TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Blockly-C: A Web application for block-based C programming

Konstantinos Danopoulos

Thesis Committee

Professor Katerina Mania (Supervisor)

Professor Antonios Deligiannakis

Professor Vasileios Samoladas

Chania, February 2024

Abstract

Blockly-C is an online platform designed as a visual-based educational resource for those learning the C programming language. Leveraging a 'recognition over recall' methodology, it simplifies the intricate aspects of the C language, directing learners to concentrate on the underlying logic rather than getting bogged down by its syntax. Beginners participating in foundational programming courses might find Blockly-C to be a more efficient learning tool compared to conventional C programming methods. The aim of this dissertation is to discuss the conception, development, and assessment of Blockly-C. It's crafted to streamline the C language learning process, incorporating visual programming and an incremental display of program operations and memory configurations. The overarching objective is to offer beginners a captivating and efficient medium to learn C, eliminating syntax-related hurdles. Furthermore, Blockly-C paves the way for learners to traverse smoothly from visual code representations to the traditional text-based format as they gain proficiency in C programming.

Acknowledgments

This thesis marks the culmination of an educational journey that has been both challenging and enriching. It would not have been possible without the support and guidance of several key individuals, to whom I owe my heartfelt gratitude.

First and foremost, I extend my deepest appreciation to Nektarios Moumoutzis, my laboratory teacher, whose invaluable assistance and mentorship have been a constant source of inspiration throughout this project. Mr. Moumoutzis's expertise, patience, and encouragement have significantly shaped my academic journey, providing me with the tools and confidence necessary to navigate the complexities of this research. His unwavering support has been instrumental in my growth and success, not only in this thesis but throughout my years of study.

I am also profoundly grateful to Mrs. Mania Aikaterini, my supervisor professor, who has been a guiding light in this endeavor. Her expert supervision and insightful feedback have been crucial in bringing this thesis to fruition. Her dedication to academic excellence and her commitment to nurturing my research skills have left an indelible mark on my educational experience. I am truly thankful for her guidance, patience, and belief in my potential.

Last but not least, I would like to thank my family and friends for their endless love, support, and encouragement. Their unwavering belief in my abilities has been a source of strength and motivation, pushing me to strive for excellence.

Contents

1 Introduction **10**
- 1.1 Motivation . . . 10
- 1.2 Brief Description . . . 10
- 1.3 Structure of the Thesis . . . 11

2 Background/Research Overview **14**
- 2.1 Introduction . . . 14
- 2.2 Related Work . . . 15
  - 2.2.1 Block-c . . . 15
  - 2.2.2 C programming environment . . . 17
  - 2.2.3 SeeC . . . 18
  - 2.2.4 PlayVisualizerC . . . 22
- 2.3 Technological Background and Definitions . . . 25
  - 2.3.1 Blockly . . . 25
  - 2.3.2 Jison . . . 27
  - 2.3.3 Python Tutor . . . 29

3 End User Experience **31**
- 3.1 Introduction . . . 31
- 3.2 Block-to-C Code Conversion . . . 32
- 3.3 C-to-Block Representation . . . 48
- 3.4 Code Execution and Memory Visualization . . . 50

4 Implementation **62**
- 4.1 Introduction . . . 62
- 4.2 Block-to-C Code Conversion . . . 62
- 4.3 C-to-Block Representation . . . 93
- 4.4 Code Execution and Memory Visualization . . . 155

5 Evaluation **159**
5.1 Introduction 159
5.2 Evaluation Methodology and Data Collection 159
5.3 Analysis of Questionnaire Results 161

6 Conclusion and future work **176**
6.1 Introduction 176
6.2 Conclusions 176
6.3 Future Work 178

References **179**
.1 Appendices 182

List of Figures

2.1 A illustration of graphic environment of the Block-C. 16
2.2 An illustration of graphic environment of Abe and Yuki. . . . 18
2.3 An illustration of debugging tool SeeC. 20
2.4 An illustration of PlayVisualizerC. 23
2.5 An illustration of Blockly. 26
2.6 Jison logo. 28
2.7 An illustration of PythonTutor. 29

3.1 A detailed illustration of graphic environment of the tool. . . . 33
3.2 Initial view upon opening the web application. 34
3.3 A figure illustrating the categories sidebar, with a focused view on the blocks within the 'Loops' category. 35
3.4 Connector Types in Blockly-C. 36
3.5 Highlighted Connections in Blockly-C. 37
3.6 Text Input Fields on Blocks. 38
3.7 Drop-down menu on Blocks. 39
3.8 Interactive Features of Blockly-C UI: Code Representation and Data Type Handling 41
3.9 Mutational stages of "If" block. 43
3.10 Mutability Demonstrated: The 'Variable Declare' Block with Multiple Variable Declarations in single line of code. 44
3.11 How the user deletes a block. It's noticeable that the trash bin lid opens, signaling its responsiveness to the user's action. 44
3.12 Context Menu Options Displayed for a Variable Declaration Block in Blockly-C. 45
3.13 Real-time C Code Generation Display in the 'Code' Component. 46
3.14 Total view of real-time C Code Generation Display in the 'Code' Component. 47
3.15 Top bar of Blockly-C. 48

3.16 Example of Error Message Prompted by a Faulty .c File Upload. 49
3.17 Visualization Page Interface Overview. 51
3.18 Code Pane Overview. 52
3.19 Output Pane Overview. 54
3.20 Visualization Pane Overview. 55
3.21 Presentation of scalar variables in Visualization Pane. 56
3.22 Presentation of arrays in Visualization Pane. 56
3.23 Presentation of pointers in Visualization Pane. 57
3.24 Presentation of structures in Visualization Pane. 58
3.25 Presentation of dynamic memory allocations in Visualization Pane. 59
3.26 Presentation of functions in Visualization Pane. 60

4.1 Illustration of Input Types in Blockly with Corresponding JavaScript Code and Visual Blocks. 66
4.2 Illustration of Field Types in Blockly with Corresponding JavaScript Code and Visual Blocks. 68
4.3 Illustration of Connection Types in Blockly with Corresponding JavaScript Code and Visual Blocks. 70
4.4 Illustration of code for block "variable_declare". 73
4.5 Illustration of code for block "controls_for_2". 75
4.6 Illustration of code for block "library_stdio_printf" (part1). . 77
4.7 Illustration of code for block "library_stdio_printf" (part2). . 78
4.8 Illustration of code for block "library_stdio_printf" (part3). . 79
4.9 Illustration of "variables_declare_2" block (part 1). 80
4.10 Illustration of "variables_declare_2" block (part 2). 81
4.11 Illustration of "variables_declare_2" block (part 3). 82
4.12 Illustration of "variables_declare_2" block (part 4). 83
4.13 Illustration of "variables_declare_2" block (part 5). 84
4.14 Illustration of "variables_declare_2" block (part 6). 85
4.15 Illustration of "variables_declare_2" block (part 7). 86
4.16 Illustration of C generator for "variables_declare_2" block. . 88
4.17 Unique Storage Mechanism for Variables in the variableHash map variable. 91
4.18 Structure and Flow of a Jison Parser Definition File. 96
4.19 Jison's lexical grammar for C language (part 1). 98
4.20 Jison's lexical grammar for C language (part 2). 99
4.21 Jison's lexical grammar for C language (part 3). 100

4.22 Token Declarations and Operator Precedence in Jison for C Language Parsing. 102
4.23 Snippet from the C Transpiler Grammar Implementation. . . . 103
4.24 Parse tree for example code in C. 105
4.25 Parse tree for expression (part 1). 107
4.26 Parse tree for expression (part 2). 108
4.27 XML Representation of a Blockly Block. 110
4.28 XML representation of a block with a field. 110
4.29 XML representation of a block with value and statement inputs. 111
4.30 XML representation of a value input nested block structure. . 111
4.31 XML representation of statement connection structure. 112
4.32 Comparison of UI representation and corresponding XML format for the while loop block. 113
4.33 Comparison of UI representation and corresponding XML format for the "variables_declare_2" block. 114
4.34 Grammar Declaration for "type_specifier" unit. 115
4.35 Developed Util Functions in JavaScript for repetitive procedures (part 1). 116
4.36 Developed Util Functions in JavaScript for repetitive procedures (part 2). 117
4.37 Developed Util Functions in JavaScript for repetitive procedures (part 3). 118
4.38 Comparison of UI representation and corresponding XML format for "procedures_defnoreturn" block. 121
4.39 Comparison of UI representation and corresponding XML format for "procedures_defreturn" block. 121
4.40 Grammar rules and embedded code for "function_definition" unit. 122
4.41 Grammar rules and embedded code for "declaration" unit. . . 125
4.42 Grammar rules and embedded code for "init_declarator_list" unit. 127
4.43 Grammar rules and embedded code for "block_item" unit. . . 128
4.44 Grammar rules and embedded code for "statement" unit. . . . 128
4.45 Grammar rules and embedded code for "selection_statement" unit. 130
4.46 Grammar rules and embedded code for "iteration_statement" unit. 131
4.47 Grammar rules and embedded code for "jump_statement" unit. 132

4.48 Block representations of Util functions from popular C libraries (part 1). 133
4.49 Block representations of Util functions from popular C libraries (part 2). 133
4.50 Grammar rules and embedded code for "postfix_expression" unit. 134
4.51 Utility function "createFunctionsAndLibraries" (part 1). . . . 135
4.52 Utility function "createFunctionsAndLibraries" (part 2). . . . 136
4.53 Utility function "createFunctionsAndLibraries" (part 3). . . . 137
4.54 Utility function "createFunctionsAndLibraries" (part 4). . . . 138
4.55 Utility function "createFunctionsAndLibraries" (part 5). . . . 139
4.56 Utility function "createFunctionsAndLibraries" (part 6). . . . 140
4.57 Utility function "createFunctionsAndLibraries" (part 7). . . . 141
4.58 HTML and JavaScript code for Import button of the web application. 143
4.59 Utility function "removeIncludes". 146
4.60 Code implementation for "transpileCode" (part 1). 147
4.61 Code implementation for "transpileCode" (part 2). 148
4.62 Code implementation for "transpileCode" (part 3). 149
4.63 Code implementation for "transpileCode" (part 4). 150
4.64 Code implementation for "transpileCode" (part 5). 151
4.65 Code implementation for "transpileCode" (part 6). 151
4.66 Utility function "getDeclaredVariables". 152
4.67 Utility function "changeSetAndGetVariables". 153
4.68 Utility functions "removeCollapsedTags", "stringToXML" and "xmlToString". 154
4.69 Implementation of function "runCode". 156
4.70 The "iframe" tag within visualize.html page. 157
4.71 The "script" code within visualize.html page. 158

5.1 Right-angled triangle made of asterisks (“*”) that should be printed in console. 160
5.2 Evaluation questionnaire for Blockly-C (part 1). 162
5.3 Evaluation questionnaire for Blockly-C (part 2). 163
5.4 Evaluation questionnaire for Blockly-C (part 3). 164
5.5 Gender Statistics. 165
5.6 Knowledgeable of C programming language statistics. 165
5.7 Total amount of scores per item of evaluation. 166

5.8 Total amount of score values per scale. 167
5.9 Score percentage rates. 168
5.10 Different categories of the evaluation items. 169
5.11 Number of positive, negative and neutral evaluations per category. 170
5.12 Benchmark results of UEQ (part 1). 171
5.13 Benchmark results of UEQ (part 2). 171
5.14 Benchmark results of UEQ (part 3). 171
5.15 Replies for short anwser questions (part 1). 173
5.16 Replies for short anwser questions (part 2). 174
5.17 Replies for short anwser questions (part 3). 175

Chapter 1

Introduction

1.1 Motivation

Programming is considered as one of the most desired skills of the 21st century and has become one of the highest paying and stable career options for many. But programming is also a challenging subject to learn, and educators have investigated many strategies for making it more accessible to students. One of those strategies are the visual, block-based programming environments that present an alternative way of teaching programming to novices and have proven successful in classrooms and universities [14].

All beginner programmers face specific problems in their initial stages. The primary issues for a junior programmer are to understand the logic of programming and to familiarize themselves with the syntax of the chosen programming language. The C programming language, recognized for its intricate syntax, is notoriously challenging. It delves deep into the perplexing concept of memory management, spanning pointers and dynamic memory allocation, which can be exceedingly daunting for beginners [3, 12, 10].

1.2 Brief Description

Blockly-C is a web application designed to simplify the complexities of programming. It seeks to separate the intricacies of C language syntax from foundational learning. This application encourages students to prioritize understanding programming logic first. By leveraging a visual block-based programming environment, learners are exempt from immediately grappling

with text-based code. Blockly-C transitions the programming process from visual blocks to textual representation. Through consistent practice and interaction with the block-based coding environment, learners are introduced to the correct syntax of the C language over time.

To further support novice programmers, Blockly-C integrates an auxiliary tool that compiles the code and offers real-time visualization of the heap and stack memory during a program's execution. Visualizing the stack is pivotal for understanding recursion, another complex concept for newcomers. This feature also integrates step-forward and step-backward functionalities. Consequently, users can oversee the console, pointers, dynamic memory allocation, and the values of variables throughout the execution process.

Additionally, Blockly-C facilitates file Input/Output capabilities. Users can export their code as text at any juncture and save it to their local filesystem. Conversely, they can import text code in C from a file, and the application will transform it into a block-based visual format. This two-way conversion is invaluable, enhancing productivity for every programmer.

This platform, accessible from any device, eliminates the need for software installation, guaranteeing cross-platform compatibility. A modern, refined graphical user interface is anticipated to entice more students to immerse themselves in programming, leading to increased productivity. The foundation for Blockly-C's development is Google and MIT's open-source tool: Blockly. This tool's core will undergo a complete overhaul, with a plethora of features added to facilitate the transformation of graphic blocks into legitimate C code. A comprehensive discussion regarding the design and deployment of Blockly-C will be covered in a subsequent section.

1.3 Structure of the Thesis

Chapter 1 presents the driving factors behind this research and provides a concise overview of the project's aims and contributions.

Chapter 2 delves into the foundational knowledge required to set the context for grasping the techniques employed in the system elaborations of this thesis. In the first sections, the state-of-art related work of visual programming are demonstrated. Next the technological background and all the definitions of this thesis are analysed. Also, are presented all the tools, libraries and methodologies that were used for the development of this web application.

Chapter 3 presents collection of user requirements and presents various use cases that exemplify the interactions between end-users and the visual programming web application. In this segment, the focus is on thoroughly analyzing the scenarios in which the application proves beneficial and the different contexts in which it can be utilized effectively. Additionally, this chapter provides a detailed design overview of the web application, outlining its user interface and functionality as presented in this thesis. By exploring these elements, the chapter aims to give a comprehensive view of the user experience, highlighting the application's usability, accessibility, and its impact on facilitating visual programming for users.

Chapter 4 offers an in-depth exploration of the implementation process of the visual programming web application. This section meticulously details the development stages, from the initial design concepts to the final execution. It includes a comprehensive analysis of the programming methodologies employed, the selection of technologies and frameworks, and the challenges encountered during the development process. The chapter also delves into the specific coding practices and software engineering principles applied to build robust and efficient functionalities within the application. Through this chapter, the reader gains a thorough understanding of the technical intricacies and the decision-making processes that underpin the successful creation of the visual programming platform as detailed in this thesis.

