DOMAIN SPECIFIC LANGUAGE FOR DATA STRUCTURE MANIPULATION

CĂLUGĂREANU Ana, CHICHIOI Iuliana, REABCIUC Daria-Brianna*

Department of Software Engineering and Automation, FAF-221, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chisinau, Republic of Moldova

*Corresponding author: Reabciuc Daria-Brianna, daria.brianna.reabciuc@gmail.com

Tutor/coordinator: **Irina COJUHARI,** conf. univ., dr., Technical University of Moldova

Abstract. *This paper presents ManipulaPy, a new Domain Specific Language. The domain specific language that is being developed for the purpose of manipulating data structures is specifically designed to meet the needs of different software development industries. The document provides users with a comprehensive grasp of the operation of the framework by explaining the grammatical and syntactical subtleties of ManipulaPy as well as the details of its implementation. Furthermore, the research presents interesting possibilities for ManipulaPy's improvement and future development. The intention is to develop a language with strong abstractions, easy-tounderstand syntax, and effective primitives for data manipulation.*

Keywords: *analysis, data structures, domain-specific language, grammar, syntax, parser.*

Introduction

Data structures play a fundamental role in organizing and managing data efficiently in software development. Among these, linear data structures hold particular significance due to their sequential arrangement of elements. The project will involve comprehensive domain analysis to understand the requirements and operations needed for numerical data manipulation within linear data structures. Following this, a grammar will be created to define the syntax, ensuring clarity and conciseness in expressing data manipulation tasks.

Through this Domain Specific Language (DSL), the aim is to provide developers with a tool for efficiently manipulating numerical data within linear data structures, thereby enhancing code quality and facilitating software development in various domains.

Domain Analysis

Data structures are fundamental components in computer science that enable efficient organization, storage, and manipulation of data. They provide a way to represent complex data in a structured manner, allowing for easy access and modification [1], as shown in Fig. 1. The importance of data structures lies in their ability to optimize various operations on data, such as searching, sorting, and retrieving. By choosing the right data structure for a given problem, developers can significantly improve the performance and efficiency of their software. In this project, the goal is to harness the power of data structures to provide users with a simple yet powerful tool for manipulating linear data [2].

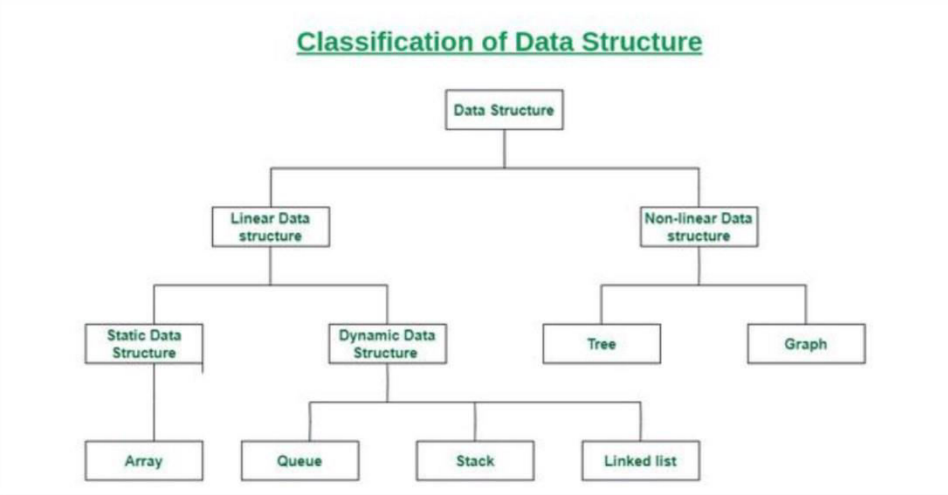

Figure 1. Representation of Classification of Data Structures

Array

An array is a collection of items of the same data type stored at contiguous memory locations. It is characterized by having homogeneous elements, meaning all elements within an array must be of the same data type. In most programming languages, elements in an array are stored in contiguous (adjacent) memory locations, fig. 2 [1].

