Text

Description automatically generated with medium confidence

CPS2000

Compiler Theory and Practice

Course Assignment 2022/2023

Programmed using Python

By Mark Dingli

20703H

**Table of contents**

Statement of Completion …………………………………………………………………………………… 3

Introduction …………………..…………………………………………………………………………………… 4

Creating and Outputting sets X,Y,Z ……………………………………………………………………… 5

Determining the Intersection ……………………………………………………………………………… 6

AVL Tree ..…………………..………………………………………………………………………………………. 7

Red-Black Tree ……………..……………………………………………………………………….…………. 10

Binary Search Tree (Unbalanced) .…………………………………………………………………….. 13

Inserting into Trees Statistics .…..………………………………………………………………………. 16

Deleting in Trees Statistics …….………………………………………………………………….………. 18

Searching in Trees Statistics .…..…………………….………………………………………….………. 20

AVL Trees VS Red-Black Trees .………………………………………………………………….………. 22

Conclusion ……………….…..…………………………………………………………………………………… 23

References ……………….…..………………………………………………………..………………………… 24

**Introduction**

**Task 1 - Table-driven Lexer**

**Description**

The code you provided is a lexer for the PixArLang language, which is implemented using the table-driven approach. The lexer is responsible for scanning the input program and identifying the tokens in it. It can also report any lexical errors in the input program.

The PixArLang micro-syntax has been determined prior to implementing the lexer. This micro-syntax defines the tokens that the lexer needs to identify. The EBNF for the language has also been used to determine the selection of tokens.

To implement the lexer, a transition table or DFA encoding has been hard-coded as a 2D array in the code. This transition table simulates the DFA transition function of the PixArLang micro-syntax.

The lexer starts by reading the input program character by character. For each character, the lexer looks up the corresponding column in the transition table based on the current state of the DFA. The value in the table at the intersection of the current state and the column index gives the next state of the DFA.

If the next state is an accepting state, the lexer emits the corresponding token and resets the DFA to the initial state. If the next state is an error state, the lexer reports a lexical error and terminates.

The implementation of the transition table allows for efficient scanning of the input program as the DFA is simulated using a simple lookup in the table, rather than executing a series of complex state transitions. This makes the lexer faster and more reliable.

In summary, your code implements a lexer for the PixArLang language using the table-driven approach. It reads the input program character by character and identifies the tokens in it using a hard-coded transition table. The implementation of the transition table allows for efficient and reliable scanning of the input program.

A transition table: The transition table is a dictionary that represents the finite state machine used to recognize tokens in the language. The keys in this dictionary are pairs consisting of the current state and the current character, and the values are the new states. There are different states for recognizing different types of tokens, such as identifiers, integer literals, operators, delimiters, and others. The transition table is built in the build\_transition\_table method, which sets the transitions for all states and input characters according to the rules of the language.

The get\_token method: This method is responsible for reading the source code character by character and using the transition table to recognize tokens. It keeps track of the current state, the current 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 there is no transition for the current state and character, the method checks if the current lexeme forms a valid token and, if so, returns it. If the current lexeme does not form a valid token, an error is raised.

The Lexer class also includes other helper methods. The get\_next\_char method is responsible for reading the next character from the source code, handling escape sequences in string literals. The get\_token\_type\_from\_state method is used to determine the type of token that has been recognized, based on the current state and lexeme.

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

**Description**

This part describes the implementation of a hand-crafted LL(k) parser [1][3], 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, an 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) [2] 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.

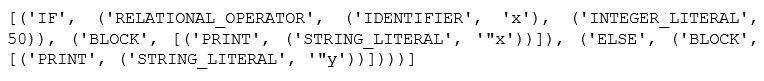


Figure 1 - AST Output of an IF-ELSE Statement

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

A screen shot of a computer program

Description automatically generated with medium confidence

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) {  \_\_print(x);  } | Parsed program:  [('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;  let y: float = 3.14;  let x\_1: bool = true;  let x\_2: bool = false; | Parsed program:  [('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 {  return x + y;  } | Parsed program:  [('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 not present in the output. |

|  |  |  |
| --- | --- | --- |
| **INPUT 4** | **OUTPUT 4** | **COMMENTS** |
| let x: int = 10; // This should result in an already defined error  for (let x: int = 0; x < 10; x = x + 1) {  \_\_print(x);  } | 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 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.

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.

A picture containing text, font, screenshot

Description automatically generated

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) {  \_\_print(x);  } |  | 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. |

|  |  |  |
| --- | --- | --- |
| **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. |

|  |  |  |
| --- | --- | --- |
| **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". |

**Task 4 - Semantic Analysis Pass**

**Description**

This section provides a semantic analyser 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, 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.

|  |  |  |
| --- | --- | --- |
| **INPUT 1** | **OUTPUT 1** | **COMMENTS** |
| for (let x: int = 0; x < 10; x = x + 1) {  \_\_print(x);  } | Parsed program:  [('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. |

**References**

[1] https://www.geeksforgeeks.org/compiler-design-ll1-parser-in-python/

[2] https://www.geeksforgeeks.org/compiler-design-variants-of-syntax-tree/

[3] https://devguide.python.org/internals/parser/

[4]

[5]

[6]

[7]

[8]

[9]

[10]