Chapter 5 is dedicated to the comprehensive evaluation of the visual programming web application developed in this thesis. This chapter systematically examines the effectiveness, efficiency, and usability of the application through various evaluation methods and metrics. It details the criteria and methodologies used for assessing the application, including both qualitative and quantitative approaches. This section also presents the results of user testing, performance analysis, and feedback collection, providing critical insights into the application's functionality and user experience. The evaluation process not only assesses the application's current state but also identifies areas for improvement and future enhancements. Through this chapter, the reader gains a deep understanding of the application's impact and its alignment with the intended goals and user needs outlined in the thesis.

Chapter 6 concludes by summarizing the main findings and contributions of the research. It reflects on how the project's goals were met and discusses the implications of the work in the broader context of visual programming and computer science education. The chapter also identifies potential areas

for future research and development, suggesting ways to enhance and expand the web application's capabilities. This section not only encapsulates the research but also lays the groundwork for future advancements in the field.

Chapter 2

Background/Research Overview

2.1 Introduction

The evolution of computer programming from traditional text-based methods to visual programming represents a significant shift in the field, bringing forth a new dimension that combines intuitive learning with technical adeptness. This chapter aims to provide a detailed background and research context that underpins this transformation.

Initially, the chapter presents an extensive review of related work in the domain of visual programming. This overview is essential to grasp the current state and historical advancements in this area. It highlights various key developments and contributions that have played a pivotal role in making programming concepts more accessible and interactive. This survey not only charts the progress within the field but also establishes a reference point for understanding the advancements that inform this thesis.

Subsequently, the chapter progresses into a detailed exposition of the technological background and definitions pertinent to this thesis. This section is structured to offer a comprehensive understanding of the tools, libraries, and methodologies utilized in the development of the web application. By elucidating the technical foundations and justifying the choice of specific technologies, this section seeks to provide clarity and insight into the complex mechanisms of the developed system.

Therefore, this chapter serves a dual purpose: it provides a historical and technical context, laying the groundwork for an in-depth examination of the web application's development in the later chapters. It aims to arm readers

with the necessary background information, enhancing their understanding and appreciation of the research and the intricacies of the visual programming platform discussed in this thesis.

2.2 Related Work

2.2.1 Block-c

In their innovative 2017 research, Kyfonidis and Moumoutzis [9] unveiled Block-c, a visual programming interface that simplifies the C programming experience for novices. Leveraging the Openblocks open-source framework, this tool is a desktop-based application that must be installed on a specific operating system. While Block-c does not provide capabilities for code compilation or visualizing memory, it plays a significant educational role by converting block-based interactions into valid C language code, as depicted in Figure 2.1.

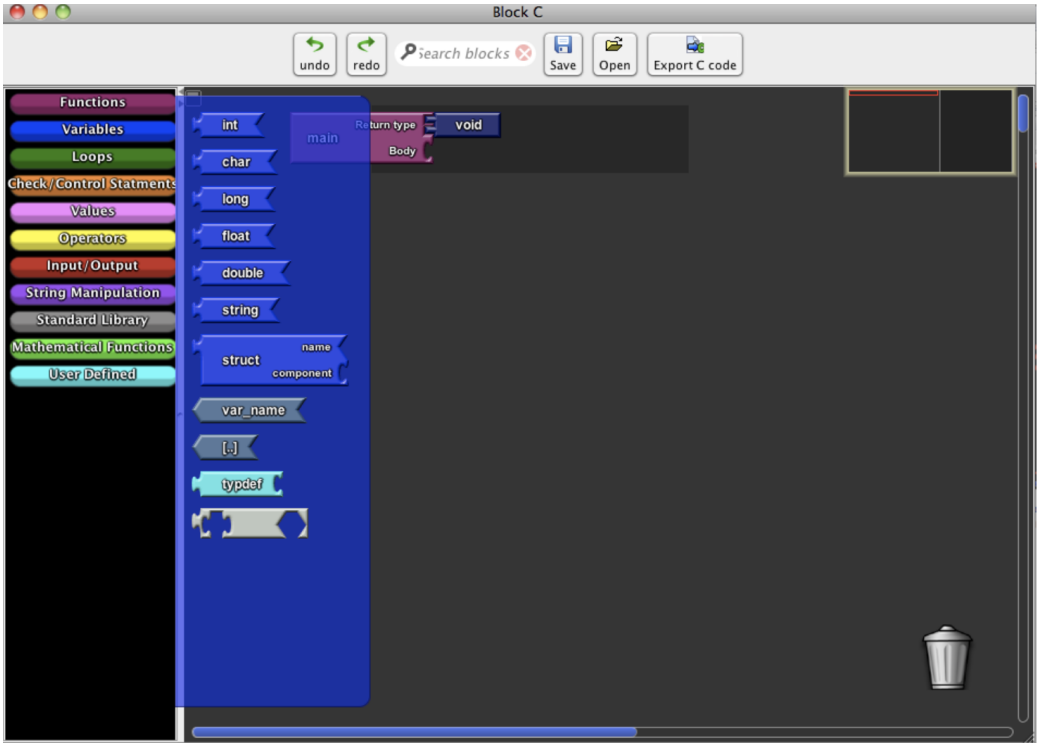


Figure 2.1: A illustration of graphic environment of the Block-C.

The authors comprehensively documented the design, implementation, and subsequent assessment of Block-C, underscoring its primary goal of allowing new programmers to focus on mastering logical structure and algorithmic thought without getting bogged down by the complex syntax specific to C programming. Its user-friendly graphical interface aims to minimize syntactic errors that often discourage beginners.

The evaluation phase of Block-c provided robust evidence of its effectiveness in reducing the incidence of syntax errors. More than just an error-prevention tool, it actively engaged its users through an interactive visual platform. The tool's design is imbued with best practices from the field of Human-Computer Interaction, which are aimed at enhancing user engagement and learning outcomes. As a result, Block-c has been praised for its role in supporting the learning journey of programmers by providing a supportive scaffold that bridges the gap between conceptual understanding and practical application. Its positive reception by learners, demonstrated through high acceptance rates, indicates that Block-c's approach resonates with users, of-

fering them a more intuitive and less intimidating entry point into the world of programming.

Furthermore, the success of Block-c suggests potential for future developments. The tool's framework could be expanded to include features such as real-time code compilation and memory management visualization, which would enhance its utility as a comprehensive learning aid. By building on the solid foundation that Block-c has established, there is a clear pathway for developing more advanced tools that can continue to lower the barriers to entry for programming students and enrich their educational experiences.

2.2.2 C programming environment

In 2019, the innovative work of Abe and Yuki [1] unveiled a new development environment tailored for C programming, crafted to intuitively teach users about variable declarations and the inclusion of libraries. This environment goes further, providing a graphical representation that captures the changes in variable values before and after execution, aiding in the understanding of expression evaluation. It was designed to be user-centric, allowing for the assembly and modification of C programs via a drag-and-drop interface within a block panel, as shown in Figure 2.2. Its functionality includes stepwise code execution, which users can navigate forward or backward, granting them insight into variable states and console output at any stage. This web-based tool, built on JavaScript, ingeniously converts expressions into reverse Polish notation for step-by-step assessment, clearly exhibiting the breakdown and calculation of expressions. An execution history feature fortifies the learning experience by providing the capability to review and understand each step taken. Studies assessing the effectiveness of this tool have indicated that it empowers students who find C programming challenging, helping them to code more accurately and quickly than they would with conventional text coding.

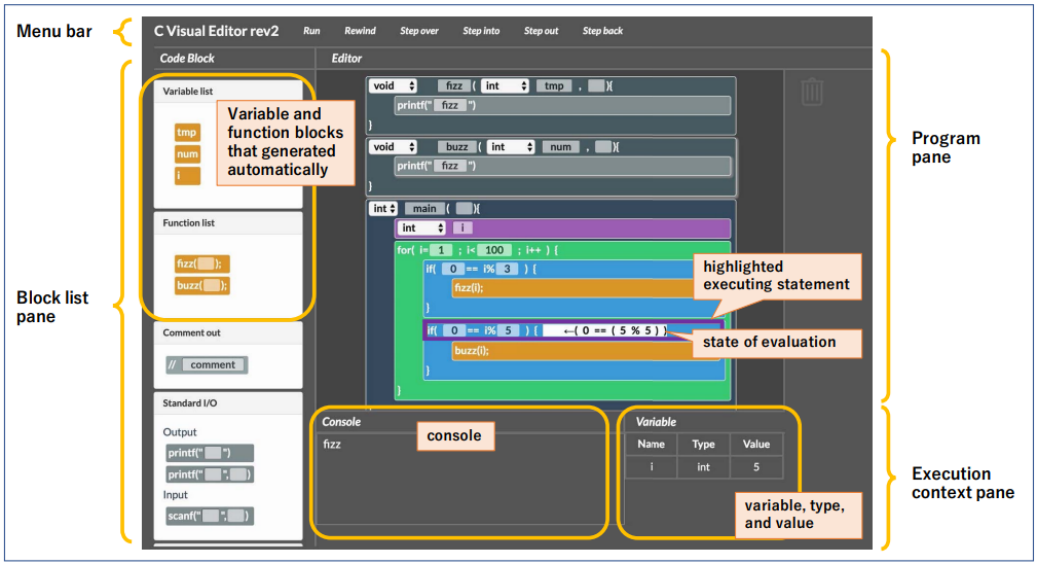


Figure 2.2: An illustration of graphic environment of Abe and Yuki.

This environment's programming editing feature significantly streamlines the code construction process by offering a block-based coding approach. It allows learners to select and place blocks representing various elements of the C language into the programming space, doing away with the need for rote memorization of keywords and syntax. This method effectively reduces input errors, which in turn decreases the likelihood of syntactical mistakes. Highlighted functionalities of this innovative platform include: the simplified creation of programs through block manipulation, obviating the need for deep syntax knowledge; specialized blocks for declarations to aid comprehension of these fundamental concepts; and advanced step execution tools that grant meticulous tracing of the program's behavior, including a 'step-back' feature for easy identification of missteps. Trials have shown that this approach is particularly beneficial for those who are not adept at C programming, enabling them to craft code that is both more precise and rapid than the code produced by traditional, text-based coding methods.

2.2.3 SeeC

Debugging often presents a formidable challenge for novice programmers, as standard tools are generally too complex and lack essential information

needed by beginners. The 2013 paper by Heinsen Egan and McDonald [5],[6] introduces SeeC, a project dedicated to developing a debugging tool specifically for novices in the C programming language. An illustration of this tool can be seen in figure 2.3. SeeC focuses on detecting common runtime errors that confuse student programmers, such as invalid memory access and the use of uninitialized memory, and incorporates execution tracing for comprehensive error analysis and program history review.

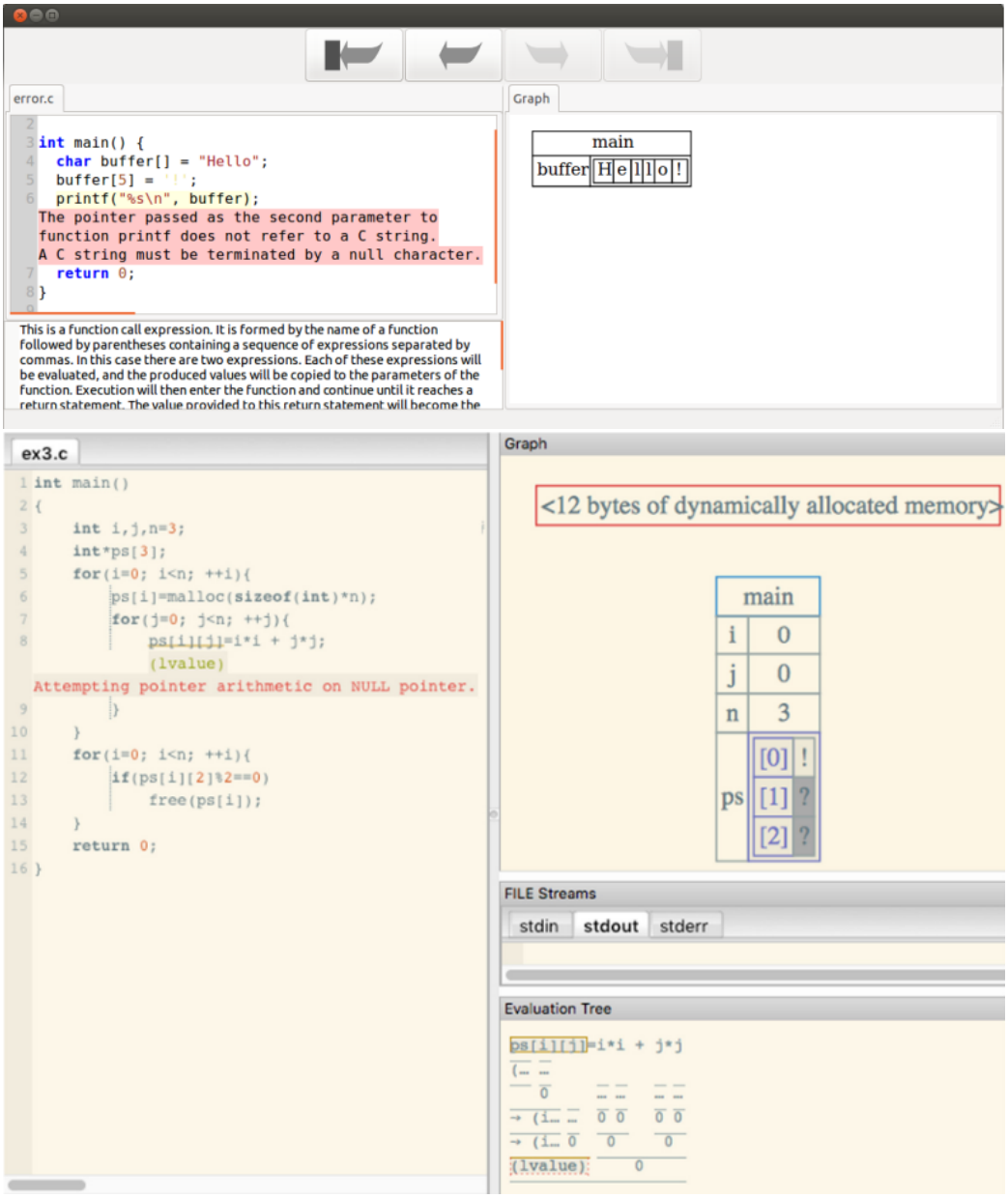


Figure 2.3: An illustration of debugging tool SeeC.

This complexity extends beyond standard tools for beginners in programming, who often lack the broad knowledge required for effective debugging. This can lead to extended troubleshooting sessions, potentially discouraging

them from continuing their education in programming. The C programming language, with its complexities like pointers and manual memory management, adds to these challenges. The scarcity of novice-friendly tools in C programming, due to the focus of recent tool developments on object-oriented languages, exacerbates the situation.

The advancement of programming languages and compilers has resulted in sophisticated debugging techniques, primarily aimed at experienced programmers. However, these tools are often too complex for novices, who struggle with even basic usage. SeeC was designed to address this gap by simplifying debugging for beginners. It records program execution automatically, detects runtime errors, and provides a graphical interface for reviewing these errors.

The paper further explores the complexities of the C programming language and the challenges it presents to both students and educators. It outlines the framework used for assessing and designing debugging systems, reviews existing tools for both novice and expert programmers, and discusses the implementation requirements for the proposed system. The paper highlights the use of the LLVM compiler infrastructure for execution tracing and the Clang project for in-depth syntactic and semantic analysis of C code. It concludes with a comparative analysis with the Memcheck tool and future directions for the project.

