

**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**

Table of Contents

[Implementation 3](#_Toc133599742)

[Task 1 - Table-driven Lexer 5](#_Toc133599743)

[Deterministic Finite State Automata (DFSA) 5](#_Toc133599744)

[CAT Table 7](#_Toc133599745)

[Token Table 7](#_Toc133599746)

[Identifier/Keyword Table 7](#_Toc133599747)

[Transition Table 7](#_Toc133599748)

[Overview of Lexical Analysis 10](#_Toc133599749)

[Task 2 - Hand-crafted LL(k) parser 12](#_Toc133599750)

[Lookahead Tokens 12](#_Toc133599751)

[Changes to EBNF 14](#_Toc133599752)

[Parse Functions 15](#_Toc133599753)

[Syntax Tree Construction 16](#_Toc133599754)

[Visitor Node Design Pattern 18](#_Toc133599755)

[Task 3 - AST XML Generation Pass 21](#_Toc133599756)

[Task 4 - Semantic Analysis Pass 21](#_Toc133599757)

[Task 5 - PixIR Code Generation Pass 21](#_Toc133599758)

[Testing and Conclusions 21](#_Toc133599759)

# 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.

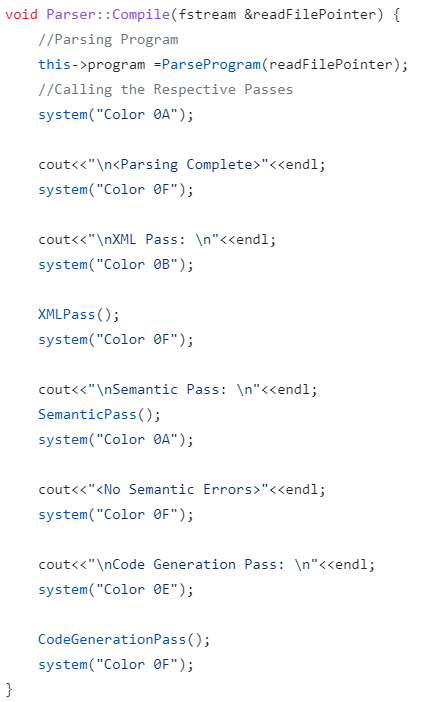
The following are the **csv table files** used:

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** whichholds the PixArLang code to compile.



**Figure 1: Main Class Method**



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

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.