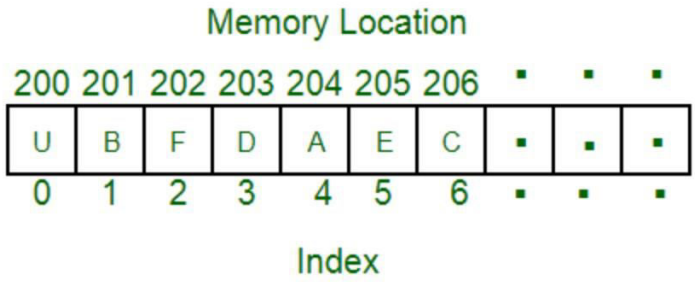

Figure 2. Representation of Array

Linkes List

A Linked List is a linear data structure consisting of a chain of nodes, each containing data and a reference to the next node. Unlike arrays, linked list elements are not stored at contiguous memory locations.

Each element in a linked list, or node, contains two components: the actual data or value associated with the element, and a reference or pointer to the next node in the linked list, fig. 3 [1].

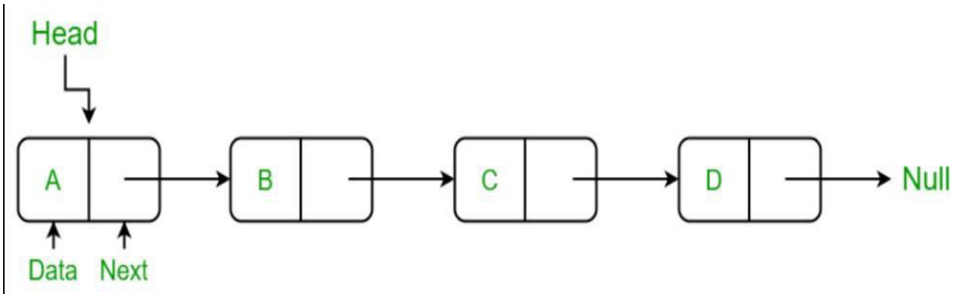

Figure 3. Representation of Linked List

Stack Data Structure

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It can be of two types: fixed-size stack and dynamic-size stack. In a fixed-size stack, the size is predetermined and cannot be changed during runtime, while a dynamic-size stack can grow or shrink dynamically as elements are added or removed, fig. 4 [1].

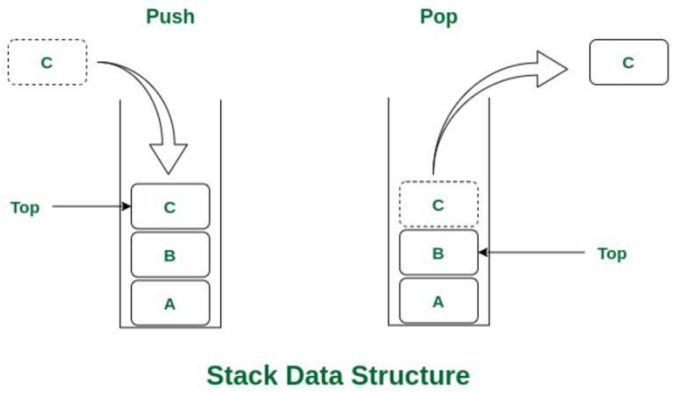

Figure 4. Representation of Stack Data Structure

Grammar

To establish the framework of DSL aimed at facilitating table manipulation within a Python environment, a detailed reference grammar is outlined. This grammar delineates the structural composition of the language, dictating the assembly of statements via the utilization of reserved keywords, data types, and previously articulated syntax. The grammar is articulated using BNF, a formal notation system employed for describing language syntax with precision and clarity [3].