In their evaluation of existing debugging tools and research on novice programmers' debugging practices, the researchers identified trace-based debugging and automatic error detection as key features for a novice-focused debugging tool. SeeC, designed with these features, supports program execution recording, reconstruction of historical program states, automatic runtime error detection, and correlating error messages with the original source code.

The LLVM Compiler Infrastructure, a collection of modular compiler technologies, plays a crucial role in this context. Its intermediate representation (LLVM IR) is a low-level, typed, and source-independent assembly language essential for the static single assignment. The LLVM IR forms the core of the compilation process, bridging the gap between language-specific front-ends and subsequent stages, including transformations and final conversion to machine code. In SeeC, a transformation is used to embed execution tracing and runtime error detection directly into the code.

The preceding section noted that SeeC conducts pre-execution instruction checks, leveraging information about the program's state maintained by Thread Listeners and the Process Listener. This process involves up-

dating the tracing library's understanding of the state, incorporating data on dynamic memory allocations, local variable allocations, and the status of memory initialization. SeeC presumes all memory as uninitialized unless associated with a global variable or altered by the program or a C standard library function, reverting it to uninitialized upon deallocation.

Clang, as part of the LLVM project, is a versatile front-end tool used for compiling programs written in various languages into LLVM IR. It utilizes parsing and semantic analysis capabilities to generate an Abstract Syntax Tree (AST) from a program's source code, which SeeC employs to contextualize runtime errors. Clang also generates debugging information correlating LLVM IR and machine code back to the source code. However, to provide more precise information for novice programmers, SeeC integrates additional coding into Clang's code generation process. The review of existing novice-focused debugging tools reveals a reliance on existing systems for execution tracing and source code analysis, such as the Java Platform Debugger Architecture. However, the C programming language lacks equivalent interfaces, leading to the development of custom tools with incomplete language support. Many C debugging tools are built upon The GNU Project Debugger (GDB), but they face limitations in their textual interface and reversible debugging capabilities. The advancements in compiler-level tools for C offer potential for enhanced language support and sustainability, crucial for making these tools suitable for novice programmers.

2.2.4 PlayVisualizerC

Building on the work of Ishizue, Sakamoto [7], and colleagues from 2018, PlayVisualizerC (PVC) was introduced as an essential visualization tool for C programming beginners, focusing on memory management as showcased in Figure 2.4.

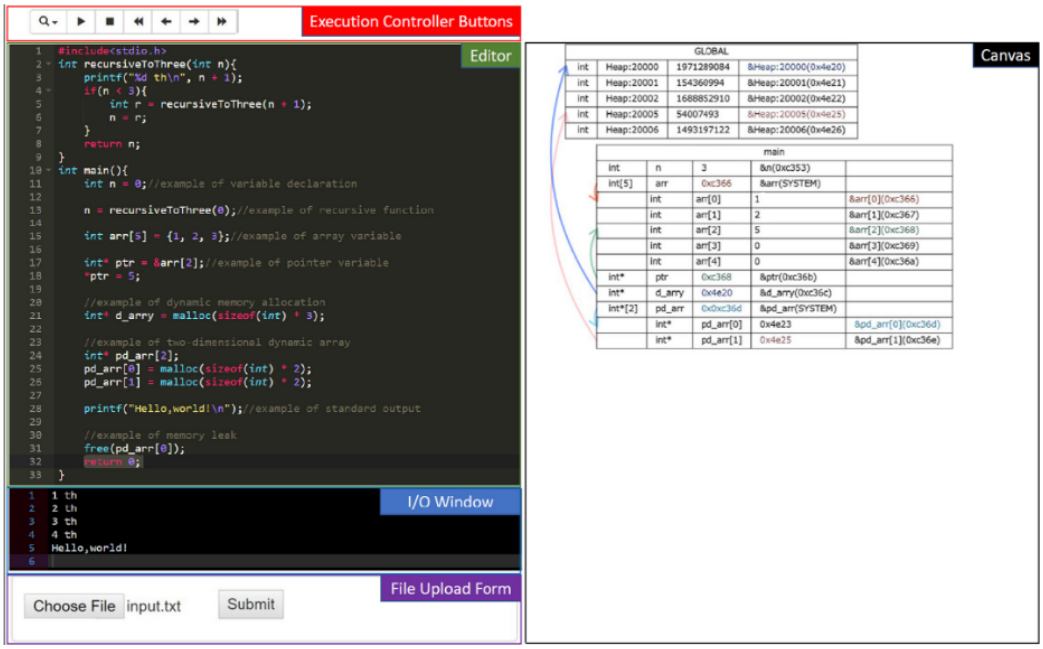


Figure 2.4: An illustration of PlayVisualizerC.

Building on the foundation established by existing visualization tools, PlayVisualizerC.js (PVC.js) emerges as an innovative response to the prevailing challenges in the realm of programming education. While tools like SeeC and Python Tutor (PT) have made strides in the visualization of memory management in C programming, PVC.js addresses the specific needs of beginners in this field. Its design is centered around three core solutions: comprehensive support for dynamic memory operations and input/output processes (S1), ease of access and use through integration into web browsers using JavaScript (S2), and a streamlined approach for revisualizing modified source code (S3).

To address the challenges of capability (P1), ease of installation (P2), and user-friendliness (P3) that affect many program visualization tools, PlayVisualizerC.js (PVC.js) has been introduced. PVC.js is a C language interpreter equipped with features for visualizing program execution. It offers three key solutions (S1–3) to these challenges. The first solution (S1) enables PVC.js to comprehensively visualize variables, pointers, arrays, dynamically allocated memory, and their interrelations, along with supporting both standard and file I/O operations. The second solution (S2) is its design as a JavaScript ap-

plication, operational directly within a web browser, thereby eliminating the need for a web server or an internet connection for installation. The third solution (S3) introduces the convenience of revisualizing programs with a single click, allowing for easy observation of changes made to the source code.

PVC.js, an innovative tool designed to tackle the challenges P1–3, is crafted using HTML and JavaScript. This includes graphical user interfaces, a C language parser, and a semantic analyzer, all functioning client-side as part of a JavaScript application. Figures 3 and 4 in the documentation offer a systematic overview and a visual representation of PVC.js respectively. For the implementation of PVC.js, ANTLR was utilized for parsing the source code, and unicoen.ts was employed to construct and execute the abstract syntax tree (AST), named UniTree, for C languages.

Remarkably, PVC.js operates as a standalone JavaScript application that does not depend on a web server. Users can simply launch PVC.js by opening the associated HTML file in a web browser. This self-contained nature of PVC.js means it operates independently of an Internet connection, making it readily accessible and addressing the installability challenge (P2). With all necessary functions integrated within, PVC.js emerges as a comprehensive solution that enhances ease of access and usability.

PVC.js features a user interface composed of five key elements: an editor for code input, execution control buttons, an I/O window, a visualization canvas, and a file upload section. The editor allows users to write and edit their source code. The execution control buttons facilitate the step-by-step execution of the program. The I/O window displays the standard output of the program, such as the output from printf, and it also accepts standard input like scanf. The canvas visually represents the program's execution status with the help of tables and figures, and its layout dynamically adjusts to fit the browser window's size.

In PVC.js's graphical interface, the program's execution controller includes six distinct buttons: adjusting editor font size, initiating program execution, stopping execution, stepping backward through all steps, moving backward one step at a time, and stepping forward one or all steps. Each of these actions corresponds to a specific phase in the execution process. Additionally, PVC.js allows users to upload and utilize local files through the file upload form. This functionality enables the use of files in programs, leveraging functions like fgets and fputc, providing a more comprehensive programming experience within the PVC.js environment.

The effectiveness of PVC.js was rigorously assessed through two distinct

experimental setups. The first experiment showcased the tool's impact on learning efficiency and accuracy: students using PVC.js were able to complete programming tasks around 1.7 times faster and with a 19% increase in accuracy over their counterparts using SeeC. This indicates a significant enhancement in both the speed and quality of learning when PVC.js is employed.

The second experiment focused on comparing the visualization efficacy of PVC.js with that of PT. The findings were notable – PVC.js not only matched PT in terms of visualization performance but also proved to be a more effective tool for novices dealing with complex programming scenarios. Particularly in situations where understanding the changes in variable values and control flow was critical, PVC.js outperformed traditional debugging tools. This underlines PVC.js's potential as a superior educational aid in programming, especially beneficial for beginners who are navigating the intricate aspects of C language memory management and control structures.

In summary, these evaluations underscore the value of PVC.js as an advanced tool that enhances the learning experience for novice programmers, offering an efficient, accessible, and effective approach to understanding key programming concepts.

2.3 Technological Background and Definitions

This chapter outlines the key technologies and libraries that form the foundation of the web application detailed in this thesis. The focus is on three primary tools: Blockly, Jison, and Python Tutor, each playing a crucial role in the application's development. By delving into the functionalities and applications of these technologies, the chapter aims to shed light on how they were effectively harnessed to build a comprehensive and user-friendly programming environment. This exploration serves not only to document the technical aspects of the application but also to provide insights into the selection process and integration strategies of these tools, illustrating their impact on the application's overall performance and user experience.

2.3.1 Blockly

Blockly, developed by Google, is a prominent web-based, visual programming editor [2]. It offers a unique interface where users can build applications by

piecing together blocks that represent code concepts (figure 2.5). This chapter provides an in-depth look at Blockly, exploring its features, capabilities, and the role it plays in the development of the web application discussed in this thesis.

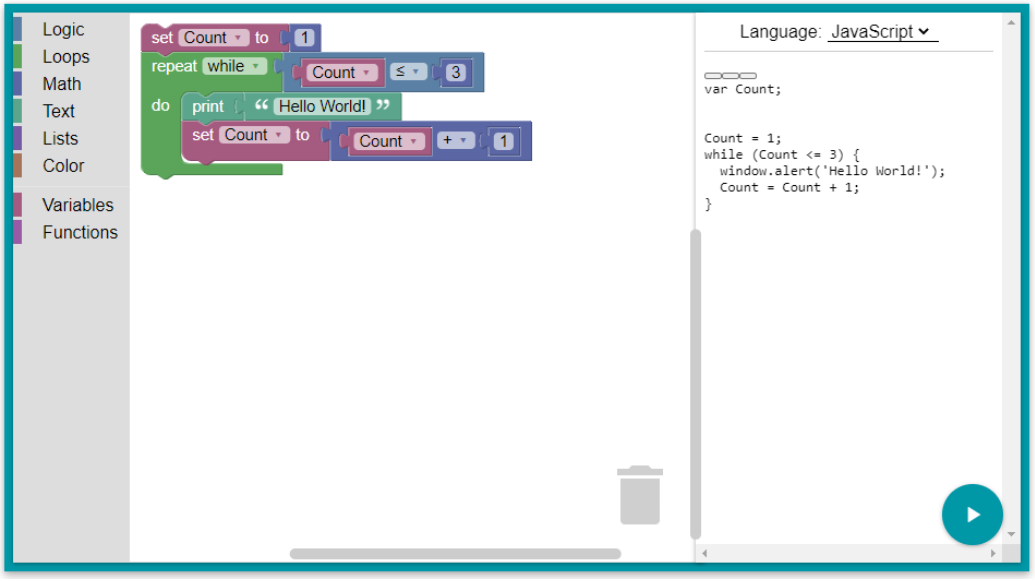


Figure 2.5: An illustration of Blockly.

Blockly stands out for its block-based coding approach, designed to simplify programming concepts, making them more accessible, especially for beginners. Each block corresponds to a piece of code, such as a function, variable, or control structure. Users can drag and drop these blocks to create executable programs. Blockly is highly versatile, supporting several programming languages like JavaScript, Python, PHP, Lua, and Dart.

Blockly comes with a comprehensive set of pre-defined blocks that cover basic programming structures. Developers can also create custom blocks to cater to specific needs. One of Blockly's most powerful features is its ability to convert block-based scripts into syntactically correct code in various programming languages. Being a web-based tool, Blockly can be integrated into any web application and is not restricted by the user's operating system. Blockly's intuitive drag-and-drop interface makes it an ideal tool for educational purposes, allowing learners to focus on logic and structure rather than syntax.

In the context of this thesis, Blockly serves as the core component for the code-creation phase of the web application. It allows users to visually assemble code, which is particularly beneficial for beginners unfamiliar with C programming syntax. The application leverages Blockly's customizable blocks to represent various C language constructs, providing an interactive and engaging way to learn programming.

To cater to the specific requirements of the C programming language, custom blocks were developed. These blocks represent various C constructs, including variables, loops, and functions. This customization extends Blockly's utility beyond basic programming, making it a powerful tool for teaching complex programming languages like C. One of the key functionalities integrated into the application using Blockly is the ability to export the block-based script into C code. This feature utilizes Blockly's code generation capabilities, transforming the visual blocks into executable C code, which users can then run or further modify.

While Blockly simplifies coding, integrating it into a web application tailored for C programming presented unique challenges, such as ensuring accurate translation of blocks to C syntax and providing an interface that accurately represents C programming logic. These challenges were addressed through careful customization of blocks and thorough testing to ensure the generated code's accuracy and reliability.

Blockly's integration into the web application represents a significant step towards making C programming more accessible and understandable, especially for novices. Its user-friendly interface, combined with powerful code generation capabilities, makes Blockly an invaluable tool in the realm of educational programming platforms. This chapter has outlined the fundamental aspects of Blockly and its implementation in the web application, highlighting its pivotal role in achieving the project's educational objectives.

2.3.2 Jison

Jison acts as a cornerstone in modern web application development, especially when it comes to implementing custom programming language interpreters or compilers [8]. This chapter delves into the intricacies of Jison, a powerful parser generator, and its role in the development of the web application presented in this thesis (figure 2.6).

Figure 2.6: Jison logo.

Jison is an open-source parser generator tool inspired by Bison and Yacc. It's designed for JavaScript applications, enabling developers to create custom parsers for various languages or data formats. Jison interprets Context-Free Grammar (CFG) specifications and generates JavaScript-based parsers.

Jison allows developers to define grammars using Bison-like syntax, making it easier for those familiar with traditional parser generators to adapt. Jison supports LALR(1) and LR(0) grammars, offering flexibility in defining complex language rules and syntaxes. It provides robust error reporting and recovery mechanisms, essential for developing interpreters and compilers. Being JavaScript-based, Jison seamlessly integrates into web applications, allowing for client-side parsing tasks.

In the web application, Jison is employed to parse C code and generate corresponding XML for Blockly. This functionality is crucial for the feature that allows importing C code into the block-based interface and translating it into visual blocks. The development process using Jison involves defining the grammar of the target language—in this case, C—and then utilizing Jison to generate the parser. This parser interprets C code, breaking it down into tokens and structures that can be translated into Blockly-compatible XML.

Parsing C code presents several challenges, such as handling complex syntax and maintaining context through different code segments. These challenges were addressed by meticulously defining the grammar and testing the parser across various C code samples to ensure accurate parsing and XML generation.

Integrating the Jison-generated parser with Blockly required careful alignment between the C code structure and the Blockly blocks. This integration was key to enabling features like code importation and ensuring that the

generated Blockly blocks accurately represent the structure and logic of the imported C code.

