L-Università ta' Malta

CPS2000

# Compiler Theory and Practice

Course Assignment 2022/2023

Programmed using Python

By Mark Dingli

mark.dingli.21@um.edu.mt
20703H

# Table of contents

## Plagiarism Declaration Form

| | | |
|---|---|---|
| Mark Dingli | CPS2000 | 01/06/2023 |
| **Student's full name** | **Study-unit code** | **Date of submission** |

**Title of submitted work:** Compiler Theory and Practice Assignment

**Student's Signature**

[illegible]

# Statement of Completion

| Task | State |
|---|---|
| **Task 1** – Table-driven lexer | COMPLETED |
| **Task 2** – Hand-crafted LL(k) parser | COMPLETED |
| **Task 3** – AST XML Generation Pass | COMPLETED |
| **Task 4** – Semantic Analysis Pass | COMPLETED |
| **Task 5** – PixIR Code Generation Pass | PARTIALLY COMPLETED |

**Comment:** It should be noted that within Task 5, the PixIR Code Generation Pass, the *'for'* loop and *'else'* statement structures were not fully implemented.

# Introduction

This documentation provides an in-depth explanation of the PixArLang Compiler, designed for the transformation of PixArLang source code into PixIR, a language designed for execution on the PAD2000 hardware. This Compiler was implemented using Python.

The *PixArLang* Compiler encompasses five significant tasks, each contributing a distinct aspect to the overall compilation process. These tasks form the full compilation pipeline, transforming *PixArLang* source code into a form executable by the PAD2000 hardware.

Task 1, Lexical Analysis, is the first step in the compilation process. It involves the translation of the source code into a sequence of tokens, each representing a distinct lexeme in the *PixArLang* language.

Task 2, Syntactic Analysis, builds upon the tokens generated in Task 1, producing an abstract syntax tree (AST) that represents the structure of the source code.

Task 3, AST XML Generation, forms an integral part of the debugging process by generating an XML representation of the AST, allowing for easier visualization of the program's structure.

Task 4, Semantic Analysis, ensures that the source code adheres to the semantic rules of the *PixArLang* language. By walking through the AST, it checks for potential semantic errors that may not have been identified during the earlier stages.

Task 5, *PixIR* Code Generation, translates the semantically validated AST into the target PixIR code, ready for execution on the *PAD2000* hardware.

Lastly, testing the *PixArLang* Compiler is a straightforward process. To evaluate *PixArLang* code, simply enter it into the file named *"input.txt"*. The compiler will read the source code from this file and initiate the compilation process. Upon running the compiler, it will progress through each of the five tasks - lexical analysis, syntactic analysis, AST XML generation, semantic analysis, and finally, PixIR code generation. To observe the entire process and all five tasks in action, you must execute the Task 5 file named *"code_generation.py"*. Running this file will lead the compiler through the entire pipeline, demonstrating the transformation of your *PixArLang* code into the executable PixIR form.

**Note:**

**To run the code effectively, make sure that you un-zip the folder called *"code.zip"*, afterwards it is important that the whole *"compiler_code"* folder is opened in the IDE. This will ensure that the path to the input file works.**

# Task 1 - Table-driven Lexer

## Description

The lexer developed for the PixArLang language employs a table-driven approach [1]. Its primary responsibility is to scan an input program, identify and also validate the tokens within it. The lexer operates based on the predefined micro-syntax of PixArLang, established prior to the lexer's implementation, and the Extended Backus-Naur Form (EBNF) of the language.

The lexer reads the input program character by character. Using the current state of the DFA and the transition table, it determines the next state. Upon encountering an accepting state, the lexer emits the corresponding token and resets the DFA to its initial state. If the next state is an error state, the lexer flags a lexical error and terminates.

At the heart of the lexer is the transition table, implemented as a dictionary. This dictionary encapsulates the DFA transition function of the PixArLang micro-syntax. The keys in this dictionary are pairs of the current state and character, while the values are the new states. These states correspond to various token types, including identifiers, integer literals, operators, delimiters and others. The transition table is constructed using the *build_transition_table* method, which sets the transitions for all states and input characters according to the language's rules [2].

The table-driven approach offers significant benefits. It facilitates efficient and predictable operations, as the lexer can promptly look up its next action in the table, bypassing the need for executing complex logic or computations. Moreover, the implementation of the transition table as a dictionary allows for swift scanning of the input program by simulating the DFA with a simple dictionary lookup. This design enhances the lexer's speed and reliability.

The *get_token* method serves a crucial role in reading the source code character by character and recognizing tokens using the transition table. It maintains the current state, the position in the source code, and the lexeme being recognized. If the next character leads to a transition to a new state, it is added to the lexeme. If no transition exists for the current state and character, the method checks if the current lexeme forms a valid token and, if so, returns it. If not, an error is raised.

This is an example of how the transition table handles the lexeme *"int"*:

- The initial state is 0.
- The character *"i"* transitions from state 0 to state 109.
- The character *"n"* transitions from state 109 to state 110.
- The character *"t"* transitions from state 110 to state 111.
- The final state is 111, which corresponds to the token type *"TYPE_INT"*.
- The method get_token_type_from_state then retrieves the token type *"TYPE_INT"* .

Auxiliary methods are also present in the Lexer class. The *get_next_char* method manages reading the next character from the source code and handling escape sequences in string literals. The *get_token_type_from_state* method determines the recognized token type based on the current state and lexeme.

