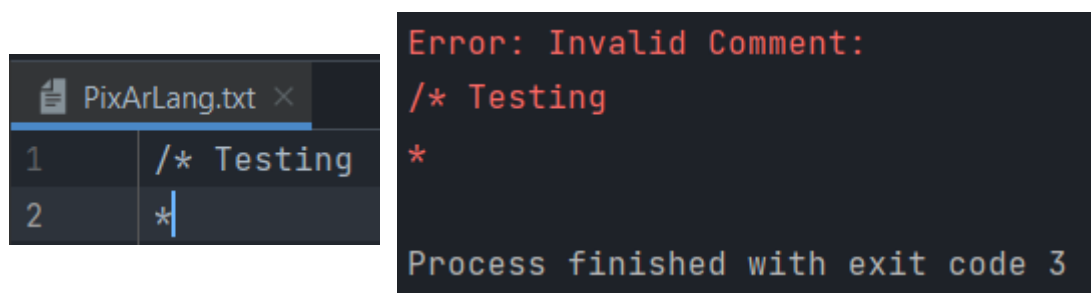


Figure 47: Input and Output for Test Case 1

**Testing for Invalid Comments (Open Comments):**
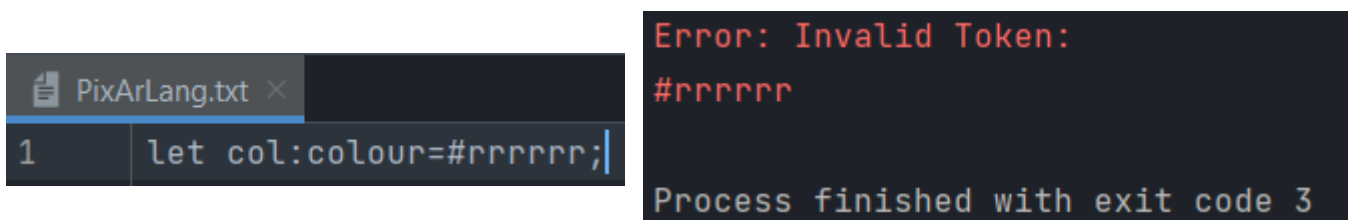


Figure 48: Input and Output for Test Case 2

## Task 2 Testing:

**Testing for Invalid Colour Literal Tokens:**
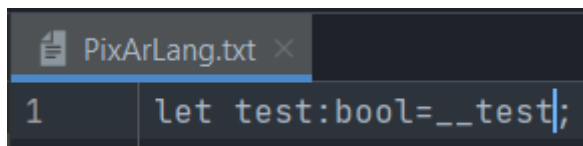


Figure 49: Input and Output for Test Case 3

**Testing for Invalid Special Cases:**
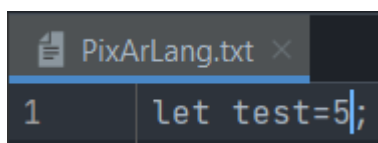


Figure 50: Input and Output for Test Case 4

**Testing for Invalid EBNF statements:**
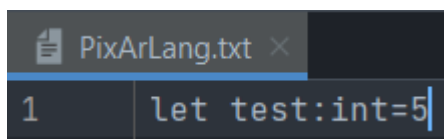


Figure 51: Input and Output for Test Case 5
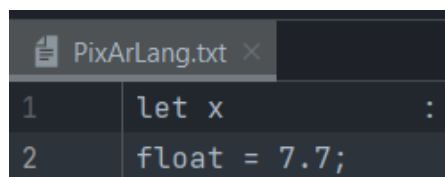