Jison's parser generation capabilities play a pivotal role in bridging the gap between textual C code and visual Blockly blocks in the web application. Its ability to handle complex parsing tasks, coupled with seamless integration into JavaScript environments, makes it an invaluable tool in the development of educational programming platforms. This chapter has provided a comprehensive overview of Jison, highlighting its functionalities, integration challenges, and solutions within the context of the web application. The successful incorporation of Jison demonstrates its effectiveness in enhancing the application's capabilities, particularly in facilitating the learning of C programming through a visual approach.

2.3.3 Python Tutor

Python Tutor, originally developed by Philip Guo [4], is an open-source educational tool designed to help learners visualize the execution of code in various programming languages, including Python, Java, JavaScript, and C. It offers a clear view of how code runs, displaying the call stack, heap, and variables at each step of execution (figure 2.7).

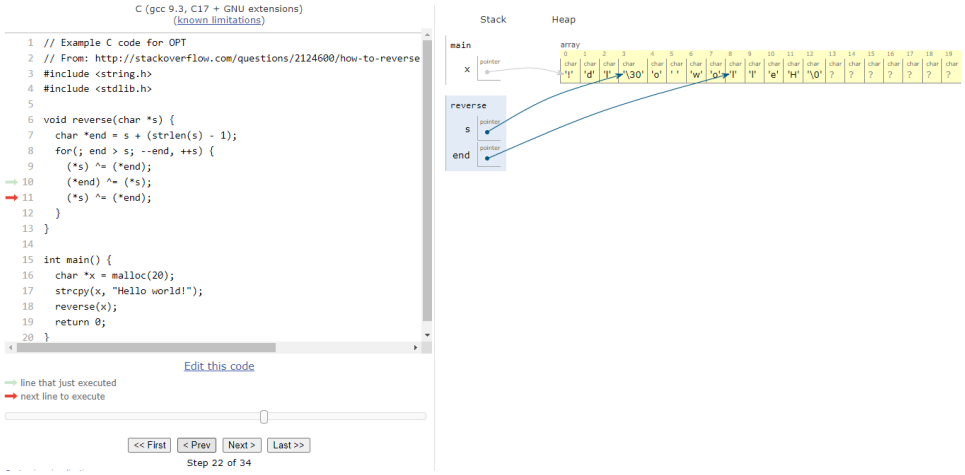


Figure 2.7: An illustration of PythonTutor.

Python Tutor supports several programming languages, making it versatile for different educational contexts. It allows users to step through

their code line-by-line, observing changes in the program state at each step. Python Tutor graphically represents the program's memory, showing variables, arrays, objects, and pointers, which is crucial for understanding complex data structures and pointers in C. The tool's interactivity enhances learning, allowing students to experiment with code and instantly see the results. In the context of this web application, Python Tutor plays a pivotal role in the code execution and memory visualization page. It provides an essential feature allowing users to understand the dynamic behavior of C programs, particularly the manipulation of memory and execution flow.

Integrating Python Tutor with the web application involved several steps. The application converts Blockly blocks into C code, which Python Tutor can process. The C code is URL-encoded to be compatible with Python Tutor's web interface. Python Tutor's visualization is embedded within the web application using an iframe, ensuring a seamless user experience.

Integrating Python Tutor posed challenges, particularly in ensuring the accurate translation of block-based code to syntactically correct C code. Additionally, managing communication between the iframe and the main application required careful handling to maintain synchronization and user experience.

Python Tutor significantly enhances the educational value of the web application. It offers an interactive and engaging way for users to learn programming concepts, particularly those related to memory management in C, which are often challenging for beginners.

Python Tutor's inclusion in the web application enriches the learning experience by providing a powerful visualization tool for understanding code execution. Its ability to demystify complex programming concepts, combined with its seamless integration into the application, underscores its utility in educational programming platforms. This chapter has delved into Python Tutor's functionalities, its integration challenges, and its role in enhancing the web application's capabilities, particularly in aiding the learning process of programming, especially in the context of C language.

Chapter 3

End User Experience

3.1 Introduction

The End User Experience chapter elucidates the interactive journey of novice programmers within the Blockly-C application. With an increasing surge towards demystifying programming and making it more accessible, Blockly-C emerges as a powerful tool that redefines the learning curve for understanding and mastering the C programming language. This chapter aims to cast light upon the practical, user-friendly elements integrated into Blockly-C, offering a robust understanding of its myriad functionalities from a user's perspective.

The goal of Blockly-C is to significantly reduce the obstacles faced by emerging programmers. By converting intricate syntax into an intuitive, block-based interface, Blockly-C seamlessly blends learning with interaction, making programming more approachable. The application's environment prioritizes a clean, clutter-free interface allowing users to concentrate wholly on understanding programming logic, laying down a solid foundation before transitioning to mastering syntax.

This exploration encompasses a detailed walk-through of the Blockly-C interface and functionalities, including the Block-to-C Code Conversion, C-to-Block Representation, Code Execution, and Memory Visualization. Each section will delve into the specific elements of these functionalities, underscoring their contribution to a streamlined, effective learning experience.

Within the Block-to-C Code Conversion, the focus rests on the user's interaction with the block interface, demonstrating how users construct a program using the drag-and-drop blocks. Following this, the C-to-Block

Representation section highlights the reverse translation, showcasing the application's ability to translate C code back to block form, further enhancing comprehension and learning.

The subsequent sections, Code Execution and Memory Visualization, bring forward Blockly-C's integrated tools for running the constructed C code and visualizing the behind-the-scenes memory management. This feature, essential for understanding the underlying operations of C programming, is presented with a focus on its user-friendly implementation within the Blockly-C environment.

In essence, this chapter serves as a comprehensive guide, offering insight into the rich, interactive user experience crafted within Blockly-C, establishing it as a pioneering platform for learning C programming. Through detailed explanations and illustrative examples, the following sections elucidate each aspect of the application, reinforcing its significance as an innovative educational tool in the programming world.

3.2 Block-to-C Code Conversion

In this section, delve into the core functionality of the Blockly-C web application: the Block-to-C Code Conversion. Blockly-C's intuitive design allows users to construct C programs using visually representative blocks, eliminating syntax errors and providing immediate, clear feedback on their code structure.

The finalized version of this tool encompasses four separate sections (figure 3.1).

1. The initial section is designated for buttons that operate the tool's functions, such as downloading and importing

2. The second section contains the blocks grouped into categories.

3. The third section is the "Workspace", an area where users arrange the blocks to formulate a program.

4. The fourth section is the "Code" component that displays in real time the generated code in plain text C.

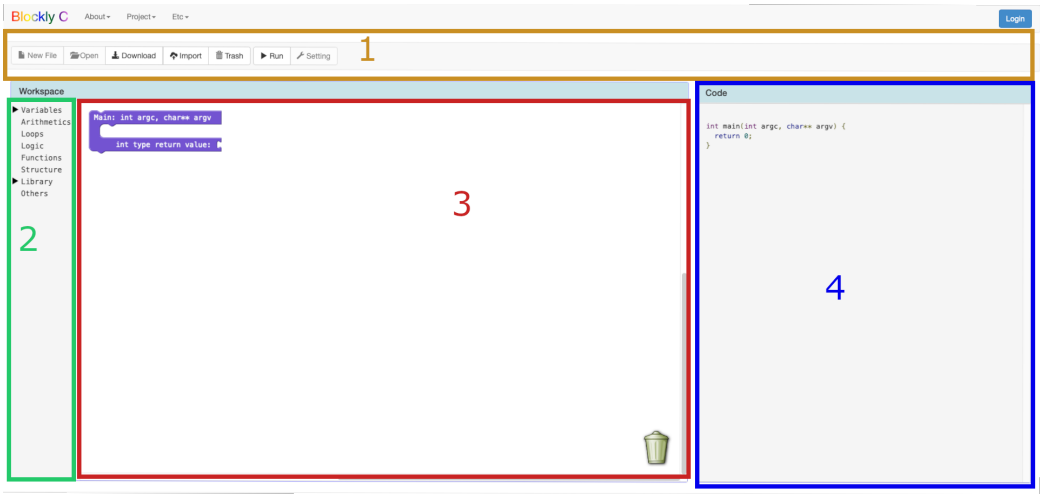


Figure 3.1: A detailed illustration of graphic environment of the tool.

The initial view upon opening the web application is displayed in Figure 3.2. Upon accessing the Blockly-C workspace, users are greeted with a default block representing the `'main'` function in C. This indispensable block remains a constant, guiding novices to frame their code within the essential `'main'` function. All graphical elements are directly manipulated with the mouse. Users can move any block by pressing and holding the left mouse button, and release it by letting the button go. The system's operation relies on the drag-and-drop functionality, a feature prevalent in numerous graphical environments for file management.

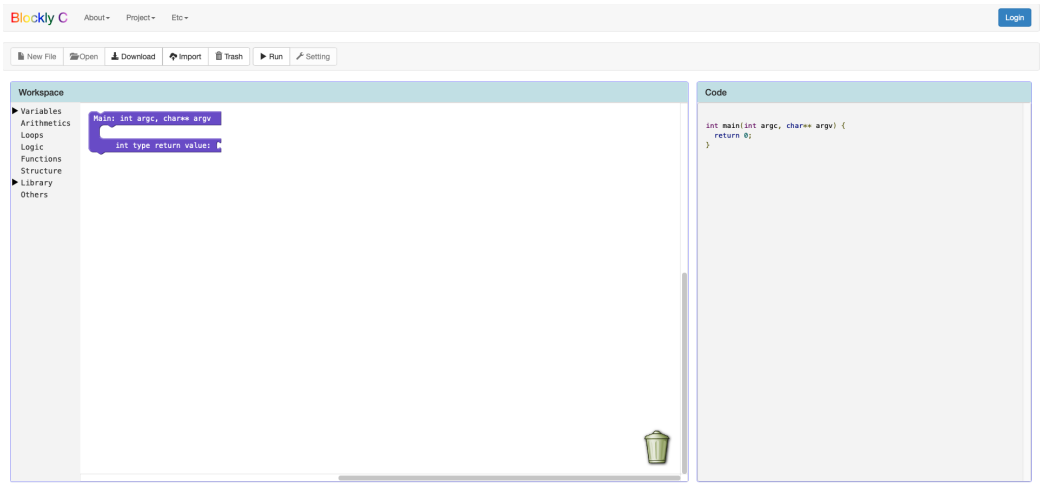


Figure 3.2: Initial view upon opening the web application.

The Blockly-C application categorizes blocks for efficient selection and use. On the left sidebar, users will find distinct categories like `'Variables'`, `'Arithmetics'`, `'Loops'`, and others, further broken down into subcategories such as `'Variable'`, `'Pointer'`, `'Array'` under `'Variables'`, and various libraries under `'Library'`. This categorization simplifies the search and selection process, enabling users to quickly find and select the desired block. This approach organizes blocks according to their specific functionalities, enabling users to easily identify and select the necessary blocks based on recognition, rather than relying solely on memory recall [11].

When a category or sub-category is clicked, a palette of related blocks is displayed. Users can click, drag, and place these blocks into the workspace, constructing the structure and logic of their C program. The process is instinctive, mirroring real-world object manipulation and contributing to the approachable and engaging user experience (figure 3.3).

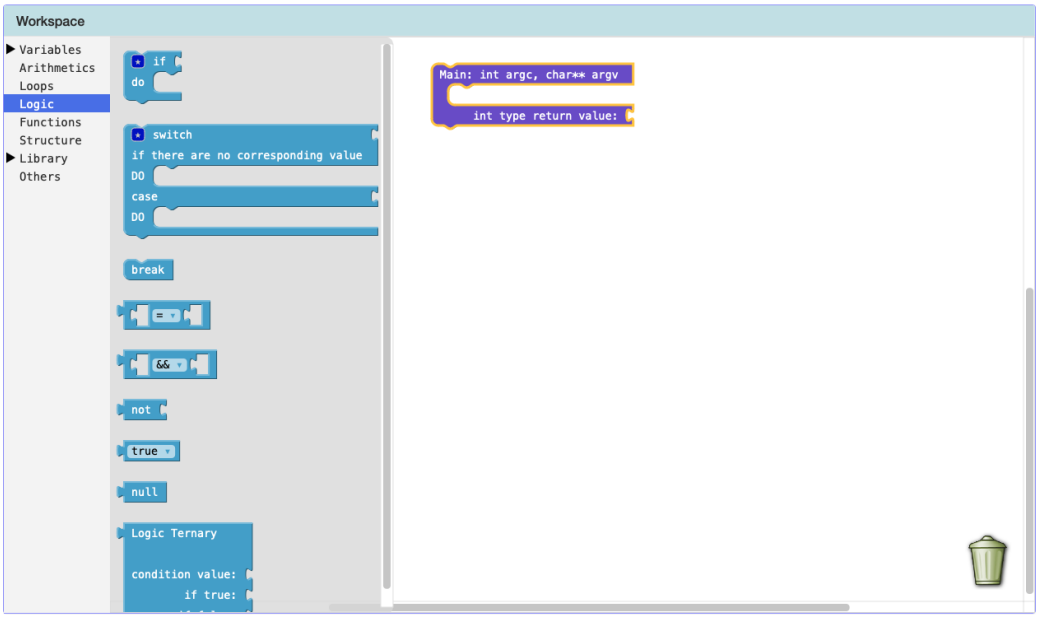


Figure 3.3: A figure illustrating the categories sidebar, with a focused view on the blocks within the 'Loops' category.

Each block in Blockly-C is specifically designed to interlock with others in one of three particular ways, reflecting the allowable syntax structures in C. The primary connection type is the encapsulated connection, suitable for blocks inside functions or loops. The second type, the sequential connection, links individual statements based on their execution sequence in the program. The third, known as the value connection, caters to assigning values to variables or expressions. To establish a connection between two blocks, users are required to drag one block over the other, ensuring one connector aligns above its counterpart. It's crucial that both connectors are of identical types, with one being designed as male and the other as female. The varied connector types and their underlying conceptual significance are elaborated upon in figure 3.4.

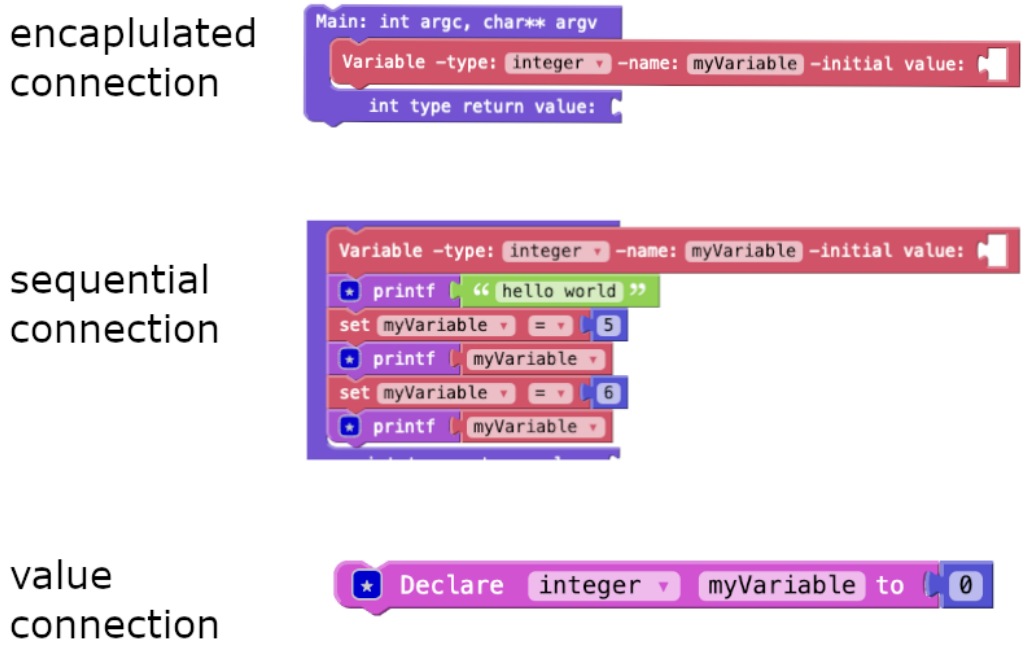


Figure 3.4: Connector Types in Blockly-C.