**It is important to mention that there were some deviations from the original EBNF. More specifically, expressions in statements need to be enclosed in brackets. For instance, the statement:**

*"__pixel u , v,#00ff00;"*

**would now be written as** "__pixel (u , v,#00ff00);"**.**

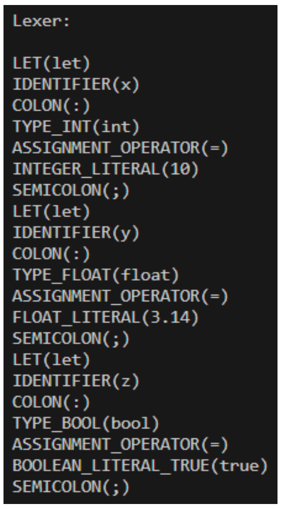
**Another example is:**

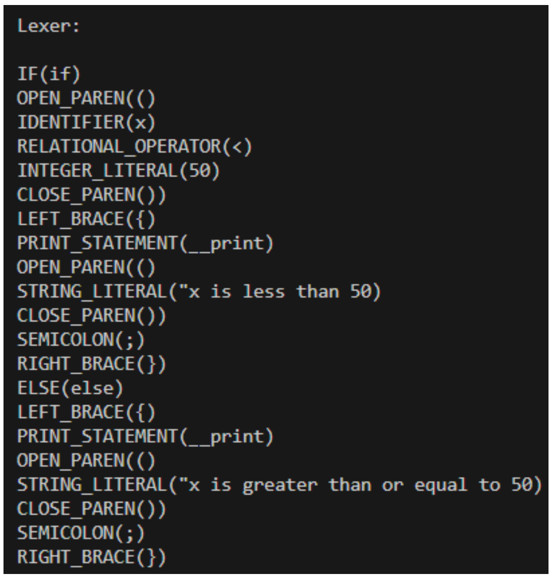
*"__width;"*

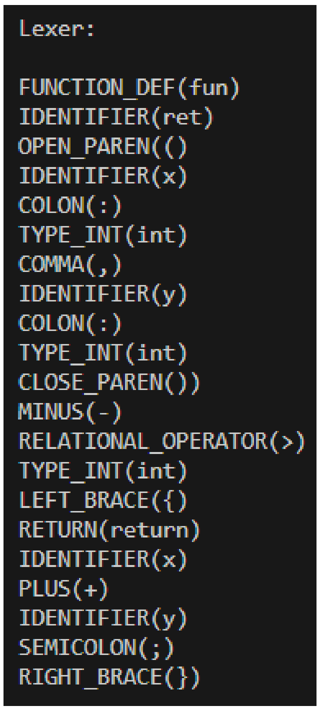
**which should be written as** "__width();". **These changes, while minimal, enhance the structure and readability of the language, making it easier to use these statements.**

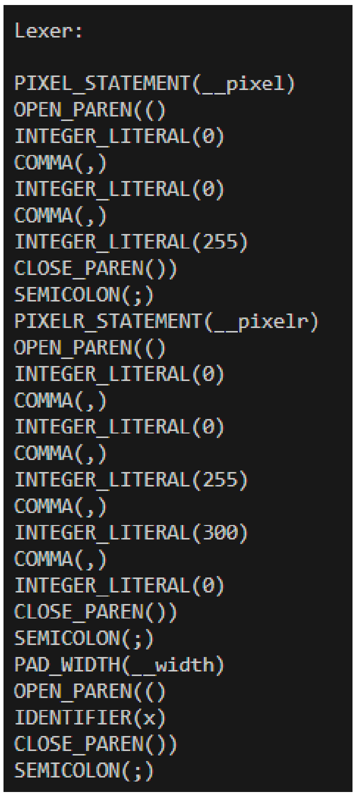
In conclusion, the implemented PixArLang lexer effectively illustrates the efficiency and reliability of the table-driven approach in a practical application. By integrating a dictionary-based transition table and leveraging the benefits of a DFA, the lexer efficiently scans the input program, identifies tokens, and handles lexical errors. Furthermore, the adaptations to the original EBNF, like the inclusion of brackets in expressions, demonstrate the lexer's flexibility to accommodate modifications for better usability and readability.

## Testing & Output

| INPUT 1 | OUTPUT 1 | COMMENTS |
|---|---|---|
| let x: int = 10; // declaring int<br>let y: float = 3.14; // declaring float<br>let z: bool = true; // declaring boolean | <br>`Lexer:`<br><br>`LET(let)`<br>`IDENTIFIER(x)`<br>`COLON(:)`<br>`TYPE_INT(int)`<br>`ASSIGNMENT_OPERATOR(=)`<br>`INTEGER_LITERAL(10)`<br>`SEMICOLON(;)`<br>`LET(let)`<br>`IDENTIFIER(y)`<br>`COLON(:)`<br>`TYPE_FLOAT(float)`<br>`ASSIGNMENT_OPERATOR(=)`<br>`FLOAT_LITERAL(3.14)`<br>`SEMICOLON(;)`<br>`LET(let)`<br>`IDENTIFIER(z)`<br>`COLON(:)`<br>`TYPE_BOOL(bool)`<br>`ASSIGNMENT_OPERATOR(=)`<br>`BOOLEAN_LITERAL_TRUE(true)`<br>`SEMICOLON(;)` | The output shows the Lexer's tokenization of the input script. It identifies keywords (LET), identifiers (x, y, z), type indicators (TYPE_INT, TYPE_FLOAT, TYPE_BOOL), assignment operators (=), literal values (INTEGER_LITERAL, FLOAT_LITERAL, BOOLEAN_LITERAL_TRUE), and statement terminators (;). Each token is grouped with its respective value in parentheses. |

| INPUT 2 | OUTPUT 2 | COMMENTS |
|---|---|---|
| if (x < 50) {<br>  __print("x is less than 50");<br>} else {<br>  __print("x is greater than or equal to 50");<br>} | <br>`Lexer:`<br><br>`IF(if)`<br>`OPEN_PAREN(()`<br>`IDENTIFIER(x)`<br>`RELATIONAL_OPERATOR(<)`<br>`INTEGER_LITERAL(50)`<br>`CLOSE_PAREN())`<br>`LEFT_BRACE({)`<br>`PRINT_STATEMENT(__print)`<br>`OPEN_PAREN(()`<br>`STRING_LITERAL("x is less than 50)`<br>`CLOSE_PAREN())`<br>`SEMICOLON(;)`<br>`RIGHT_BRACE(})`<br>`ELSE(else)`<br>`LEFT_BRACE({)`<br>`PRINT_STATEMENT(__print)`<br>`OPEN_PAREN(()`<br>`STRING_LITERAL("x is greater than or equal to 50)`<br>`CLOSE_PAREN())`<br>`SEMICOLON(;)`<br>`RIGHT_BRACE(})` | The output showcases the Lexer's tokenization of the input script. It recognizes keywords (IF, ELSE), identifiers (x), relational operators (<), literal values (INTEGER_LITERAL, STRING_LITERAL), punctuation marks (OPEN_PAREN, CLOSE_PAREN, LEFT_BRACE, RIGHT_BRACE, SEMICOLON), and function calls (PRINT_STATEMENT). |