# Task 1 - Table-driven Lexer

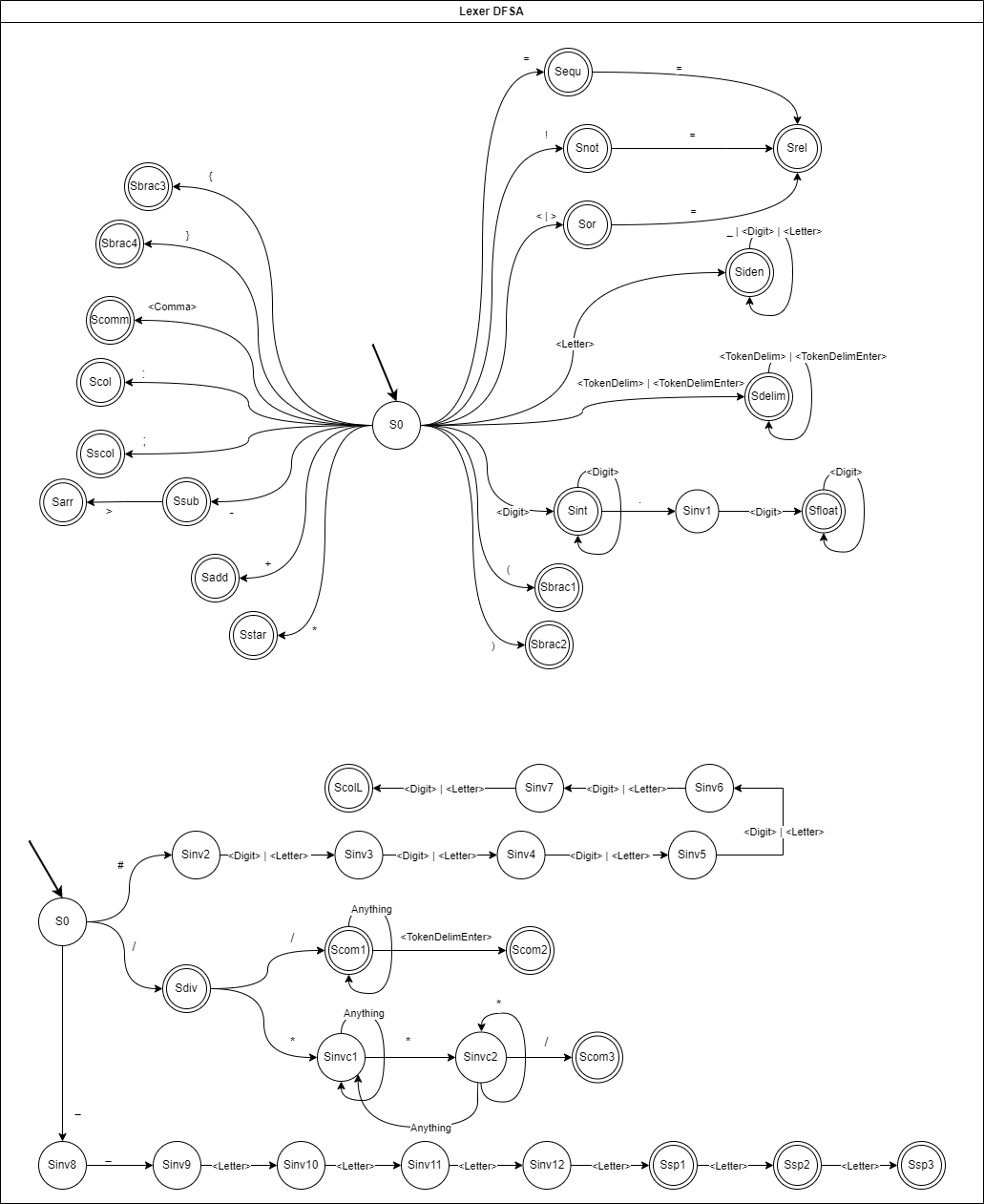
A crucial step in the Compiling process, is the **Lexical Analysis**, which ultimately is responsible for tokenising the program into tokens, whilst identifying erroneous tokens and characters. Implementation of the Lexer class can be seen in Figure 3.



**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

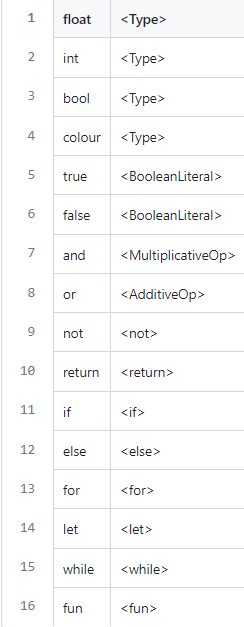
The identifier/Keyword table is used to map special keywords which have been mapped as identifiers. This was done as special keywords are composed of <Letter> tokens, which through the DFSA, are mapped as <Identifier> tokens. An example of such token would be the “let” keyword, which is used specifically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

## Transition Table

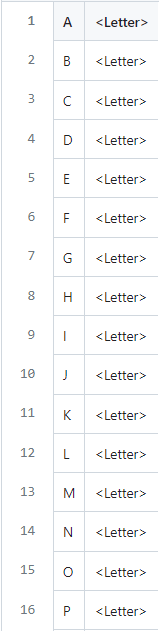
The transition table is used to map the transition between two states in the DFSA, i.e., given the current state and the transition label (current token), one would be able to determine the new state. For example, given the current state of S0 and the token of <Comma>, one can easily use such table to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