Whenever a connection is initiated, Blockly-C's intelligent design highlights valid connection spots in yellow, ensuring users a seamless, error-free coding experience. Invalid attempts are promptly responded to by automatically distancing the blocks, signaling the user about the unacceptable connection (figure 3.5). Connections serve as the primary mechanism to prevent syntax errors. They guide users to craft accurate expressions by visually indicating potential block pairings, particularly for blocks having multiple attachment points. These connectors also help users develop the right mental models, helping them distinguish and group blocks based on their syntactic category (e.g., logical operators versus numerical ones). Upon establishing a connection, a distinctive sound is emitted, and the encompassing "parent" block visually enlarges to accommodate the nested "child" block.

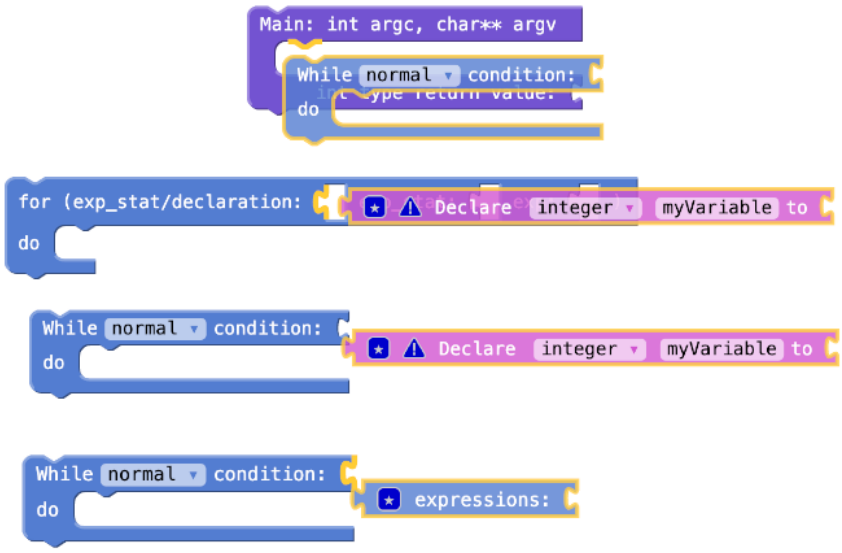


Figure 3.5: Highlighted Connections in Blockly-C.

Beyond the basic connections inherent to each block, Blockly-C integrates a series of additional functionalities that bolster the versatility and the overall user experience. Some blocks, for instance, come equipped with text input fields, allowing users to type in specific values. An apt example of this is the `"math_number"` block, which permits users to input any number. Similarly, the `"variable_declare"` block has a text input feature, enabling users to declare a new variable and assign it a distinctive name (figure 3.6).

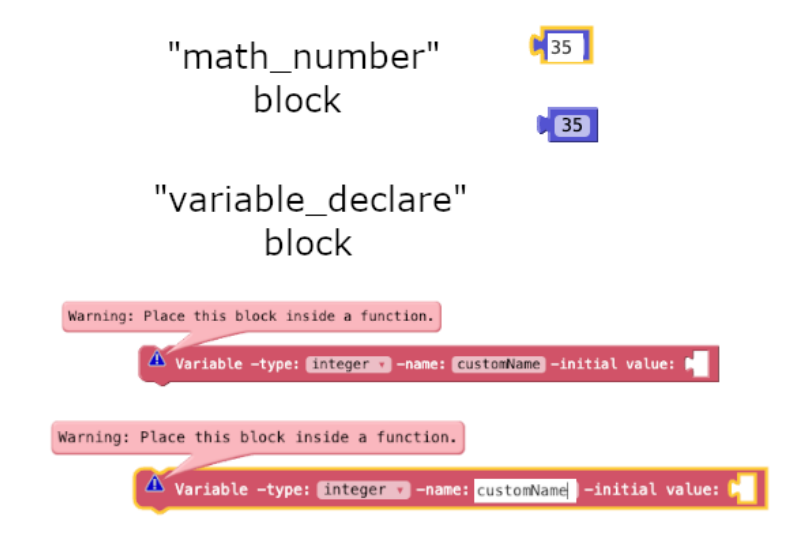


Figure 3.6: Text Input Fields on Blocks.

To facilitate a more intuitive user experience, some blocks are equipped with a drop-down menu that provides specific customization options. For instance, the `"variable_declare"` block possesses such a menu that enables users to specify the data type for the variable they are declaring. This drop-down menu presents a variety of data type choices, including `"integer"`, `"unsigned int"`, multiple real number types like `"float"` and `"double"`, as well as `"character"`. This feature ensures users have a clear and guided approach to setting variable types (figure 3.7).

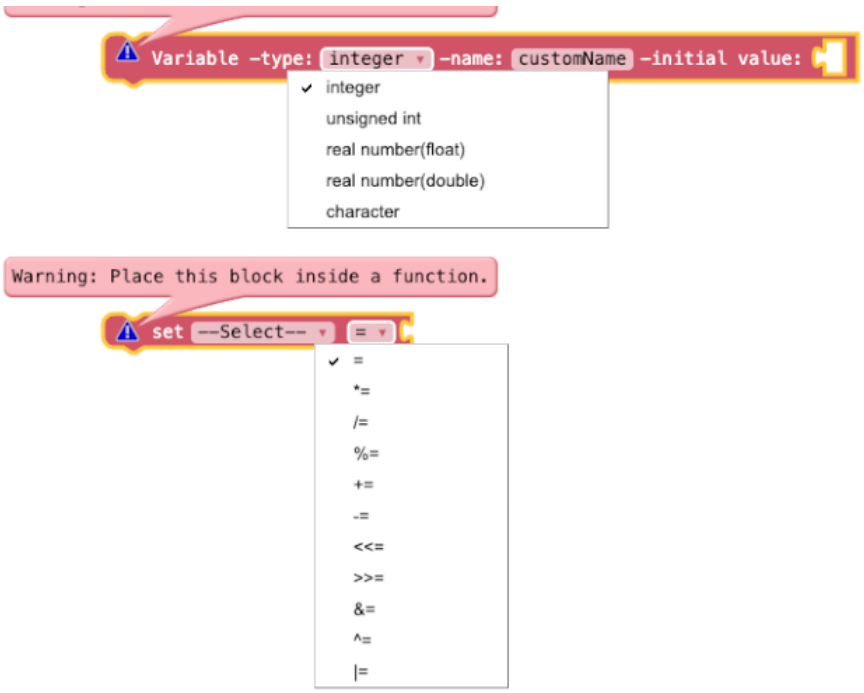
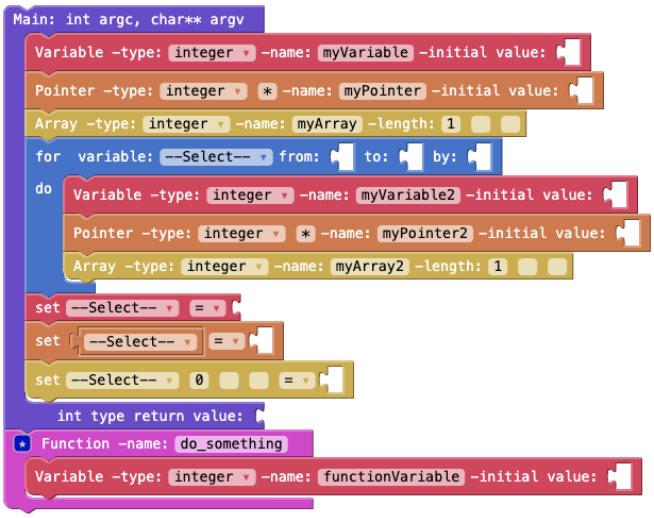


Figure 3.7: Drop-down menu on Blocks.

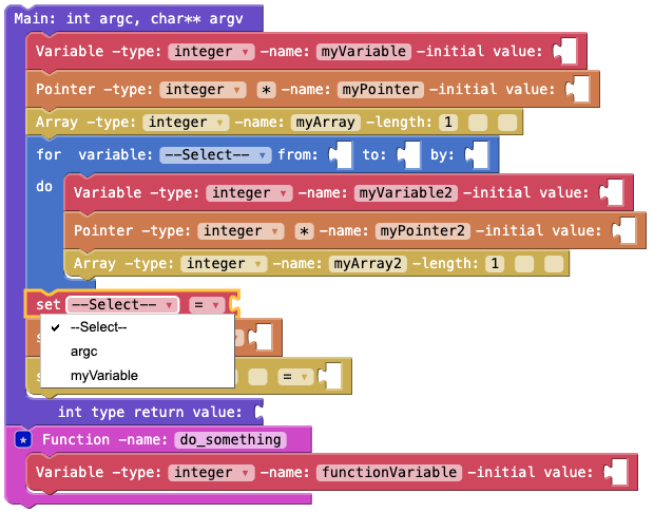
Furthermore, Blockly's intelligent design ensures a dynamic interaction between blocks. Once a variable, array, or pointer is declared within a function or the main program, its name and type are registered within the Blockly environment. Consequently, when adding a "setter" or "getter" block to the workspace, it features a drop-down menu to select from the already declared variables. This selection menu is context-aware; its list of available variables is updated dynamically based on the block's position in the code.

For instance, if the "setter" or "getter" block is placed within the main function, it will not display variables that are exclusive to another function's scope. Similarly, the position of the block within the same function matters. A "setter" block placed on line 9 won't recognize a variable declared on line 10. This approach extends to other structures as well; if a variable is declared within a `for` loop, it remains invisible to blocks positioned outside that loop, even if chronologically placed after the loop in the code. Such dynamic and context-sensitive features not only greatly aid in preventing scope-related errors but also help novice programmers develop a clear understanding of variable accessibility and scope rules in C, actively assisting them in avoiding

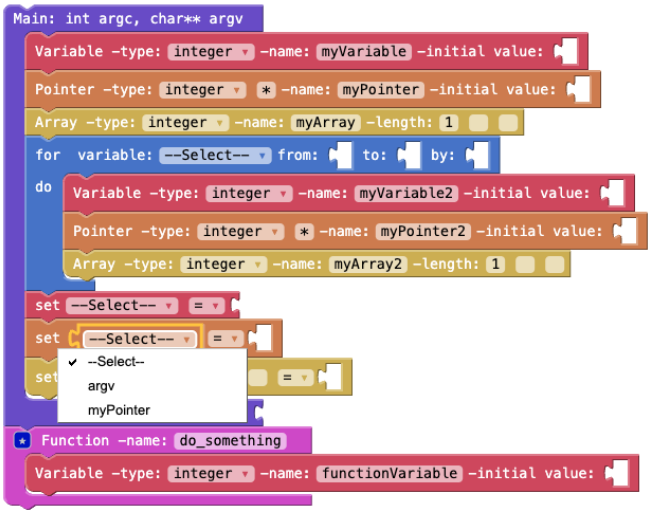
common coding errors (figure 3.8).



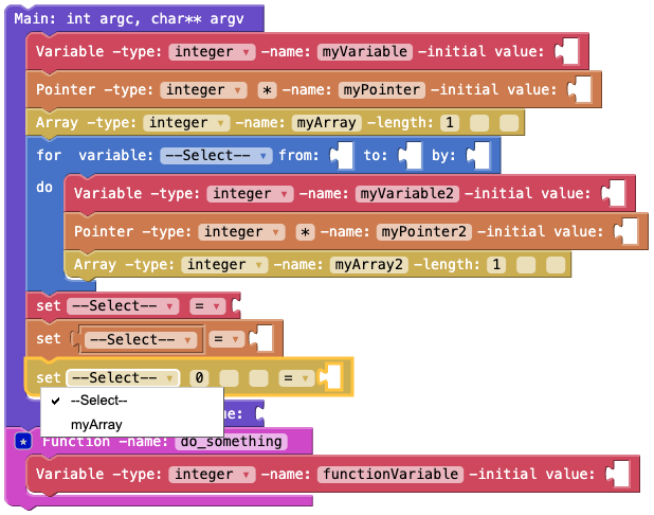
Example of C Code Representation in Blockly-C UI.



Display of Available Variable Options from a Variable Setter's Dropdown Menu.



Accessing Available Pointer Options through a Pointer Setter's Dropdown Menu.



Showcasing Array Options Using an Array Setter's Dropdown Menu.

Figure 3.8: Interactive Features of Blockly-C UI: Code Representation and Data Type Handling

One last feature that certain blocks possess is a clickable blue square in the top left corner, marked with a star icon. This icon is an indicator that the block is mutable, meaning its structure can be altered or "mutated". The `"if"` block stands out as an illustrative example. By default, this block provides an input for condition expression and an encapsulated connection for inner statements, mirroring the familiar `if(condition){statements}` syntax in C. When users engage with the star icon on any mutable block, a window emerges. This window is called "mutator UI" or "mutator window". This window serves as an interactive editing interface for the block's structure. The mutator UI typically manifests as a separate, smaller workspace that resembles the main Blockly workspace, but with a more streamlined design tailored to the mutable block's components. It floats above the primary Blockly canvas, ensuring it is the focal point of interaction. On the left of this window is a list of available blocks, while the right side displays a mini workspace featuring an undeletable block that contains only an encapsulated connection type. When working with the `"if"` block, for instance, the left side presents two selectable blocks: `"else if"` and `"else"`. Users can drag these components and integrate them into the undeletable block on the right side, leading to the mutation of the original `"if"` block. As these additions are made, the primary block dynamically reshapes, reflecting the user's selections (figure 3.9). Another example of mutational block is the 'Variable Declare' block. This allows users to add multiple variable declarations in a single line of code, separated by commas, mirroring the C syntax, as illustrated in Figure 3.10.

While Blockly-C provides extensive customization options for blocks, it doesn't grant unfettered freedom. The platform ensures that modifications align with the syntax of the C language. As an example, you cannot incorporate more than one 'else' condition within an `"if"` block as this would breach C's syntactical norms. Through such deliberate constraints, Blockly-C supports users in creating accurate and syntactically correct code, a feature that is invaluable, particularly for those just embarking on their programming journey.