Figure 52: Input and Output for Test Case 6

**Testing for the influence of whitespace and new lines:**
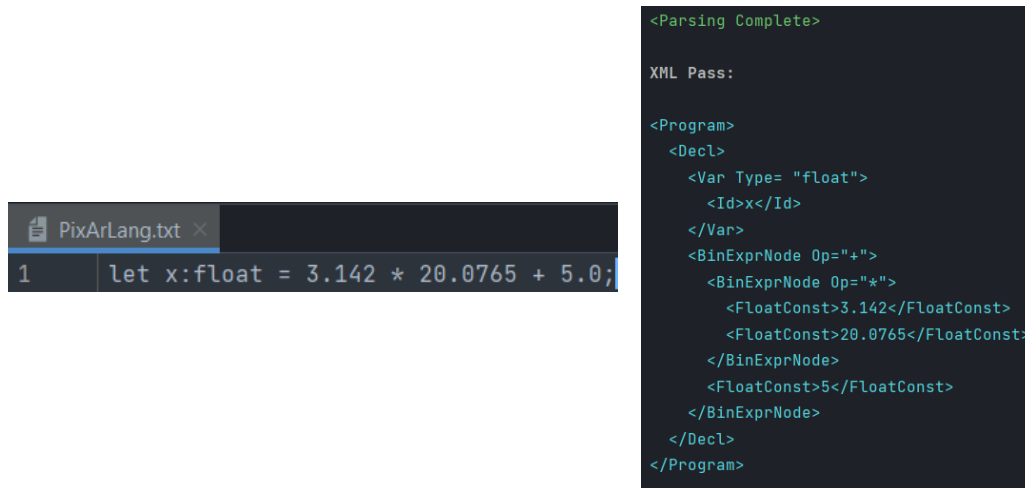


Figure 53: Input and Output for Test Case 7

## Task 3 Testing:

**Testing for Correct Format Printing with Indentation:**



```
<Parsing Complete>

XML Pass:

<Program>
  <Decl>
    <Var Type= "float">
      <Id>x</Id>
    </Var>
    <BinExprNode Op="+">
      <BinExprNode Op="*">
        <FloatConst>3.142</FloatConst>
        <FloatConst>20.0765</FloatConst>
      </BinExprNode>
      <FloatConst>5</FloatConst>
    </BinExprNode>
  </Decl>
</Program>
```

```
PixArLang.txt
1    let x:float = 3.142 * 20.0765 + 5.0;
```
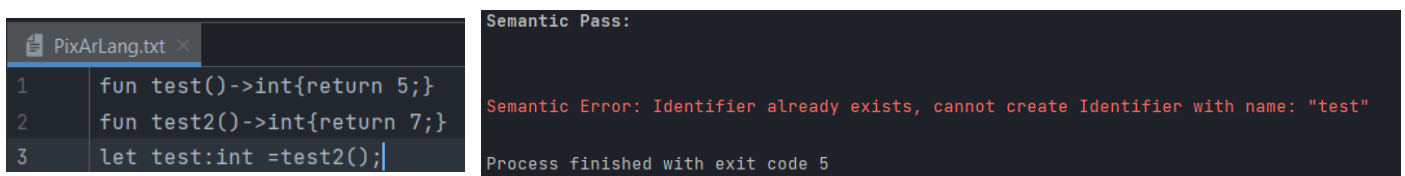
**Figure 54: Input and Output for Test Case 8**

## Task 4 Testing:

**Testing for Semantic Errors:**

```
PixArLang.txt
1    fun test()->int{return 5;}
2    fun test2()->int{return 7;}
3    let test:int =test2();
```
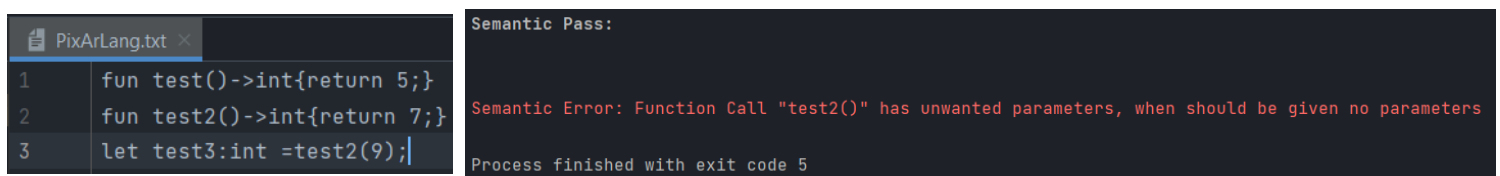
```
Semantic Pass:

Semantic Error: Identifier already exists, cannot create Identifier with name: "test"

Process finished with exit code 5
```

**Figure 55: Input and Output for Test Case 9**

```
PixArLang.txt
1    fun test()->int{return 5;}
2    fun test2()->int{return 7;}
3    let test3:int =test2(9);
```

```
Semantic Pass:

Semantic Error: Function Call "test2()" has unwanted parameters, when should be given no parameters

Process finished with exit code 5
```