| INPUT 3 | OUTPUT 3 | COMMENTS |
|---|---|---|
| fun ret(x: int, y: int) -> int {<br>  return x + y;<br>} | <br>Lexer:<br><br>FUNCTION_DEF(fun)<br>IDENTIFIER(ret)<br>OPEN_PAREN(()<br>IDENTIFIER(x)<br>COLON(:)<br>TYPE_INT(int)<br>COMMA(,)<br>IDENTIFIER(y)<br>COLON(:)<br>TYPE_INT(int)<br>CLOSE_PAREN())<br>MINUS(-)<br>RELATIONAL_OPERATOR(>)<br>TYPE_INT(int)<br>LEFT_BRACE({)<br>RETURN(return)<br>IDENTIFIER(x)<br>PLUS(+)<br>IDENTIFIER(y)<br>SEMICOLON(;)<br>RIGHT_BRACE(}) | The output shows how the Lexer tokenizes the input script. It accurately identifies keywords (FUNCTION_DEF, RETURN), identifiers (x, y, ret), type indicators (TYPE_INT), punctuation marks (OPEN_PAREN, CLOSE_PAREN, LEFT_BRACE, RIGHT_BRACE, SEMICOLON, COLON, COMMA), and operators (PLUS, MINUS). However, the arrow indicating the return type in the function signature appears to have been tokenized incorrectly as a combination of the MINUS and RELATIONAL_OPERATOR tokens. |

| INPUT 4 | OUTPUT 4 | COMMENTS |
|---|---|---|
| __pixel(0, 0, 255);<br>__pixelr(0, 0, 255, 300, 0);<br>__width(x); // Testing expressions with brackets | <br>Lexer:<br><br>PIXEL_STATEMENT(__pixel)<br>OPEN_PAREN(()<br>INTEGER_LITERAL(0)<br>COMMA(,)<br>INTEGER_LITERAL(0)<br>COMMA(,)<br>INTEGER_LITERAL(255)<br>CLOSE_PAREN())<br>SEMICOLON(;)<br>PIXELR_STATEMENT(__pixelr)<br>OPEN_PAREN(()<br>INTEGER_LITERAL(0)<br>COMMA(,)<br>INTEGER_LITERAL(0)<br>COMMA(,)<br>INTEGER_LITERAL(255)<br>COMMA(,)<br>INTEGER_LITERAL(300)<br>COMMA(,)<br>INTEGER_LITERAL(0)<br>CLOSE_PAREN())<br>SEMICOLON(;)<br>PAD_WIDTH(__width)<br>OPEN_PAREN(()<br>IDENTIFIER(x)<br>CLOSE_PAREN())<br>SEMICOLON(;) | The output shows how the Lexer tokenizes the input script. It accurately identifies function calls (PIXEL_STATEMENT, PIXELR_STATEMENT, PAD_WIDTH), identifiers (x), integer literals, punctuation marks (OPEN_PAREN, CLOSE_PAREN, SEMICOLON, COMMA), and operators.<br><br>Each token in the input script is correctly identified by the Lexer and paired with its respective value in parentheses. |

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

## Description

This part describes the implementation of a hand-crafted LL(k) parser [3][4], an integral part of a compiler responsible for translating source code into an intermediary format.

The parser's operations commence with the reception of a list of tokens from the Lexer, the earlier stage in the compiler pipeline. Once the parser has the list of tokens, it sequentially iterates over them, examining the type of each token and comparing it with the expected token type. When a match is found, the parser advances to the next token, repeating this process until the entire program is parsed. This parsing process is guided by a set of grammar rules specific to the language being parsed. Each function in the Parser corresponds to a different grammar rule, enabling the detection and appropriate parsing of various kins of statements like assignments, if-statements, function definitions, and more.

The parser is housed within a class named Parser, which holds the list of tokens and the current index of the token being parsed. It also maintains a symbol table to store information about declared variables, facilitating the tracking of variable usage and scope within the source code.

The entry point to the parsing process is the *parse()* method. This method starts parsing the tokens and appends the parsed statements into the program until all tokens have been processed. The *parse_statement()* method is utilized to handle different types of statements based on the current token's type, delegating the parsing task to the appropriate function according to the statement type.

One of the main objectives of the parser is to generate an Abstract Syntax Tree (AST) from the sequence of tokens. The AST is a tree-like representation of the source code's structure, where each node represents a construct occurring in the source code. This structure is more compact than a parse tree while retaining all the necessary information to facilitate the compiler's understanding of the code's structure and control flow.

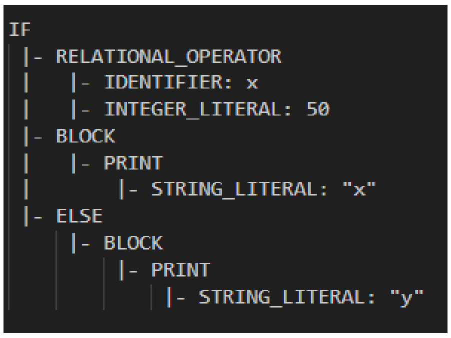
The generation of the Abstract Syntax Tree (AST) [5] is a central part of the parsing process, representing the syntax of the source code in a hierarchical and intuitive manner. As the parser scans through the tokens, it constructs the AST using the grammar rules of the language. Each token, depending on its type and context, triggers a specific method in the parser that corresponds to a grammar rule. These methods create nodes in the AST, each representing a different construct in the code such as a variable declaration, an assignment, an if statement, and so forth. The relationships between these nodes mirror the relationships between the corresponding constructs in the source code, thereby encapsulating the source code's syntax and structure.