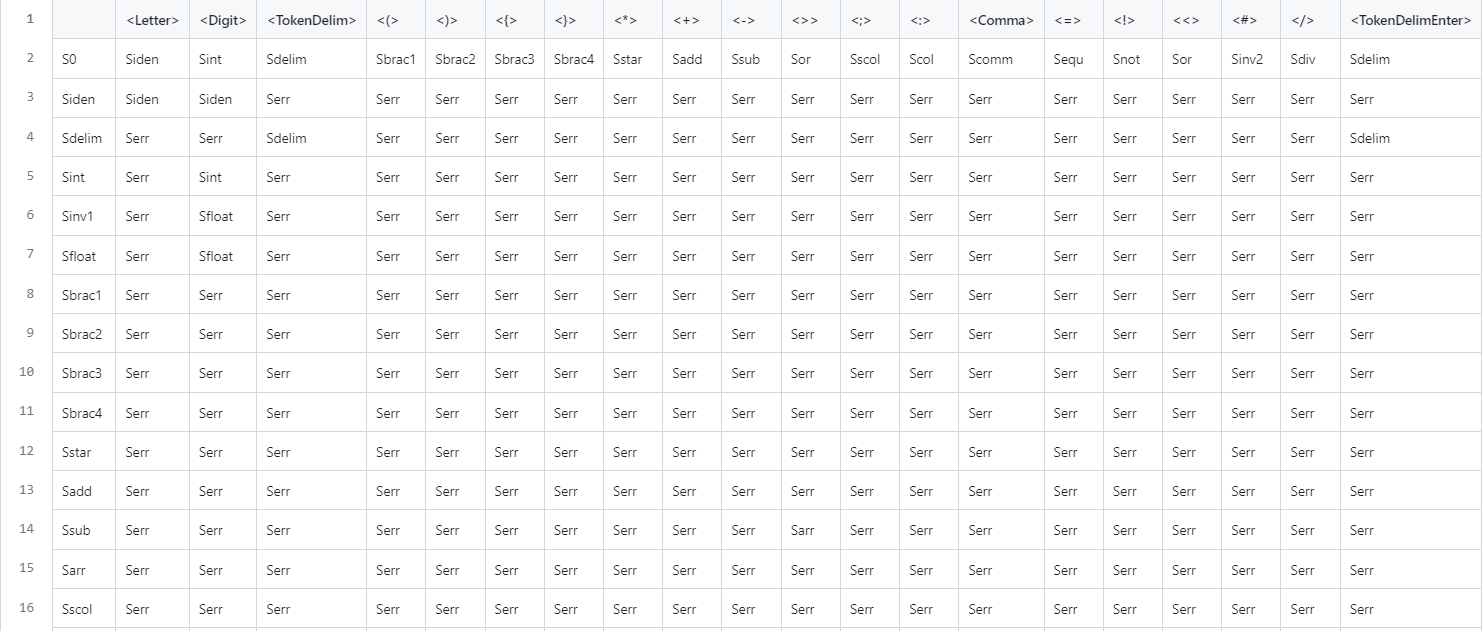
**Figure 5: Token Table**



**Figure 6: Identifier/Keyword Table**



**Figure 7: CAT Table**



**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.



**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

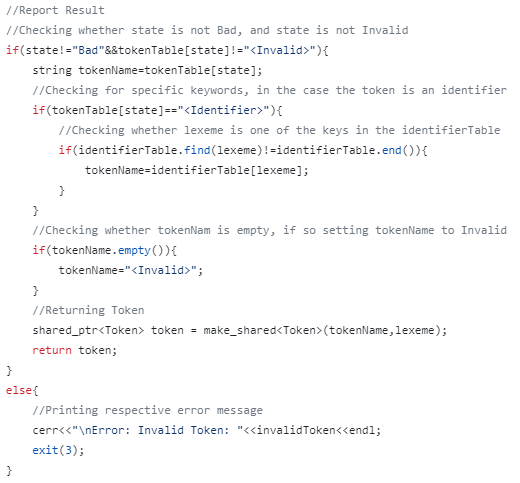
This process entails looping until the current state is Invalid and the Lexer stack is not empty. Moreover, for every iteration in the Rollback Loop, a state is popped from the stack, and the lexeme is truncated. This process aims to extract a valid token with the largest size. Moreover, this process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

Printing Results

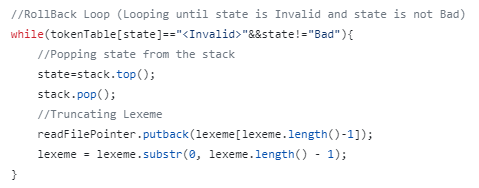
This process entails returning the relevant token, through the state look up in the token table. Furthermore, in this step, the Lexer also checks whether Identifier tokens map to special keywords, through lookup in identifier table. Conclusively, a token data structure is returned, which holds the current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