**Figure 56: Input and Output for Test Case 10**

**Figure 57: Input and Output for Test Case 11**



**Figure 58: Input and Output for Test Case 12**



**Figure 59: Input and Output for Test Case 13**



**Figure 60: Input and Output for Test Case 14**



**Figure 61: Input and Output for Test Case 15**

**Figure 62: Input and Output for Test Case 16**



**Figure 63: Input and Output for Test Case 17**

**Testing for Type Mismatch Errors:**



**Figure 65: Input and Output for Test Case 18**



**Figure 64:  Input and Output for Test Case 19**



**Figure 66: Input and Output for Test Case 20**

**Figure 67: Input and Output for Test Case 21**



**Figure 68: Input and Output for Test Case 22**
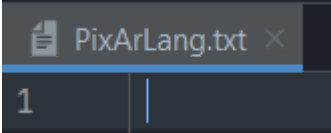


**Figure 70: Input and Output for Test Case 24**

## Task 5 Testing:

**Testing with Empty File:**
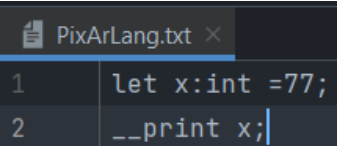


**Figure 71: Input and Output for Test Case 25**

**Testing functionality of Code Generation utilising VM Box:**



**Figure 72: Input, Compiler Output and VM Output for Test Case 26**

**Testing Correct functionality of variables for the Generated Code utilising VM Box:**

```
PixArLang.txt  ×
1      let x:int=5;
2      fun test(y:int)->int{__print x; return y;}
3      let z:int=test(8);
4      __print z;
```

```
.main                             =>>> logs start here
push 0                            [1/35] push 0 |=> [ 0 ]
oframe                            [2/35] oframe |=> [ ]
push 1                            [3/35] push 1 |=> [ 1 ]
alloc                             [4/35] alloc |=> [ ]
push 5                            [5/35] push 5 |=> [ 5 ]
push 0                            [6/35] push 0 |=> [ 0 5 ]
push 0                            [7/35] push 0 |=> [ 0 0 5 ]
st                                [8/35] st |=> [ ]
push 1                            [9/35] push 1 |=> [ 1 ]
alloc                             [10/35] alloc |=> [ ]
push 1                            [11/35] push 1 |=> [ 1 ]
oframe                            [12/35] oframe |=> [ ]
push 8                            [13/35] push 8 |=> [ 8 ]
push 0                            [14/35] push 0 |=> [ 0 8 ]
push 0                            [15/35] push 0 |=> [ 0 0 8 ]
st                                [16/35] st |=> [ ]
push .test                        [17/35] push .test |=> [ 28 ]
call                              [18/35] call |=> [ ]
cframe                            [28/35] push 0 |=> [ 0 ]
push 1                            [29/35] oframe |=> [ ]
push 0                            [30/35] push [0:3] |=> [ 5 ]
st                                -- 5
push [1:0]                        [31/35] print |=> [ ]
print                             [32/35] push [0:2] |=> [ 8 ]
cframe                            [33/35] cframe |=> [ 8 ]
halt                              [34/35] ret |=> [ 8 ]
                                  [19/35] cframe |=> [ 8 ]
.test                             [20/35] push 1 |=> [ 1 8 ]
push 0                            [21/35] push 0 |=> [ 0 1 8 ]
oframe                            [22/35] st |=> [ ]
push [0:3]                        [23/35] push [1:0] |=> [ 8 ]
print                             -- 8
push [0:2]                        [24/35] print |=> [ ]
cframe                            [25/35] cframe |=> [ ]
ret                               [26/35] halt |=> [ ]
                                  [Program Execution] HALT
```

Figure 73: Input, Compiler Output and VM Output for Test Case 27

## Conclusion

In summary, constructing a compiler is quite a challenging task that necessitates a thorough knowledge of Computer Architecture and Programming Languages. Furthermore, in this project we have discussed the different stages of constructing a **Front-End Compiler**, such as Lexical and Syntax Analysis. Moreover, throughout the construction of the aforementioned PixArDis Compiler, there were various difficulties which were encountered. To name a few:

1. Construction of the Transition Table was quite a tedious and monotonous task. Moreover, there were several times, where a small typo in of the tables would drastically change the whole functionality of the Lexer.
2. Creating the different Parsing function and AST Nodes was also quite a repetitive task. Additionally, it was also somewhat challenging to understand the structure of the class Hierarchy, particularly with respect to the AST Expression node.
3. Finally, it was also slightly tricky to understand the PixIR code generation translation cases, especially for the calling of specific identifiers, due to the stack of frames implementation, which the VM Box utilises.