In this implementation, the AST is represented as a list of tuples. Each tuple contains information about a specific construct in the code, such as its type and associated value. For instance, a construct like an if-statement would be represented as a tuple where the first element indicates that it's an *'IF'* statement, and the subsequent elements represent the condition and the statement block to execute. This way, the list of tuples serves as a form of AST, encapsulating the hierarchical structure of the source code in a format that's easy for the subsequent stages of the compiler to process.

```
[('IF', ('RELATIONAL_OPERATOR', ('IDENTIFIER', 'x'), ('INTEGER_LITERAL',
50)), ('BLOCK', [('PRINT', ('STRING_LITERAL', '"x'))]), ('ELSE', ('BLOCK',
[('PRINT', ('STRING_LITERAL', '"y'))]))))]
```

*Figure 1 - AST Output of an IF-ELSE Statement*

This tuple structure AST output (Figure 1) corresponds to this:



```
IF
 |- RELATIONAL_OPERATOR
 |   |- IDENTIFIER: x
 |   |- INTEGER_LITERAL: 50
 |- BLOCK
 |   |- PRINT
 |       |- STRING_LITERAL: "x"
 |- ELSE
     |- BLOCK
         |- PRINT
             |- STRING_LITERAL: "y"
```

*Figure 2 - Tuple AST Structure of IF-ELSE Statement*

If the Parser encounters a token sequence that does not conform to any of the expected patterns, it raises a *ParserError*. This exception interrupts the normal flow of the program and provides an error message, aiding in the debugging and correction of syntax errors.

In essence, the Parser translates the source code into a structured format, the AST, simplifying the compiler's subsequent understanding and processing of the code. This conversion is a crucial step in the compilation process, laying the foundation for the subsequent stages like semantic analysis and code generation.

## Testing & Output

| INPUT 1 | OUTPUT 1 | COMMENTS |
|---|---|---|
| for (let x: int = 0; x < 10; x = x + 1) {<br>__print(x);<br>} | Parsed program:<br><br>[('FOR', ('DECLARATION', 'TYPE_INT', 'x', ('INTEGER_LITERAL', 0)), ('RELATIONAL_OPERATOR', ('IDENTIFIER', 'x'), ('INTEGER_LITERAL', 10)), ('ASSIGNMENT', '=', ('PLUS', ('IDENTIFIER', 'x'), ('INTEGER_LITERAL', 1))), ('BLOCK', [('PRINT', ('IDENTIFIER', 'x'))]))] | The parsed output represents a for-loop construct, consisting of the variable declaration, loop condition, loop body, and increment statement. Each construct is a tuple, with the first element indicating the type and subsequent elements providing additional information. |

| INPUT 2 | OUTPUT 2 | COMMENTS |
|---|---|---|
| let x: int = 5 * 20 + 5;<br>let y: float = 3.14;<br>let x_1: bool = true;<br>let x_2: bool = false; | Parsed program:<br><br>[('DECLARATION', 'TYPE_INT', 'x', ('PLUS', ('MUL', ('INTEGER_LITERAL', 5), ('INTEGER_LITERAL', 20)), ('INTEGER_LITERAL', 5))), ('DECLARATION', 'TYPE_FLOAT', 'y', ('FLOAT_LITERAL', 3.14)), ('DECLARATION', 'TYPE_BOOL', 'x_1', ('BOOLEAN_LITERAL', True)), ('DECLARATION', 'TYPE_BOOL', 'x_2', ('BOOLEAN_LITERAL', False))] | The parsed program represents a series of variable declarations, each with a specific data type and associated value. The output is represented as a list of tuples, with each tuple containing information about a particular declaration. The first element of the tuple indicates that it's a declaration statement, followed by the data type, variable name, and the associated value. |

| INPUT 3 | OUTPUT 3 | COMMENTS |
|---|---|---|
| fun ret(x: int, y: int) -> int {<br>  return x + y;<br>} | Parsed program:<br><br>[('FUNCTION_DEF', 'ret', [('x', 'TYPE_INT'), ('y', 'TYPE_INT')], 'TYPE_INT', ('BLOCK', [('RETURN', ('PLUS', ('IDENTIFIER', 'x'), ('IDENTIFIER', 'y')))]))] | The output corresponds to a function definition construct named 'ret', which takes two input parameters of type 'int' and returns a value of type 'int'. The function body is represented as a block construct containing a return statement that adds the two input parameters. However, it is worth noting that the arrow symbol ('->') that typically separates the function signature and return type is left out of the output. |

| INPUT 4 | OUTPUT 4 | COMMENTS |
|---|---|---|
| let x: int = 10; // This should result in an already declared error<br><br>for (let x: int = 0; x < 10; x = x + 1) {<br>  __print(x);<br>} | Error: Variable 'x' is already declared | The output is an error message indicating that a variable named 'x' has already been declared, hence violating the rule of variable uniqueness in the parser's symbol table. This error message demonstrates the parser's ability to detect and handle syntax errors during the parsing process. |

# Task 3 - AST XML Generation Pass

## Description

This section describes the implementation of a comprehensive and detailed Abstract Syntax Tree (AST) XML generator [6] for the *PixArLang Programming Language*. This is a tree-walking generator, which means it traverses the AST that was generated in the Parser code and for each node, it generates an equivalent XML representation. This process is often referred to as an AST XML Generation Pass. It turns an AST, which is a more abstract and high-level representation of the source code, into a concrete, human-readable XML format [7].

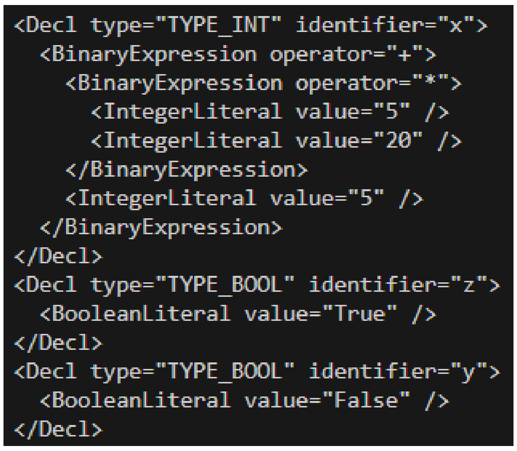
The core functionality of the XML generator is encapsulated within the *ASTXMLGenerator* class. It's constructed with an AST, and maintains an *indent_level* for pretty-printing the XML.