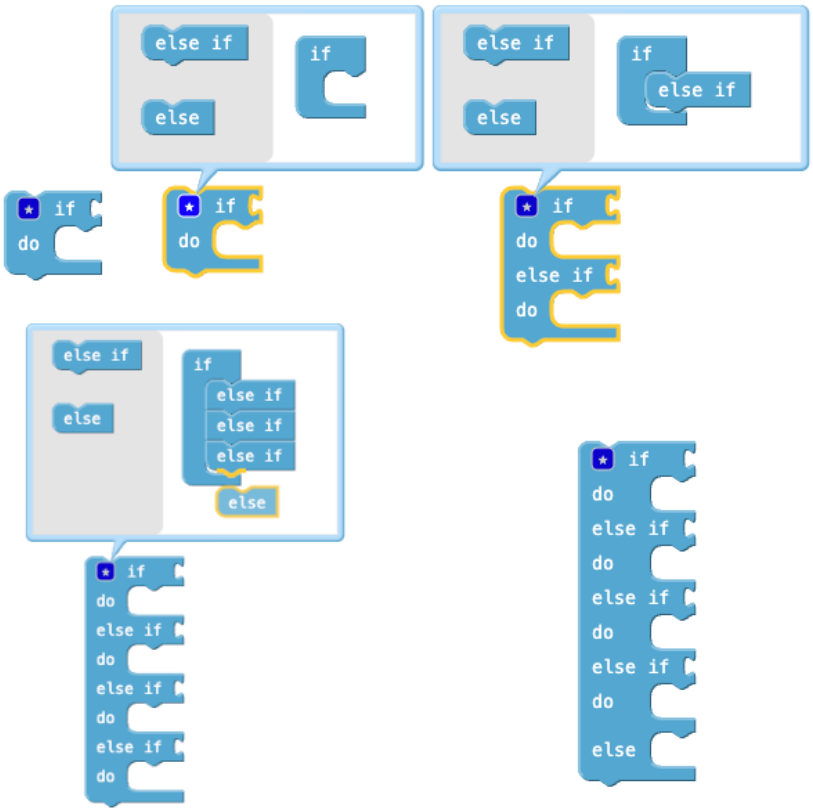


Figure 3.9: Mutational stages of "If" block.

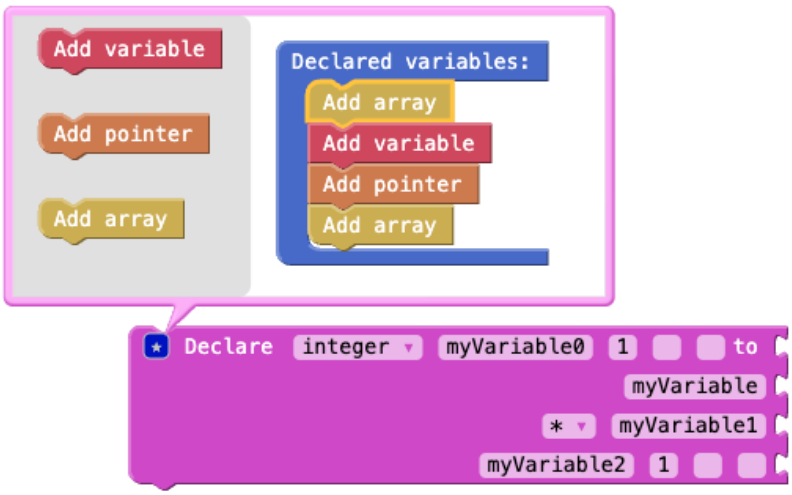


Figure 3.10: Mutability Demonstrated: The 'Variable Declare' Block with Multiple Variable Declarations in single line of code.

Users can effortlessly remove any created block they no longer need by dragging it to the trash icon situated at the screen's bottom right corner (figure 3.11).

Figure 3.11: How the user deletes a block. It's noticeable that the trash bin lid opens, signaling its responsiveness to the user's action.

In Blockly-C, right-clicking on a block reveals a context menu that provides a suite of options to enhance the user's interaction with the block. The 'Duplicate' option creates a clone of the selected block, retaining its current configuration and any nested blocks it contains. Through the 'Add Comment' option, users can input comments, which are then incorporated into the generated code to offer context or clarification. If a block already contains a comment, the 'Remove Comment' option allows for its removal.

The 'Inline Inputs' feature offers a cosmetic adjustment, enabling users to switch the visual display of block inputs between external and inline styles without affecting the block's actual functionality. For a more streamlined view, the 'Collapse Block' option minimizes a block to a concise line, reminiscent of hiding code snippets within curly braces '{}' in many code editors. The 'Disable Block' feature, especially handy during debugging or code experimentation, grays out the block, indicating to Blockly-C that it should be excluded during code generation. Importantly, this action preserves the block's data, allowing for easy reactivation later. An alternative method to remove a block is provided by the 'Delete Block' option, which lets users remove a block without dragging it to the trash bin. Lastly, for users requiring more details about a block's functions, the 'Help' option delivers notes and guidance specific to the block in focus. If the selected block pertains to variable declaration, the context menu introduces an additional option titled 'Create get *variableName*'. Upon selection, this option swiftly generates a getter block tailored specifically for the declared variable. This intuitive feature serves as a convenient shortcut, further streamlining the process of crafting the desired code (figure 3.12).

Figure 3.12: Context Menu Options Displayed for a Variable Declaration Block in Blockly-C.

To the right of the workspace lies the dynamic 'code' component. As users piece together blocks and create logical structures, this section actively and consistently refreshes to showcase the corresponding C code that's being generated. This real-time translation bridges the visual world of block-based programming with the textual realm of the C language. Not only does this

feature enhance the user's grasp of how block structures translate to actual code, but it also presents the C syntax in a neatly formatted and color-coded manner. This design aids in clarity, making it intuitive for users to understand the relationship between their block arrangements and the resulting C code, fostering a deeper comprehension of the programming process (3.13, 3.14).

Code

```c
#include <stdio.h>

void multiplyMatrices(int first[1][10], int second[1][10 ], int result[2][1
void getMatrixElements(int matrix[1][10], int row, int column);

  int myVariable;
int main(int argc, char** argv) {
  return 0;
}

void getMatrixElements(int matrix[1][10], int row, int column) {

  printf("\nEnter elements: \n");
  for(int i = 0;i < row;++i){
    for(int j = 0;i < column;++j){
      printf("Enter a%d%d", (i + 1), (j + 1));
    }
  }
}

void multiplyMatrices(int first[1][10], int second[1][10 ], int result[2][1

  printf("\nEnter elements: \n");
  for(int i2 = 0;i2 < r1;++i2){
    for(int j2 = 0;j2 < c2;++j2){
      result[1][2] = 0;
    }
  }
  for(int i3 = 0;i2 < r1;++i2){
    for(int j3 = 0;j2 < c2;++j2){
      for(int k = 0;k < c1;++k){
        result[1][2] += first[0][2];
      }
    }
  }
```

Figure 3.13: Real-time C Code Generation Display in the 'Code' Component.

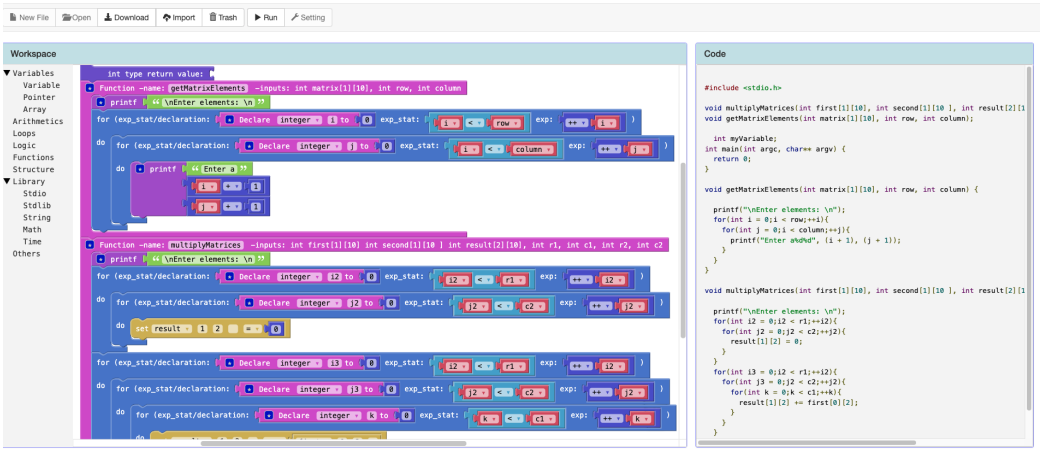


Figure 3.14: Total view of real-time C Code Generation Display in the 'Code' Component.

Above the workspace, an intuitive toolbar extends across, equipped with a series of user-friendly buttons: 'Download,' 'Import,' 'Trash,' and 'Run.' These buttons serve distinct purposes, further magnifying Blockly-C's versatility and user-centric approach.

The 'Download' button is particularly beneficial for users who wish to archive or share their work. By clicking it, they can effortlessly save their crafted C code directly to their computers, bridging the gap between the Blockly-C environment and external platforms. This means that users can not only build their programs within Blockly-C but also execute, share, or modify them using other platforms or applications.

On the other hand, the 'Trash' button offers a reset function, allowing users to declutter their workspace by removing all added blocks. However, the main block remains consistent and undisturbed, ensuring that the foundational structure is always in place. This thoughtful inclusion ensures that while users have the flexibility to start from scratch, they never lose sight of the primary framework, making the process less daunting.

The 'Import' and 'Run' functionalities, integral components of Blockly-C, have not been delved into here. However, a comprehensive exploration of their dynamics and applications will be presented in subsequent chapters, underscoring their pivotal roles within the platform.

Conclusively, the Block-to-C Code Conversion utility of Blockly-C embodies the platform's unwavering commitment to providing an integrated

and intuitive programming experience. Marrying visual clarity with real-time feedback, complemented by a suite of multifaceted tools, Blockly-C emerges as an indispensable ally for individuals navigating the intricate terrains of C programming.

3.3 C-to-Block Representation

This section delineates the reverse process from the one we previously discussed. Instead of transitioning from the graphical block interface to plain text C code, we'll delve into the functionality that transforms plain C code—sourced from a .c file—into a graphical Blocks UI.

Positioned within the top bar, amidst other buttons such as download and trash, you'll find the "Import" button. When activated, this prompts the web application to launch a standard file selection window, allowing the user to browse and select a file from their local machine. It's crucial to note that only files with the ".c" extension can be selected for this process (figure 3.15).

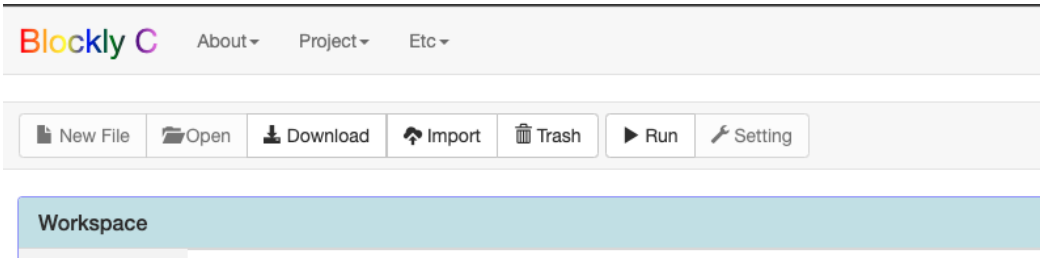


Figure 3.15: Top bar of Blockly-C.

Upon selecting the desired C file, Blockly-C takes the reins. The initial step involves compiling the contained C code to ascertain its syntactical and structural integrity according to established C language standards.

Should the .c file contain any syntactical or structural errors, a notification will be immediately generated at the interface's topmost section. This alert comprises a precise error message, delineating the specific nature and location of the detected discrepancy, providing the user with the necessary feedback for rectification. A representative example of an error message is illustrated in Figure 3.16

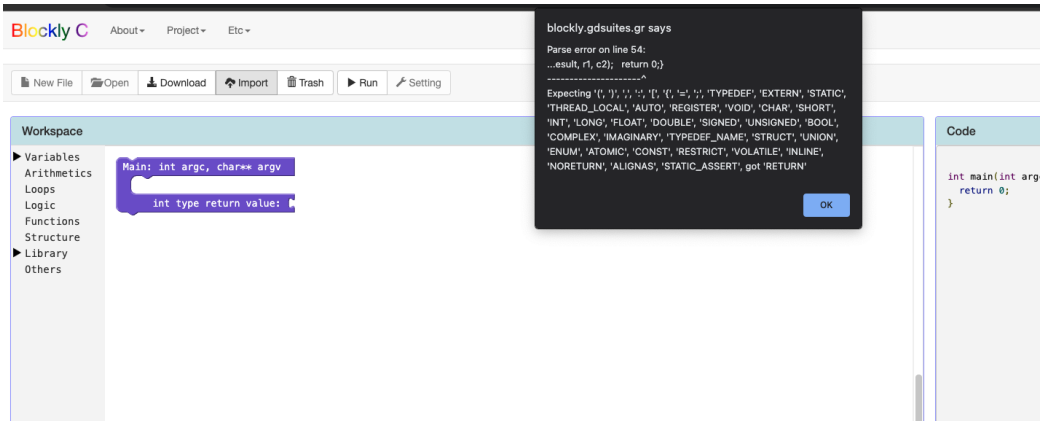


Figure 3.16: Example of Error Message Prompted by a Faulty .c File Upload.

In cases where the code is free from errors and successfully compiles, Blockly-C proceeds to the translation phase. This involves converting the C code into its equivalent Blockly graphical representation. This transformation ensures a one-to-one mapping between the code's elements and their respective graphical blocks. Consequent to this translation, the workspace, as detailed in the preceding chapter, becomes populated with the equivalent blocks, visually representing the content of the uploaded C code.

This functionality not only serves educational purposes by enabling learners to better grasp C grammar and syntax but also proves indispensable to seasoned C programmers. For the seasoned programmer, it's a fresh perspective, unraveling even the most complex codes into visually appealing blocks, making debugging or refining an easier endeavor. It's not uncommon in the coding world to stumble upon vast codebases with minimal documentation, left behind by former colleagues or predecessors. Here, Blockly-C can be a real game-changer, making such daunting tasks significantly more manageable.

But the benefits of this tool aren't restricted to the programming community. Imagine a scenario where administrative staff, despite their non-technical backgrounds, can comprehend code. Blockly-C's interface potentially demystifies the realm of coding, making it accessible even to those who lack a traditional coding background. Through this, the quintessential coding experience can be shared beyond its native community, even reaching the hands of administrative professionals.

This functionality serves as a conduit, effortlessly connecting traditional

text-based programming with Blockly-C's graphic-based block environment, and highlights the platform's commitment to versatility and adaptability, catering to both conventional coders and visual learners.

3.4 Code Execution and Memory Visualization

The step-by-step code execution and memory visualization is described in this section. Beyond the existing functionalities provided to the user, such as the visualization of programming through graphic tiles, the ability to compile, import external code, and convert it to graphics, an additional feature is introduced. Blockly-C now offers the user the capability to execute their created code step by step. Additionally, one can navigate forward and backward through the execution of each line of code. Simultaneously, during this execution, there is a live graphical representation of the program's stack and heap memory. Also, all output results directed to the console are displayed.

Having established the broader functionalities, let's delve deeper into the execution mechanism. The "Run" button, which is positioned on the top bar, facilitates this entire process. When pressed, the user is transitioned to the "Visualization" page. Central to the Visualization page is an integrated version of the PythonTutor web application, an esteemed platform recognized for its proficiency in executing C code. When the 'Run' command is initiated, the Blockly-C interface transmits the created or imported user's code for processing within the PythonTutor framework. The visualization page is illustrated in Figure 3.17.

