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.

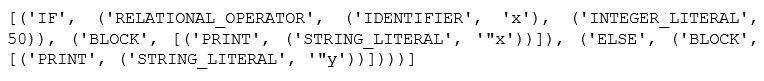


Figure 1 - AST Output of an IF-ELSE Statement

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 | OUTPUT | COMMENTS |
|  |  |  |

**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]