The main mechanism by which the generator works is by defining a visit method for each kind of node in the AST. Each visit method knows how to generate the XML for that specific kind of node.

For instance, the *visit_PROGRAM* method is invoked when the generator encounters a *'Program'* node in the AST. This method prints the opening tag *'<Program>'*, increases the indentation level, visits each child node recursively, then decreases the indentation level and prints the closing tag *'</Program>'*.

The *visit_DECLARATION* method handles *'Declaration'* nodes. It similarly prints an opening tag, but this one includes the type and identifier of the declared variable as attributes. If the declaration includes an initialization, the method recursively visits this node as well.

The *visit_INTEGER_LITERAL* and *visit_BOOLEAN_LITERAL* methods handle nodes representing literal values. These methods simply print self-closing XML tags with the literal value as an attribute.



```
<Decl type="TYPE_INT" identifier="x">
  <BinaryExpression operator="+">
    <BinaryExpression operator="*">
      <IntegerLiteral value="5" />
      <IntegerLiteral value="20" />
    </BinaryExpression>
    <IntegerLiteral value="5" />
  </BinaryExpression>
</Decl>
<Decl type="TYPE_BOOL" identifier="z">
  <BooleanLiteral value="True" />
</Decl>
<Decl type="TYPE_BOOL" identifier="y">
  <BooleanLiteral value="False" />
</Decl>
```

*Figure 3 - Example XML for Integer and Boolean*

Methods like *visit_BINARY_EXPRESSION*, *visit_PLUS*, *visit_MUL*, *etc.*, handle binary operations. They print an opening tag with the operator as an attribute, then recursively visit the left and right operands, and finally print the closing tag.

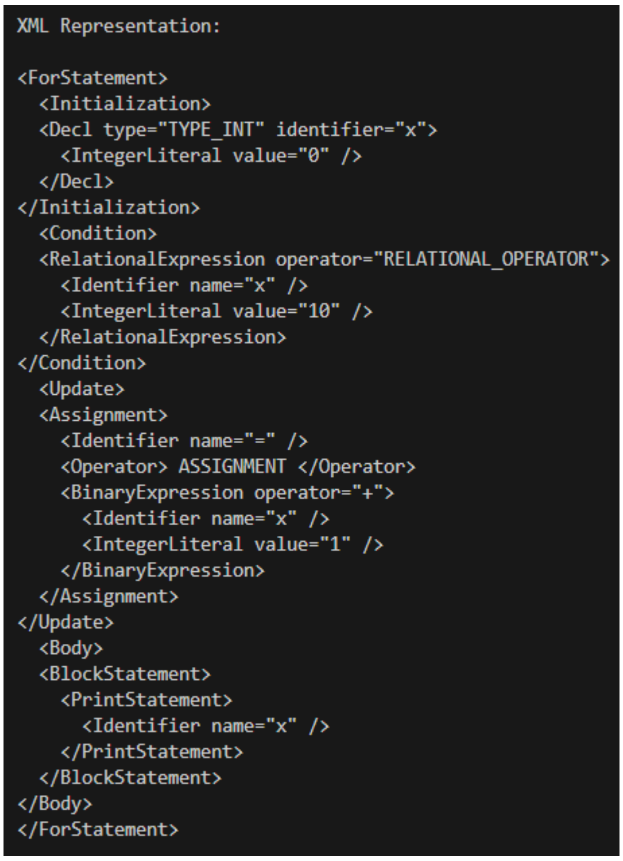
There are many other visit methods in the class for handling different kinds of nodes, such as unary operations, control flow constructs (like if statements and loops), function definitions and calls, etc. Each of these methods follows a similar structure: print the opening tag (including any relevant attributes), visit any child nodes, and print the closing tag.

In cases where a node type doesn't have a dedicated visit method, the generator falls back to the *generic_visit* method. This method raises a *NotImplementedError*, indicating that the generator doesn't know how to handle this kind of node. This is a safety feature that ensures the generator fails loudly if it encounters an unexpected kind of node.

The script concludes with a simple test of the *ASTXMLGenerator*. It creates a Lexer and Parser for a snippet of source code, generates the tokens and the AST, and then creates an instance of *ASTXMLGenerator* to generate and print the XML representation of the source code. If any error occurs during lexing or parsing, the code catches it and prints an error message.

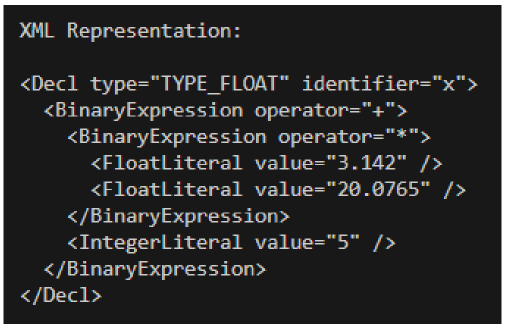
This part showcases a key phase of the process of interpreting or compiling a programming language. The AST XML Generation Pass is a crucial step that bridges the gap between the abstract syntax tree and the final Code Generation.