Nevertheless, the result obtained outweighed the difficulties encountered, and it is quite rewarding to see through the illustrations of Figures 74-76, the visually pleasing patterns produced through the PixArLang code, which is fed to the constructed Compiler, and whose PixIR code is later fed to the VM Box.
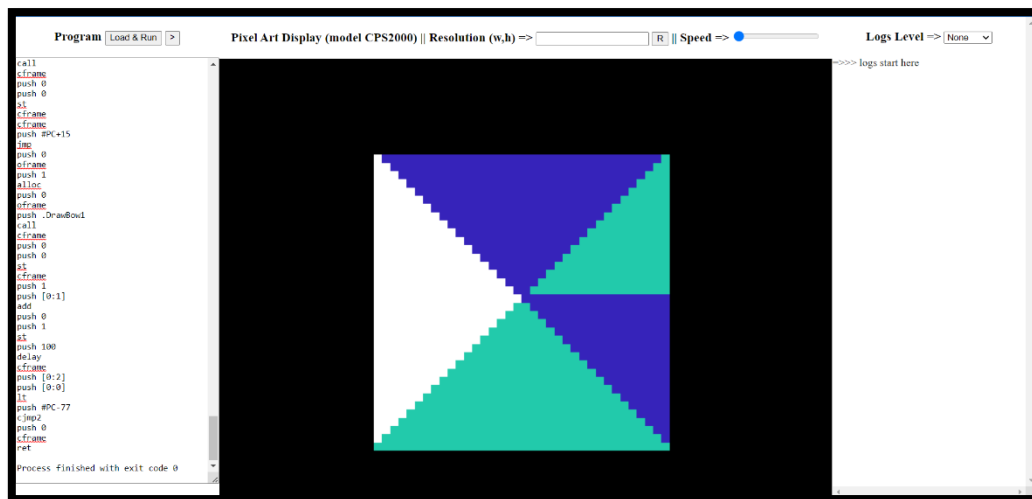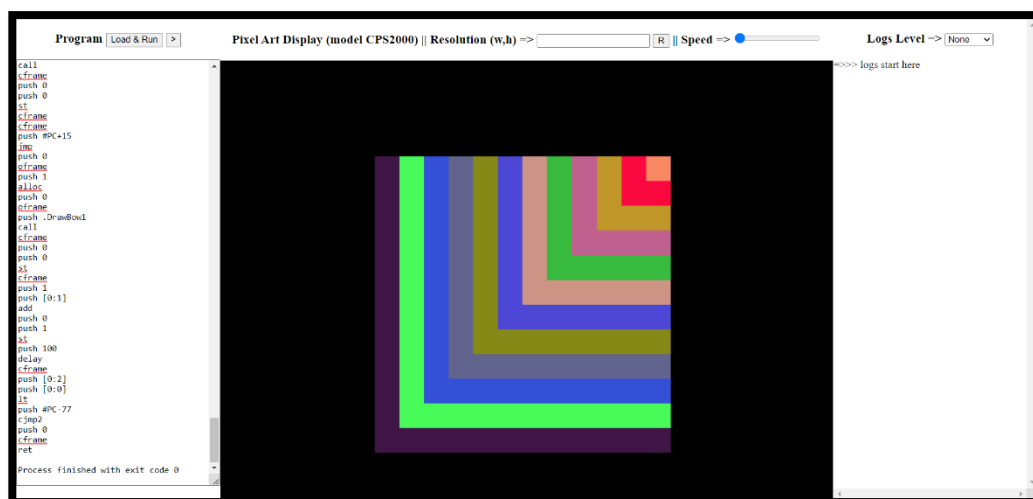
**Figure 75: VM Pattern Design Output 1**



**Figure 74: VM Pattern Design Output 2**



**Figure 76: VM Pattern Design Output 3**

# Plagiarism Declaration Form