The grammar consists of various production rules, each defining a symbol in relation to other symbols and literals. Non-terminal symbols, indicated by <symbol>, can be decomposed into sequences comprising terminal symbols (keywords and literals) and additional non-terminal symbols. Terminal symbols are identified by lowercase notation for keywords or designated symbols (e.g., (,), *) [4].

grammar DataStructureDSL;

// Parser rules program: statement*; statement: arrayStatement | linkedListStatement | stackStatement | queueStatement; arrayStatement: 'ARRAY' '[' INT (',' INT)* ']' (insertArray | deleteArray | searchArray | sortArray)? ';';

linkedListStatement: 'LINKEDLIST' (insertLinkedList | deleteLinkedList | searchLinkedList)? ';'; stackStatement: 'STACK' (pushStack | popStack | topStack | isEmptyStack)? ';'; queueStatement: 'QUEUE' (enqueueQueue | dequeueQueue | peekQueue | isFullQueue | isNullQueue)? ';'; // Array operations insertArray: 'INSERT' '[' INT ']' 'INTO' 'ARRAY' '[' INT ']' ';'; deleteArray: 'DELETE' 'FROM' 'ARRAY' '[' INT ']' ';'; searchArray: 'SEARCH' 'ARRAY' '[' INT ']' 'FOR' INT ';'; sortArray: 'SORT' 'ARRAY' '[' INT ']' ('ASCENDING' | 'DESCENDING') ';';

// Corrected ascending/descending

// Linked list operations insertLinkedList: 'INSERT' 'INTO' 'LINKEDLIST' '[' INT ']' 'VALUE' INT ';'; deleteLinkedList: 'DELETE' 'FROM' 'LINKEDLIST' '[' INT ']' ';'; searchLinkedList: 'SEARCH' 'LINKEDLIST' '[' INT ']' 'FOR' INT ';';

// Stack operations pushStack: 'PUSH' INT 'TO' 'STACK' ';'; popStack: 'POP' 'FROM' 'STACK' ';'; topStack: 'TOP' 'ELEMENT' 'OF' 'STACK' ';'; isEmptyStack: 'CHECK' 'IF' 'STACK' 'IS' 'EMPTY' ';';

// Queue operations enqueueQueue: 'ENQUEUE' INT 'TO' 'QUEUE' ';'; dequeueQueue: 'DEQUEUE' 'FROM' 'QUEUE' ';'; peekQueue: 'PEEK' 'FRONT' 'ELEMENT' 'OF' 'QUEUE' ';'; isFullQueue: 'CHECK' 'IF' 'QUEUE' 'IS' 'FULL' ';'; isNullQueue: 'CHECK' 'IF' 'QUEUE' 'IS' 'EMPTY' ';';

This reference grammar methodically outlines the DSL's syntax, elucidating the construction of statements pertinent to data structure manipulation. Each rule encapsulates distinct facets of the language, from the articulation of data structures and operations to the processes of data insertion, deletion, and querying.

Lexical Consideration

When designing the lexical elements of the DSL grammar, it's important to consider the following aspects:

Keywords: The DSL uses keywords such as array, linked list, stack and queue to define different data structures and operations. Ensure these keywords are clearly defined and reserved for their specific purposes.

Identifiers: Define rules for identifiers, such as variable names or labels within the DSL. These rules should specify valid characters and any naming conventions that need to be followed.

Literals: Determine the types of literals supported in the DSL, such as integers represented by the INT rule. Ensure that the grammar adequately handles these literals.

Whitespace Handling: Specify rules for handling whitespace characters like spaces, tabs, and newlines. In the provided grammar, whitespace is ignored using the WS rule.

Comments: Define rules for comments to enhance readability and allow users to add explanatory notes. In this DSL, comments are defined using the COMMENT rule.

Special Symbols: Identify and define rules for special symbols or punctuation marks used in the DSL syntax, such as brackets, commas, or semicolons.

Error Handling: Consider how errors and invalid input should be handled in the DSL. Define rules for reporting errors and providing meaningful feedback to users.

Reserved Words: Determine if there are any words that should be reserved and cannot be used as identifiers. These may include language keywords or future extensions.

Parsing

Parsing is a crucial process in understanding and interpreting the DSL grammar for data structure manipulation. It involves several key components that work together to transform raw code into a structured representation.

After Lexical Analysis, Syntax Analysis takes place, performed by the Parser. This phase involves analyzing the token stream produced by the lexer against the grammar rules of the DSL. The Parser ensures syntactic correctness and constructs a hierarchical structure known as a parse tree or AST. This structure captures the syntactic organization of the program and its nested relationships [5].