## Testing & Output

| INPUT 1 | OUTPUT 1 | COMMENTS |
|---|---|---|
| for (let x: int = 0; x < 10; x = x + 1) {<br>__print(x);<br>} |  | The XML output is a XML representation of a 'for' loop in PixArLang. <*ForStatement*> represents the loop structure. <*Initialization*> contains a <*Decl*> tag for declaring and initializing x to 0. <*Condition*> checks x against 10 using <*RelationalExpression*>. <*Update*> increments x via an <*Assignment*>. <*Body*> contains the operations to be performed in each iteration, in this case, a <*PrintStatement*>. Each tag corresponds to a code construct, providing a structured view of the source code. |

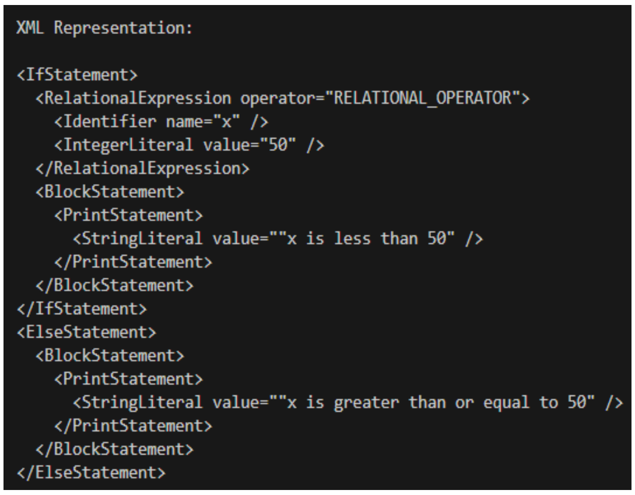
```
XML Representation:

<ForStatement>
  <Initialization>
  <Decl type="TYPE_INT" identifier="x">
    <IntegerLiteral value="0" />
  </Decl>
</Initialization>
  <Condition>
  <RelationalExpression operator="RELATIONAL_OPERATOR">
    <Identifier name="x" />
    <IntegerLiteral value="10" />
  </RelationalExpression>
</Condition>
  <Update>
  <Assignment>
    <Identifier name="=" />
    <Operator> ASSIGNMENT </Operator>
    <BinaryExpression operator="+">
      <Identifier name="x" />
      <IntegerLiteral value="1" />
    </BinaryExpression>
  </Assignment>
</Update>
  <Body>
  <BlockStatement>
    <PrintStatement>
      <Identifier name="x" />
    </PrintStatement>
  </BlockStatement>
</Body>
</ForStatement>
```

| INPUT 2 | OUTPUT 2 | COMMENTS |
|---|---|---|
| let x : float = 3.142 * 20.0765 + 5; |  | The XML output presented corresponds to the *PixArLang* source code example provided in the **assignment instructions**. It illustrates how a variable declaration and assignment are represented as XML. The *<Decl>* tag denotes a variable declaration for *x* of type TYPE_FLOAT, followed by a complex binary expression. This binary expression composed of an inner multiplication and an outer addition operation encapsulates the assignment part. |

```
XML Representation:

<Decl type="TYPE_FLOAT" identifier="x">
  <BinaryExpression operator="+">
    <BinaryExpression operator="*">
      <FloatLiteral value="3.142" />
      <FloatLiteral value="20.0765" />
    </BinaryExpression>
    <IntegerLiteral value="5" />
  </BinaryExpression>
</Decl>
```

| INPUT 3 | OUTPUT 3 | COMMENTS |
|---|---|---|
| if (x < 50) { __print("x is less than 50"); } else { __print("x is greater than or equal to 50"); } |  | The XML output provided is the AST XML representation of an 'if-else' construct in PixArLang. The *<IfStatement>* tag encapsulates the initial condition check and operation . The condition, expressed as a *<RelationalExpression>*, checks if *x* is less than 50. If this condition is true, the *<BlockStatement>* under *<IfStatement>* executes, printing "x is less than 50" via a *<PrintStatement>*. The *<ElseStatement>* denotes the alternative block of code to be executed when the if-condition is not met. The enclosed *<BlockStatement>* contains a *<PrintStatement>* to print "x is greater than or equal to 50". |

```
XML Representation:

<IfStatement>
  <RelationalExpression operator="RELATIONAL_OPERATOR">
    <Identifier name="x" />
    <IntegerLiteral value="50" />
  </RelationalExpression>
  <BlockStatement>
    <PrintStatement>
      <StringLiteral value=""x is less than 50" />
    </PrintStatement>
  </BlockStatement>
</IfStatement>
<ElseStatement>
  <BlockStatement>
    <PrintStatement>
      <StringLiteral value=""x is greater than or equal to 50" />
    </PrintStatement>
  </BlockStatement>
</ElseStatement>
```

# Task 4 - Semantic Analysis Pass

## Description

This section provides a semantic analyser [8] for the *PixArLang Programming Language*. The analyser is used to check the semantics of a source code, ensuring that it follows the rules of the language, and to detect semantic errors that are not caught at the lexical or syntactic levels.

The code is organized into two main classes: *SymbolTable* and *SemanticAnalyzer*. The *SymbolTable* class is used to manage variable declarations and their associated types within the source code. The table uses the concept of "scopes" to mimic the behaviour of real programming languages: a new scope is opened every time a new block (like a function or a loop) is entered, and closed when the block is exited. This class includes methods for adding and looking up variables, and for entering, exiting, pushing, and popping scopes.

The SemanticAnalyzer class is where the main semantic analysis occurs. The analyser works by walking the abstract syntax tree (AST) of the source code, visiting each node and checking that it follows the semantic rules of the language. The analyser uses the visitor pattern [9], where each node type has a corresponding "visit" method that performs the specific semantic checks for that node type.

For example, the *visit_DECLARATION* method checks that a variable is not already declared in the current scope, the *visit_ASSIGNMENT* method checks that a variable has been declared before it is assigned, and the visit_PLUS method checks that both operands of a plus operation are integers. Many other types of nodes and checks are implemented.

At the end of the code, a sample usage of the SemanticAnalyzer class is shown. The input source code is passed through the lexer, parsed into an AST, and then semantically analysed. If no errors are found, a success message is printed; if any errors occur at any stage of the process, they are caught and printed.