**Figure 10: Scanning Loop**



**Figure 11: Printing Results**



**Figure 12: Rollback Loop**

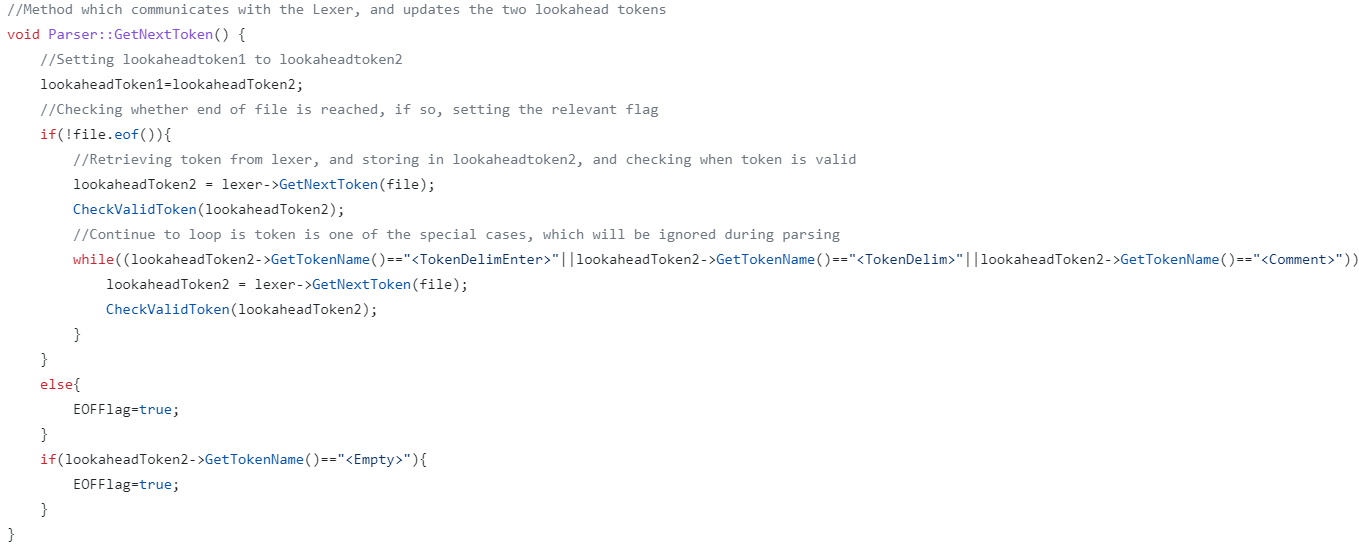
# Task 2 - Hand-crafted LL(k) parser

**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.



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



**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〉 ‘;’

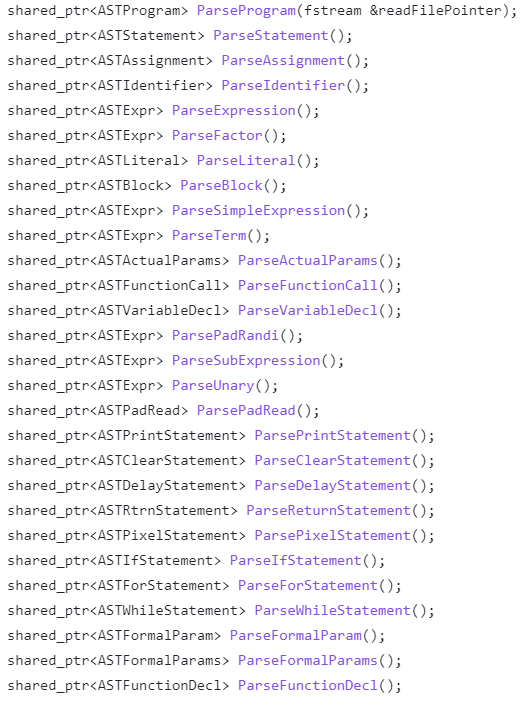
| 〈FunctionDecl〉

| 〈Block〉

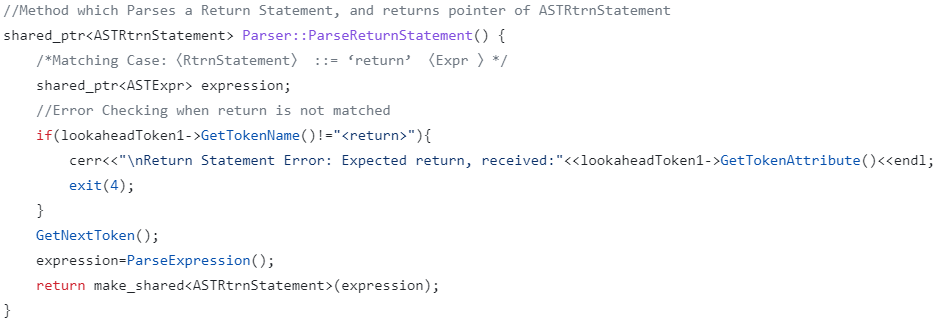
**〈ClearStatement〉 ::= ‘\_\_clear’ <ColourLiteral>**