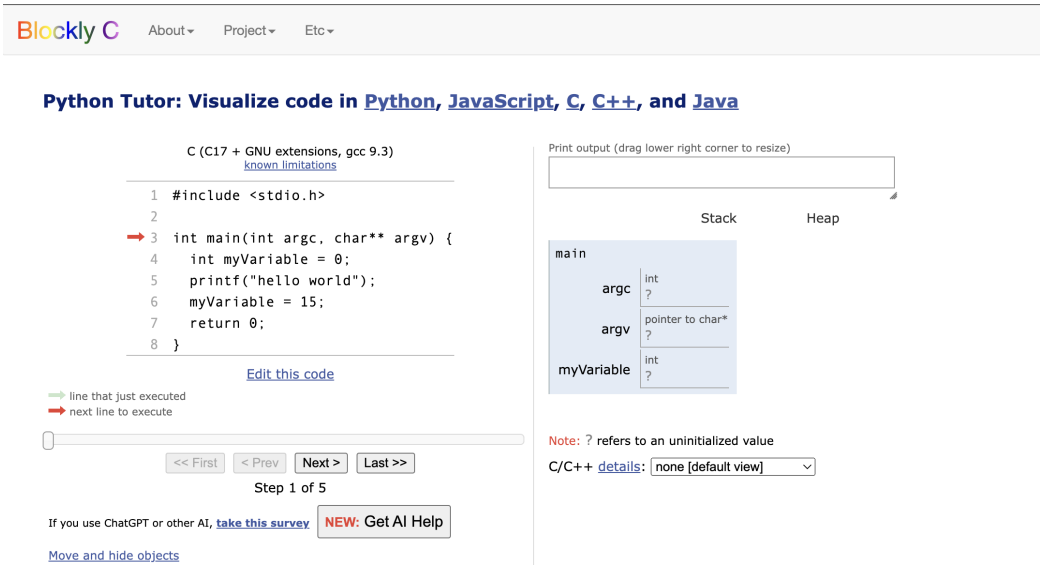


Figure 3.17: Visualization Page Interface Overview.

The Visualization page divides its display into two sections. The left section is the code pane, which is dedicated to displaying the user's code in a clear and legible manner. Here, each line of the code is presented without the usual syntax highlighting found in many editors, maintaining a consistent, monochromatic presentation. This serves to draw the user's attention primarily to the code's logic and structure, undistracted by varying colors.

Integral to the Code Pane's function during code execution are two distinctive pointers: a red pointer and a green one. The red pointer indicates the next line of code that is set to be executed. This provides users with an anticipatory cue, guiding their focus to upcoming operations. On the other hand, the green pointer marks the line of code that has just been executed, allowing for immediate reflection on the code's most recent operation. Together, these pointers play a pivotal role in visualizing the step-by-step progression of the code's execution (figure 3.18).

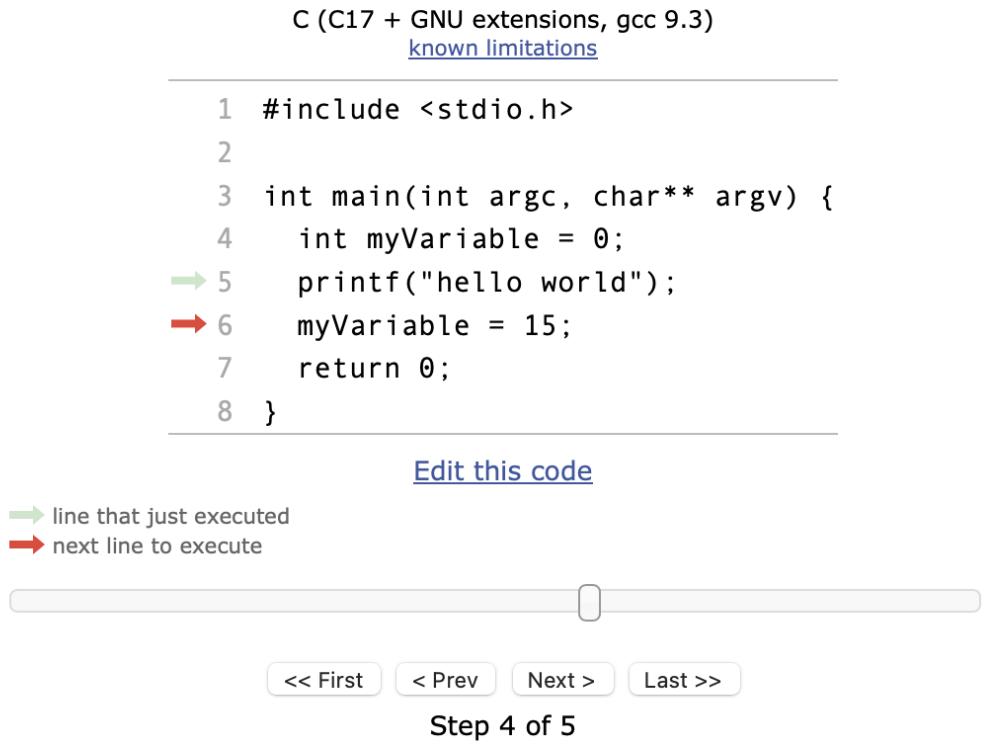


Figure 3.18: Code Pane Overview.

Directly below the main body of the code, there is an “Edit this Code” button. This is designed to offer users a straightforward mechanism to make adjustments or corrections to their code. Clicking on it redirects users to the page where they can make desired changes, ensuring that the process of refining and re-visualizing the code remains seamless and intuitive. In pressing the “Edit this Code” button, users are not redirected to the initial Blockly-c’s page, which utilizes graphical blocks for code editing. Instead, they are taken to PythonTutor’s conventional code editor where code modifications are made through traditional typing. If a user wishes to return to Blockly-c’s graphical code editor, they simply need to use the browser’s back button. Notably, there’s no risk of losing any progress. Upon selecting the "run" option, Blockly-c automatically saves the user’s work. Thus, when navigating back, the platform efficiently reloads the previously constructed code, enabling users to make any necessary adjustments without the fear of

forfeiting their work.

In essence, the Code Pane provides an environment where the dynamics of code execution are foregrounded, assisting users in comprehending the intricate behaviors and operations of their written code.

Within the interface of Visualization page, positioned right below the code pane, is a distinct set of Control Buttons. These are meticulously crafted to grant users the capability for a comprehensive, step-by-step walkthrough of their code's execution process.

The "First Button" serves as a reset tool, allowing users to leap right back to the initial stage of the code execution. It's a foundational touchpoint, setting the stage for users to embark on their walkthrough journey. Contrasting this is the "Last Button", a navigation tool that instantaneously transports users to the final step of the code execution, showcasing the end state after all operations are completed.

The "Prev" or "Previous" Button functions as a one-step backward navigation utility. It facilitates users to retract and review the operation carried out in the immediate preceding step. On the flip side, the "Next Button" propels the code execution forward to the immediate subsequent operation, letting users progress and observe upcoming actions.

Directly beneath these buttons, PythonTutor offers a status notification, often presented in the format "Step 1 of 5". This is more than just a message; it's a compass, providing real-time orientation to users about their current position within the execution journey. The first numeral indicates the current step, while the second represents the total steps in the execution process.

Above the Control Buttons, users will notice an interactive Progress Bar. This isn't just a passive visual representation of the user's progression. It comes equipped with a draggable button, allowing users to fluidly advance or rewind through execution steps. This tactile feature encapsulates the functionalities of both the "Prev" and "Next" buttons into a continuous navigation tool.

On the right section of Visualization page there are placed the Output page and the Visualization pane. Situated at the top right corner of the PythonTutor interface, the Output Pane serves as a real-time mirror to the console of a traditional Integrated Development Environment (IDE). As the code progresses step-by-step, any statements designed to produce an output, such as printf in C, get immediately reflected in this pane. The Output

Pane's directness and immediacy aid in creating a feedback loop, enabling users to correlate their code's operations with the resultant outputs (figure 3.19).

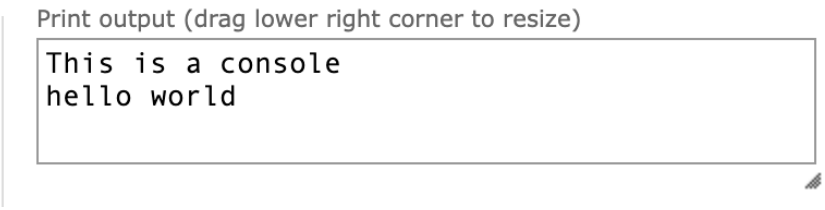


Figure 3.19: Output Pane Overview.

Positioned directly beneath the Output Pane on the right side of the screen is the **Visualization Pane**. This part is the epitome of PythonTutor's teaching methodology. The Visualization Pane offers an intuitive and graphical representation of the memory state during the execution of a C program (figure 3.20).

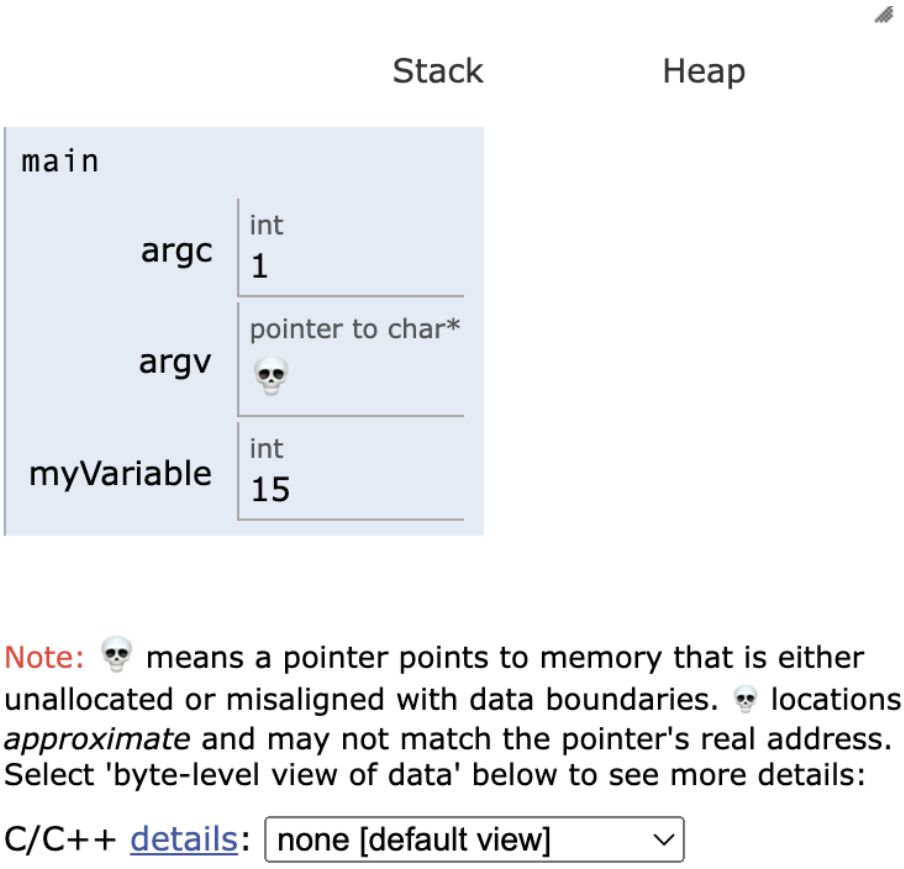


Figure 3.20: Visualization Pane Overview.

Scalar variables, like integers or characters, are depicted straightforwardly in the Visualization Pane. They appear as labeled boxes. The label displays the variable name, while the box's content showcases the variable's value. For instance, a variable `int x = 5;` will be illustrated as a box labeled "x" containing the value "5" (figure 3.21).

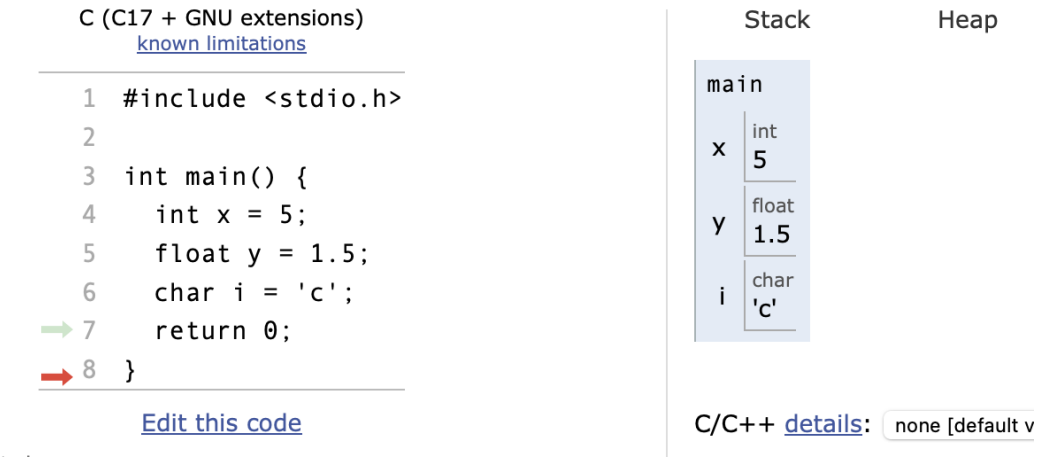


Figure 3.21: Presentation of scalar variables in Visualization Pane.

Arrays in C are another essential element presented in this pane. They are displayed as contiguous blocks. This representation is symbolic of their contiguous memory allocation in actual computation. Each block or cell is directly associated with an array index. The value at that index fills the cell's content. An array like `int arr[3] = {1, 2, 3};` would manifest as three attached cells with values "1", "2", and "3" respectively (figure 3.22).

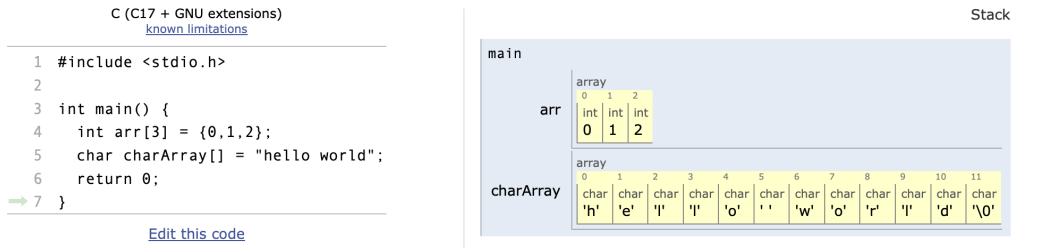


Figure 3.22: Presentation of arrays in Visualization Pane.

Pointers, a defining feature of C, are distinctly illustrated as arrows. When a pointer directs towards a variable or an array, it is visualized as an arrow that originates from the pointer's label and targets the memory location (or box) it references. For a pointer declaration like `int *p = &x;` where `x` is an integer, there would be an arrow from the label "p" pointing towards the box representing "x" (figure 3.23). When a pointer is derefer-

enced, its target location gets highlighted or underlined, providing a visual cue of the memory being accessed.