## Testing & Output

| INPUT 1 | OUTPUT 1 | COMMENTS |
|---|---|---|
| __pixelr(0, 0, 255, 300, 0);<br>// This should result in an error | Error: Incompatible argument types for PIXEL_STATEMENT: ['int', 'int', 'int', 'int', 'int'], expected: ['int', 'int', 'int', 'int', 'colour'] | In this case, the output message indicates an error detected during the semantic analysis phase.<br>The error is triggered when the *__pixelr* function, which presumably expects five arguments of types: integer, integer, integer, integer, and a colour type, receives five integer arguments instead. The last argument should have been of type 'colour' according to the *PixArLang* programming language's rules. |

| INPUT 2 | OUTPUT 2 | COMMENTS |
|---|---|---|
| let x: int = 5 * 20 + 5;<br>let y: float = 3.14;<br>let z: bool = true;<br>// This is semantically correct | Semantic analysis completed successfully! | This output message "Semantic analysis completed successfully!" indicates that the input code has passed the semantic analysis phase without any errors. It confirms that the code adheres to the semantic rules of the *PixArLang* programming language. |

| INPUT 3 | OUTPUT 3 | COMMENTS |
|---|---|---|
| while(x>10){<br>    __print(x);<br>    x++;<br>} // This will result in an error since x is not declared before | Error: Variable 'x' not declared. | The input code contains a while loop which uses a condition (x>10) and a print statement (__print(x)) inside its body. Both of these lines refer to a variable 'x' that has not been previously declared in the code, hence violating the rules of the *PixArLang* programming language. |

| INPUT 4 | OUTPUT 4 | COMMENTS |
|---|---|---|
| fun ret(x: int) -> int {<br>    for(let y : int = 0; y < x; y = y + 1){<br>    x = x+y;<br>    }<br>    return x;<br>} | Error: Variable 'y' is already declared in the current scope. | The input code is a function definition with a for loop inside it. In the initialization of the loop, a variable 'y' is declared and initialized to 0. However, during the loop's execution, there's an attempt to redeclare 'y' in the same scope, which is not allowed in *PixArLang*. |

| INPUT 5 | OUTPUT 5 | COMMENTS |
|---|---|---|
| let x: int = 5;<br>let y: float = 3.14;<br><br>if (x < 10) {<br>  __print(x+y); // This should be an error since y is a float so can’t be added to x.<br>} | Error: Operands of '+' operator must be of type 'int'. | The input script declares two variables: 'x' of type 'int', and 'y' of type 'float'. Subsequently, it attempts to execute an 'if' statement where the condition checks if 'x' is less than 10. Inside this 'if' block, it tries to print the sum of 'x' and 'y'. However, the '+' operation in *PixArLang* only accepts operands of the same type, specifically 'int' in this case. |

## Task 5 - PixIR Code Generation Pass

## Description

The final stage of the *PixArLang* Compiler is the Code Generation Pass. The Code Generator takes the Abstract Syntax Tree (AST) produced by the Parser and translates it into the target language code. In this case, the target language is *PixIR*, an intermediary language that is designed to be executed on the *PAD2000* hardware

The Code Generation process is performed by a class named *PixIRCodeGenerator*. Similar to the AST XML Generator and Semantic Analyzer, the *PixIRCodeGenerator* also uses the visitor pattern [9]. Each kind of node in the AST has a corresponding *"visit"* method in the *PixIRCodeGenerator*, which translates that node into the corresponding PixIR code.

In essence, each type of *PixArLang* AST node has a corresponding *"visit"* method in the *PixIRCodeGenerator*, which translates the *PixArLang* construct into its *PixIR* equivalent, thereby aiding in the conversion of the entire program.

For example, the *visit_DECLARATION* method translates a variable declaration in PixArLang into a memory allocation and assignment in *PixIR*, initializing the variable to a default value based on its type. The *visit_ASSIGNMENT* method translates an assignment statement in *PixArLang* into an equivalent storage operation in *PixIR*. Arithmetic and logical expressions, control flow statements, function definitions, and other language constructs are handled in a similar manner.

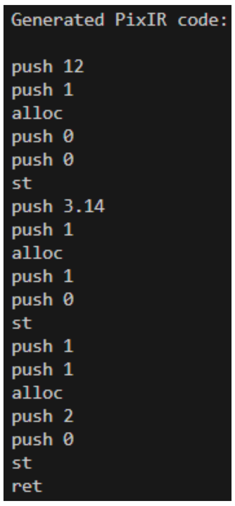
The *PixIRCodeGenerator* class also maintains a frame offset, which helps in tracking the memory location for each variable in the *PixArLang* program. This is used to properly store and load values from memory in the generated *PixIR* code.

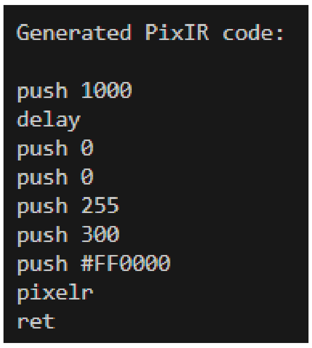
The generated *PixIR* code is stored as a list of strings, which can be joined together to form the complete program. This code is built up incrementally as the *PixIRCodeGenerator* traverses the AST, with each visit method appending the appropriate *PixIR* command to the list.

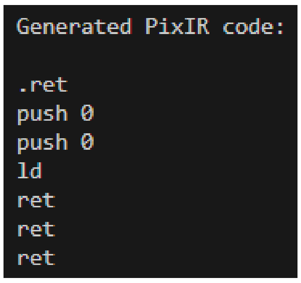
At the end of the Code Generation process, the *PixIRCodeGenerator* has a complete *PixIR* program that is equivalent to the input *PixArLang* program. This *PixIR* program can then be executed to produce the desired output.

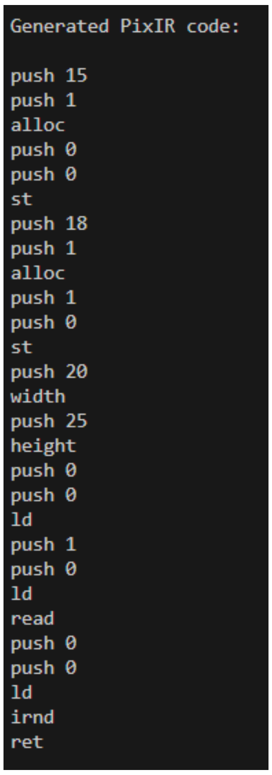
It is worth noting that not all *PixArLang* constructs are supported for Task 5. In particular, the *for loop* and *else statement* were not implemented in this task.