## 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.



**Figure 15: Parsing Functions**

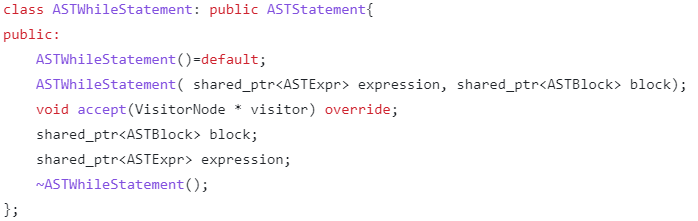


**Figure 16: Parse Return Statement function**

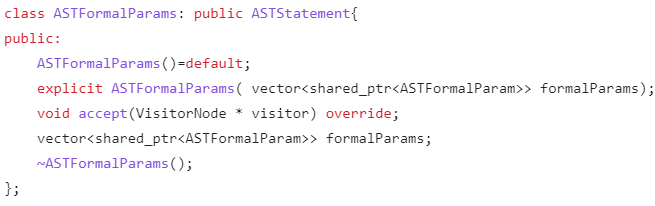
## Syntax Tree Construction

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.

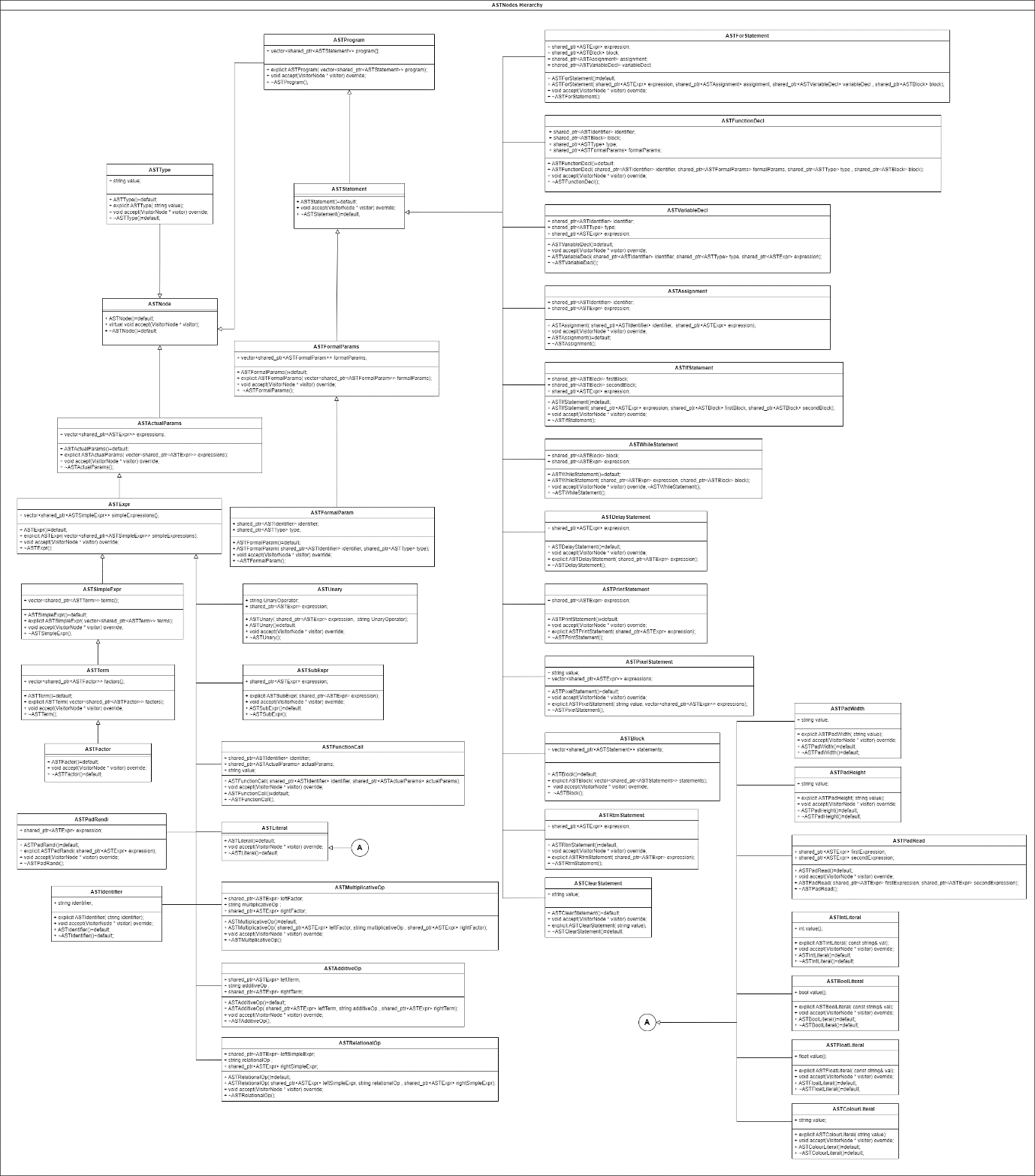
In continuation, each AST Node was designed to have different properties, for example, the ASTWhileStatement Node contains references to ASTBlock and ASTExpr Nodes. On the other hand, the ASTFormalParams Node only contains references to a vector of ASTFormalParam Nodes. This difference can be seen in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