Lexical Analysis: The lexer converts the code into tokens:

Keyword('create'), Identifier('array'), Identifier('myArray'), Keyword('of'Identifier('size'), Number('5'), Semicolon(';'),
Keyword('insert'), Number('10'), Keyword('into'), Identifier('array'),
Identifier('myArray'), Keyword('at'), Keyword('index'), Number('2'), Semicolon(';'),
Keyword('delete'), Keyword('from'), Identifier('array'), Identifier('myArray'),
Keyword('at'), Keyword('index'), Number('3'), Semicolon(';'),
Keyword('search'), Identifier('array'), Identifier('myArray'), Keyword('for'),
Keyword('value'), Number('8'), Semicolon(';')

Consider the following code snippet in our data structure DSL: create array myArray of size 5; insert 10 into array myArray at index 2; delete from array myArray at index 3;

search array myArray for value 8;

Lexical Analysis: The lexer dissects the code into tokens:

Keyword('array'), LeftSquareBracket('['), Number('5'), Comma(','), Number('10'), Comma(','), Number('15'), Comma(','), Number('20'), RightSquareBracket(']'), Keyword('search'), Keyword('array'), LeftSquareBracket('['), Number('3'), RightSquareBracket(']'), Keyword('for'), Number('15'), Semicolon(';')

Consider the given code snippet within our data structure DSL: ARRAY [5, 10, 15, 20] SEARCH ARRAY [3] FOR 15; Lexical Analysis:

The lexer tokenizes the code as follows:

Keyword('array'), LeftSquareBracket('['), Number('5'), Comma(','), Number('10'), Comma(','), Number('15'), Comma(','), Number('20'), RightSquareBracket(']'), Keyword('search'), Keyword('array'), LeftSquareBracket('['), Number('3'), RightSquareBracket(']'), Keyword('for'), Number('15'), Semicolon(';') Syntax Analysis:

The parser examines the token stream against the grammar rules and constructs an AST [6]. A simplified representation is as follows:

Program

[ArrayDeclaration, [Number: '5', Number: '10', Number: '15', Number: '20']] [SearchInArray, [Index: Number: '3', Value: Number: '15']]

The root of the tree, fig. 5, is a Program node, with each statement represented as a child node. This parsing example illustrates how the DSL code is parsed and understood by the computer, transitioning from a sequence of tokens to a structured representation that captures the hierarchical nature of data structure manipulation operations. The AST represents the syntactic structure of the DSL code, facilitating further analysis, interpretation, and execution of the program.

Figure 5. Representation of Parsing Tree

Conclusions

The domain analysis provides a comprehensive understanding of the challenges inherent in software development, particularly concerning the manipulation and presentation of data structures. It highlights the critical role played by specialized tools in addressing issues such as efficiency, scalability, complexity, flexibility, and correctness. The analysis reveals the limitations of conventional programming languages in providing built-in support for efficient data manipulation, leading to cumbersome and error-prone code that impacts developers' productivity and software quality.

In response to these challenges, the proposed DSL for data structure manipulation emerges as a promising solution. By offering specialized constructs tailored to the manipulation of numerical data, the DSL aims to streamline common data manipulation tasks, empowering developers to focus on core application logic and functionality. Through intuitive syntax, powerful abstractions, and graphical representations, the DSL seeks to revolutionize the way developers interact with and manage data structures, fostering a culture of innovation and collaboration across diverse domains within the software development landscape.

References:

[1] Data Structure Types, Classifications and Applications, [online]. [accessed 27.03.2024]. Available: https://www.geeksforgeeks.org/what-is-data-structure-types-classifications-andapplications/
[2] Lafore, R. (2002) Data Structures and Algorithms in Java. Sams Publishing.
[3] Parr, T. (2010). Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. Pragmatic Bookshelf.
[4] Brown, E., Miller, D. (2020). Design and Implementation of a Domain-Specific Language for Data Structures Manipulation. Proceedings of the International Conference on Software Engineering, 2020.
[5] Grune, D., & Jacobs, C. J. H. (2008). Parsing Techniques: A Practical Guide. Springer Science & Business Media.
[6] Johnson, A., Williams, B. (2019). A Comparative Study of Domain-Specific Languages for Data Structures Manipulation. ACM Transactions on Programming Languages and Systems, 2019.