## Testing & Output

| INPUT 1 | OUTPUT 1 | COMMENTS |
|---|---|---|
| let x: int = 12;<br>let y: float = 3.14;<br>let z: bool = true; | <br>Generated PixIR code:<br><br>push 12<br>push 1<br>alloc<br>push 0<br>push 0<br>st<br>push 3.14<br>push 1<br>alloc<br>push 1<br>push 0<br>st<br>push 1<br>push 1<br>alloc<br>push 2<br>push 0<br>st<br>ret | **push 12, push 1, alloc, push 0, st**: These lines initialize an integer x with the value 12, allocate space on the stack, and store the value at the specified memory location.<br><br>**push 3.14, push 1, alloc, push 1, st**: Similarly, these lines initialize a float y with the value 3.14 and store it.<br><br>**push 1, push 1, alloc, push 2, st**: These lines initialize a boolean z as true (represented by 1), allocate space, and store it.<br><br>**ret**: This signals the end of the code. |

| INPUT 2 | OUTPUT 2 | COMMENTS |
|---|---|---|
| __delay(1000);<br><br>__pixelr(0, 0, 255, 300, #FF0000); | <br>Generated PixIR code:<br><br>push 1000<br>delay<br>push 0<br>push 0<br>push 255<br>push 300<br>push #FF0000<br>pixelr<br>ret | **push 1000, delay:** These lines represent a delay function, pausing execution for 1000 milliseconds.<br><br>**push 0, push 0, push 255, push 300, push #FF0000, pixelr:** These commands draw a pixel at coordinates (0,0) with a color intensity of 255, radius of 300, and end color of #FF0000.<br><br>**ret:** This signals the end of the code. |

| INPUT 3 | OUTPUT 3 | COMMENTS |
|---|---|---|
| fun ret(x: int, y: int) -> int {<br>  return x;<br>} | <br>Generated PixIR code:<br><br>.ret<br>push 0<br>push 0<br>ld<br>ret<br>ret<br>ret | **.ret:** This line marks the start of the function ret.<br><br>**push 0, ld:** These lines load the value of the first parameter x from the stack (at offset 0).<br><br>**ret:** This returns the value of x from the function.<br><br>**ret, ret:** These lines indicate the end of the function and the end of the code. The repetition may indicate a minor bug in the code generator as one ret should suffice. |

| INPUT 4 | OUTPUT 4 | COMMENTS |
|---|---|---|
| let x: int = 15;<br>let y: int = 18;<br><br>__width(20);<br>__height(25);<br>__read(x,y);<br>__randi(x); | <br>`Generated PixIR code:`<br><br>`push 15`<br>`push 1`<br>`alloc`<br>`push 0`<br>`push 0`<br>`st`<br>`push 18`<br>`push 1`<br>`alloc`<br>`push 1`<br>`push 0`<br>`st`<br>`push 20`<br>`width`<br>`push 25`<br>`height`<br>`push 0`<br>`push 0`<br>`ld`<br>`push 1`<br>`push 0`<br>`ld`<br>`read`<br>`push 0`<br>`push 0`<br>`ld`<br>`irnd`<br>`ret` | **push 15, push 1, alloc, push 0, st:** These lines initialize an integer x with the value 15 and store it on the stack.<br><br>**push 18, push 1, alloc, push 1, st:** Similarly, these lines initialize an integer y with the value 18 and store it on the stack.<br><br>**push 20, width:** These lines set the width of the canvas to 20.<br><br>**push 25, height:** These lines set the height of the canvas to 25.<br><br>**push 0, ld, push 1, ld, read:** These lines read the pixel at coordinates (x,y).<br><br>**push 0, ld, irnd:** These lines generate a random integer up to x.<br><br>**ret:** This signals the end of the code. |

## Conclusion

In conclusion, the implemented *PixArLang* Compiler is a tool that converts *PixArLang* source code into the *PixIR* intermediary language. It does this by following five important steps: Lexical Analysis, Syntactic Analysis, AST XML Generation, Semantic Analysis, and PixIR Code Generation. These steps ensure that the code is accurately understood and follows the rules of the language. This documentation has presented a detailed insight into each of these stages, their purpose, and their function within the larger compilation process.

## Video

The link provided below directs you to a video presentation of the *PixArLang* Compiler. This video serves as a comprehensive guide, showcasing the step-by-step process of the compiler code. Should you encounter any difficulties accessing the video, please don't hesitate to reach out to me.

**Video Link**

https://drive.google.com/file/d/1NBFLnUFLuJ6ggcT0QfMdTH22ecggM2HD/view?usp=share_link

# References

[1] "Compiler design," Python Developer’s Guide. [Online]. Available: https://devguide.python.org/internals/compiler/. [Accessed: April 22, 2023].

[2] Franleplant, "Franleplant/lexer-py: Direct coded and table driven automatas for lexing a pseudo lisp," GitHub. [Online]. Available: https://github.com/franleplant/lexer-py. [Accessed: April 22, 2023].

[3] "Compiler design LL(1) parser in Python," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/compiler-design-ll1-parser-in-python/. [Accessed: April 27, 2023].

[4] "Guide to the Parser," Python Developer’s Guide. [Online]. Available: https://devguide.python.org/internals/parser/. [Accessed: April 27, 2023].

[5] "Compiler design - variants of syntax Tree," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/compiler-design-variants-of-syntax-tree/. [Accessed: April 28, 2023].

[6] S. Howson, "Python XML tutorial: Element tree parse & read," DataCamp. [Online]. Available: https://www.datacamp.com/tutorial/python-xml-elementtree. [Accessed: May 1, 2023].

[7] "How to convert XML to Python AST," Stack Overflow. [Online]. Available: https://stackoverflow.com/questions/51431048/how-to-convert-xml-to-python-ast. [Accessed: May 1, 2023].

[8] "Semantic Analysis in compiler design," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/. [Accessed: May 1, 2023].

[9] "Visitor method - python design patterns," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/visitor-method-python-design-patterns/. [Accessed: May 3, 2023].