**Figure 17: ASTWhileStatement Node**



**Figure 18: ASTFormalParams Node**

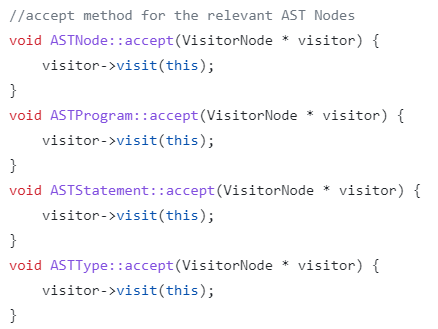


**Figure 19: Abstract Syntax Tree Node Hierarchy**

# 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 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.

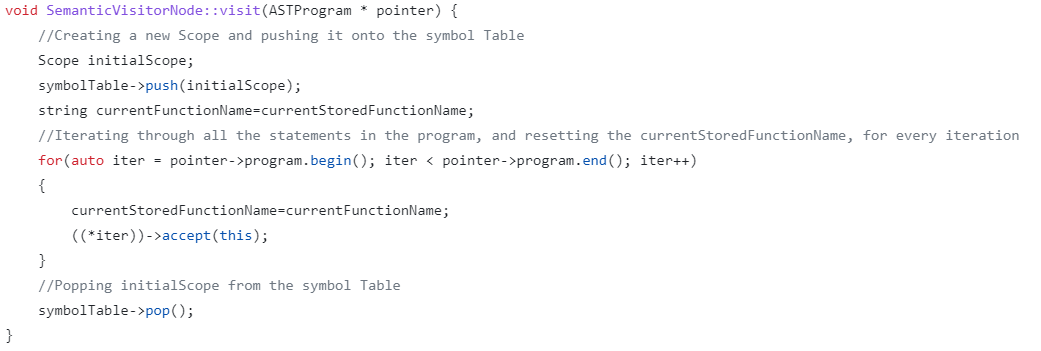


**Figure 20: AST Node accept methods.**

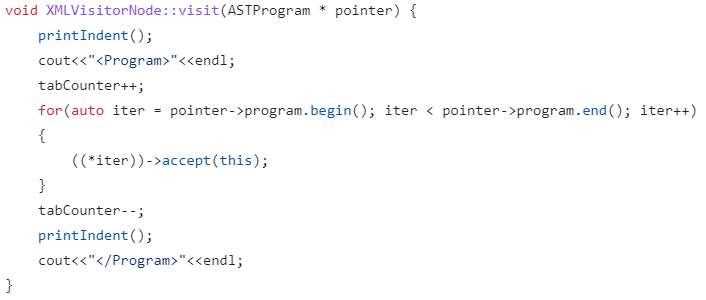


**Figure 21: Visitor Node class**

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.



**Figure 23: Semantic Visitor Node visit ASTProgram**



**Figure 22: XML Visitor Node visit ASTProgram**

# Task 3 - AST XML Generation Pass

Printing Indented Tags



Text

Description automatically generated

# Task 4 - Semantic Analysis Pass

Symbol table

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

# Task 5 - PixIR Code Generation Pass

Code optimization

# Testing and Conclusions

Plagiarism Declaration Form