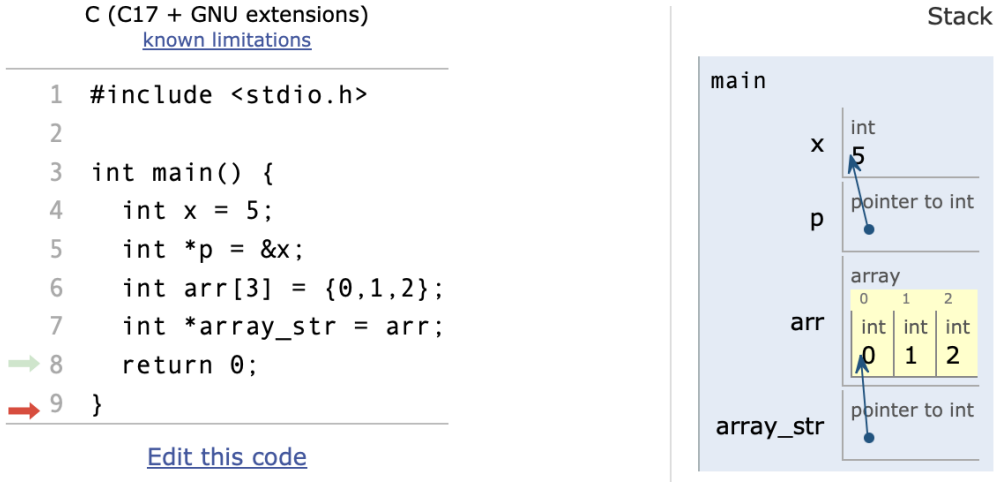


Figure 3.23: Presentation of pointers in Visualization Pane.

For structures in C, the Visualization Pane employs a slightly more complex visual. Structures are depicted as larger boxes. These boxes are partitioned into segments, where each segment corresponds to a distinct data member of the struct. Take, for instance, a struct `struct Point { int x; int y; };` and its instance `Point P = {2, 3};`. A large box labeled “P" would be partitioned into two: one segment labeled “x" with value “2" and another labeled “y" with value “3" (figure 3.24).

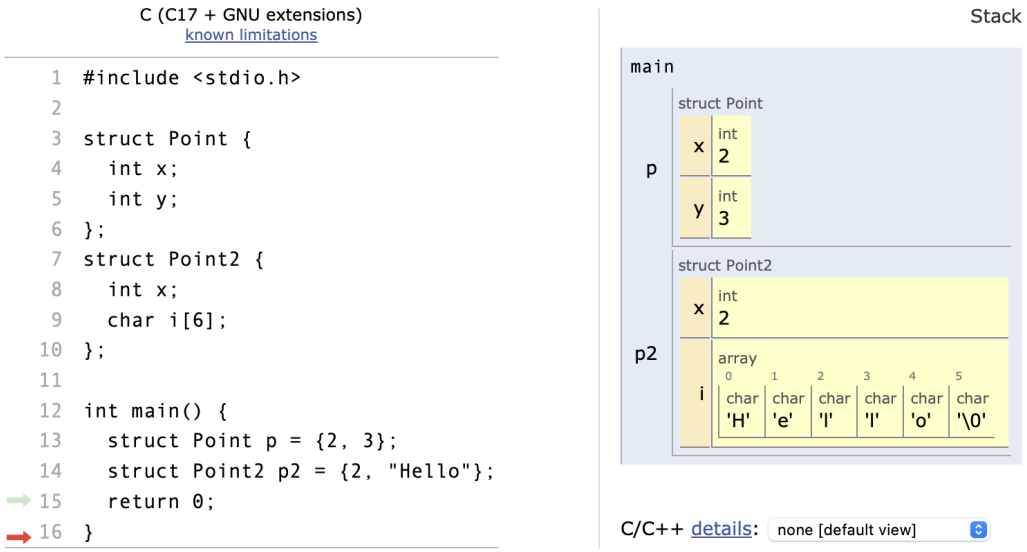


Figure 3.24: Presentation of structures in Visualization Pane.

Another crucial aspect is the visualization of dynamic memory. Memory regions that are allocated using functions like `malloc` or `calloc` are distinctly marked. Those two functions play a crucial role in many programs. Python Tutor, understanding the importance of this, provides a unique visualization for this kind of memory. When such memory allocation functions are used, the memory blocks allocated dynamically are displayed prominently within the heap section of the visualization pane (figure 3.25). This distinction subtly reminds the programmer of their duty to manage such memory, including the necessity to deallocate using `free`.

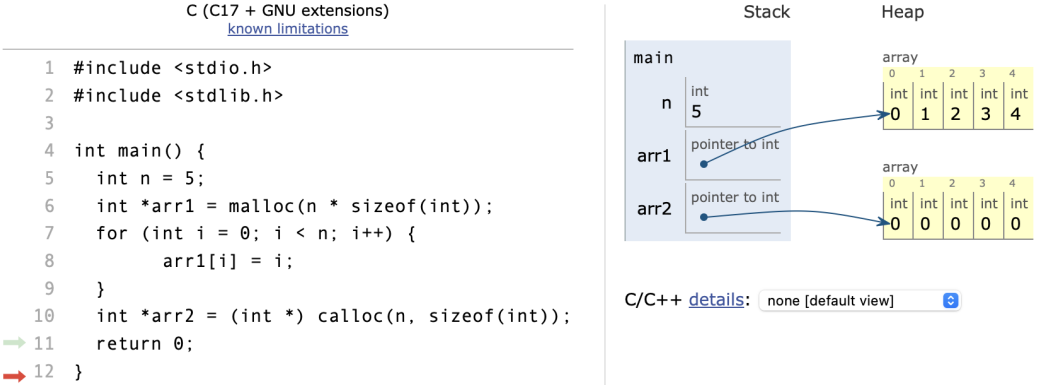


Figure 3.25: Presentation of dynamic memory allocations in Visualization Pane.

Function calls introduce an added layer of visualization complexity. With every function invocation, a new frame is introduced in the visualization, signifying a new stack level. This frame encompasses local variables, parameters of the function, and, at times, the origin or location of the function call (figure 3.26).

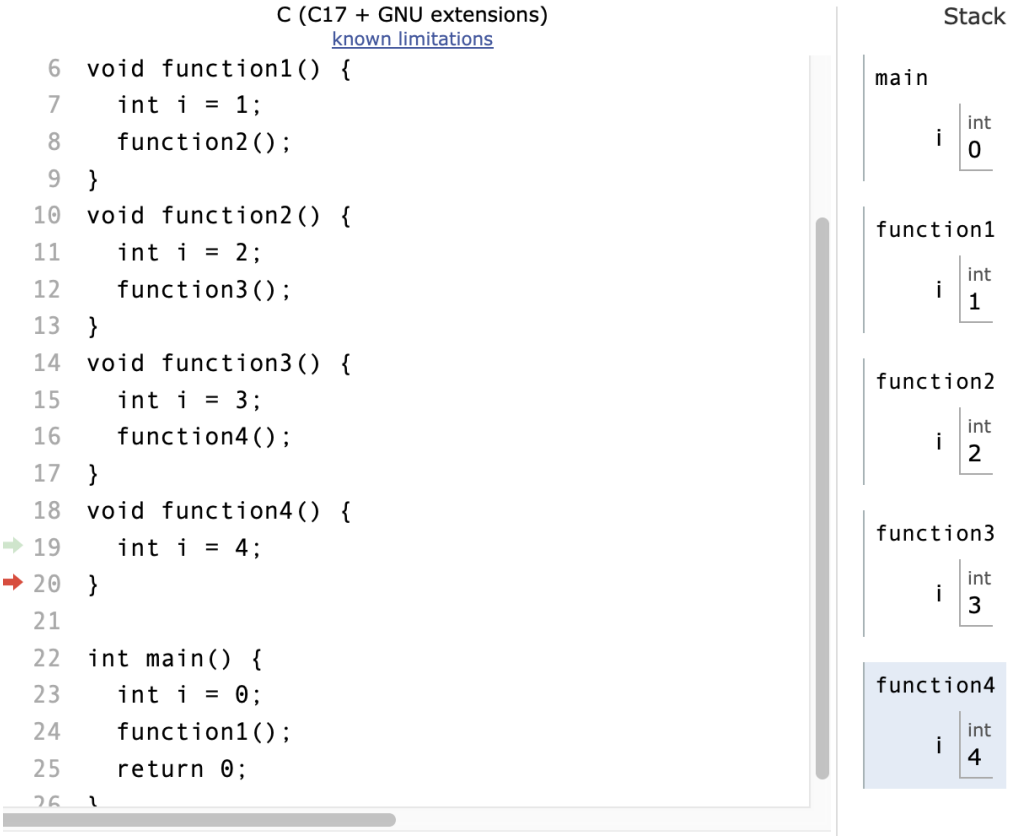


Figure 3.26: Presentation of functions in Visualization Pane.

Finally, to cater to miscellaneous variables or states, Python Tutor incorporates specialized indicators. For instance, uninitialized variables may be grayed out or might carry a unique marker. Similarly, pointers that are null or uninitialized might be portrayed differently, such as with dashed arrows, indicating they currently don't reference any valid memory location.

Overall, PythonTutor's visualization approach is a masterclass in clarity. The benefits of such an execution and visualization feature are manifold, especially for novice programmers. For beginners, understanding abstract concepts of memory management in C can be challenging. The visual representation of memory operations makes these concepts more tangible and comprehensible. By observing the step-by-step execution of code, users can quickly identify logical or runtime errors. As the code progresses, changes in memory allocation and management are graphically represented, offering

users a visual narrative of how data is manipulated, stored, or retrieved in system memory. This navigation ability enhances debugging efficiency to another level. This visualization also aids those transitioning from academic understanding to practical application by making theoretical knowledge actionable. Regular interaction with this feature fosters analytical thinking, empowering users to predict and strategize memory management in more complex coding scenarios.

In conclusion, Blockly-C's integration of the PythonTutor application augments the user's coding experience, deepening their comprehension of C programming nuances. By seamlessly connecting code creation with its tangible outcomes, Blockly-C emerges as a vital asset for both educational domains and the broader programming world.

Chapter 4

Implementation

4.1 Introduction

The "End User Experience" chapter provided an overview of the system's user interface and its interaction points from an end-user perspective without delving into the technical intricacies. This chapter, "Implementation," shifts the focus to the technical details underpinning the system.

In the subsequent sections, we will discuss the methodologies, algorithms, and processes implemented in the development of the system. We will start with the mechanisms behind the "Block-to-C Code Conversion," detailing the techniques employed to translate block-based inputs into C code. Conversely, the "C-to-block Representation" will explain the approach taken to render C code back into its block-based form. The chapter will culminate with the "Code Execution and Memory Visualization" section, which will outline how the system handles the dynamic representation of code execution and memory operations.

This chapter aims to provide a clear and detailed insight into the technical decisions, challenges, and solutions that were integral to the development and functionality of the system.

4.2 Block-to-C Code Conversion

As discussed in previous chapters, the open source library of *Blockly* was used in order to develop the final system. *Blockly* provides a comprehensive set of essential UI functionalities and utility functions. These components

enable the creation of customized web applications with workspaces housing blocks of your choice, which can be translated into text according to your preferences. While *Blockly* provides support for several modern languages, including `JavaScript`, `Python`, `PHP`, `Lua`, and `Dart`, it does not support the C language. The challenge with C arises from its typed nature, wherein variables must be declared with specific data types like `int`, `float`, or `char`. Since *Blockly*'s architecture was not designed with declarative variables in mind, a straightforward block-to-C code conversion was not feasible. Addressing this limitation it was required to be made significant modifications on the *Blockly*'s core. These changes enabled the framework to support the C language and expanded its potential applications. Before diving into what changes were required to fully support the C language in *Blockly*, let's talk about how *Blockly* was initially implemented.

Blockly's core is primarily written in `JavaScript`, leveraging a combination of `HTML` and `SVG` (Scalable Vector Graphics) for rendering its blocks. The utilization of `SVG` is crucial, as it ensures that blocks are displayed with crisp clarity at various zoom levels. *Blockly*'s codebase is designed with modularity in mind, meaning you can customize or swap out significant portions of its functionality without touching other parts of the system. Internally, *Blockly* encapsulates its core functionality within the `/core` directory, where numerous `JavaScript` files control distinct aspects of its operation. For instance, the `core/events/` directory manages *Blockly*'s event-driven architecture. User interactions within *Blockly*, such as moving a block or altering a field, trigger events that are handled by this module. One notable aspect of *Blockly* is its ability to seamlessly transform visual block configurations into textual code. For example, when snapping blocks together in *Blockly*, the platform operates behind the scenes to generate textual code. It translates the arrangement of blocks into code that corresponds to the user's intended logic or program, exemplifying *Blockly*'s powerful capabilities in code generation.

All the logic for the *Blockly* workspace is centralized within the `core/workspace.js` file. *Blockly* uses a rendering engine to draw the blocks. The renderers dictate how the blocks look, how they respond to being dragged, and more. Everything regarding the rendering procedure is placed inside the `core/renderers/`. The block renderers in *Blockly* play a pivotal role in translating each declared block within the project into an `SVG` format

as users manipulate and drag them within the workspace. These renderers ensure that the visual representation of the blocks remains synchronized with user actions.

All the pre-declared blocks are placed in another folder completely separated from the core functionality. They are inside the folder `blocks/`. These pre-existing blocks serve the purpose of visually representing code in languages already supported by *Blockly*. However, they are not tailored for compatibility with the C language. For this reason, it was necessary to delete everything and start afresh to develop a set of blocks that would align seamlessly with the specific requirements of the C language. To fully support the C language's capabilities within the application, it became imperative to design individual blocks for each distinct command in C. In total, 88 new blocks were meticulously crafted and integrated.

Let's delve into the process of creating a block in Blockly, exploring the detailed steps and considerations involved. Blockly is object-oriented. For instance, every block type is a class, and instances represent individual blocks on the workspace. In Blockly, creating a new block involves a systematic process that primarily deals with defining its structure, appearance, and behavior. Initially, one needs to define the block's specifications in a JavaScript Object Notation (JSON) file or directly in JavaScript. All the blocks were developed using JavaScript. Consequently, the subsequent examples will be presented in this format.

To craft a new block in *Blockly*, you have to define a new object within `Blockly.Blocks`. For instance, introducing a block can be represented as `Blockly.Blocks.customBlock={...}`. Here, `customBlock` signifies the designated name of the block, with all its attributes and methods encapsulated within the subsequent curly braces. For each block, it is obligatory to have an `init` function. The `init` function within a block declaration in Blockly serves as the foundational core for the visual and functional characteristics of a block. When defining a custom block in Blockly, the `init` function is imperative, acting as the constructor that initializes the block when it's created. Within this function, you define the block's appearance, inputs, connections, tooltips, colors, and more.

Diving deeper, within the `init` function, one of its primary tasks is to

define and populate the block's user interface with appropriate content. This entails the inclusion of specific inputs, which form the building blocks of the block's interface in Blockly. There are three primary types of inputs: value input, statement input, and dummy input.

The *value input* serves as a placeholder for blocks that can return a value. When one block is connected to another block's value input, the latter block retrieves the value produced by the former. Programmatically, this can be appended to a block using the method `this.appendValueInput("NAME")`, with `"NAME"` acting as a unique string identifier. This type of connections is used to assigning values to variables or expressions. On the other hand, the *statement input* caters to situations where a sequence of blocks, rather than just one, needs to be nested inside another. This is typically seen in loops or conditionals where multiple statements can be enclosed within the curly braces. To incorporate this type of input into a block, the method `this.appendStatementInput("NAME")` is used, with `"NAME"` serving as a unique identifier.

The third type, the *dummy input*, stands apart from the rest as it is used purely for adding visual or textual content, without the intention of connecting to other blocks. This is especially useful for integrating text labels or dropdown menus. The method `this.appendDummyInput()` is called upon to integrate a dummy input. By harnessing the capabilities of these input types, developers can craft a versatile and intuitive block interface in the *Blockly* workspace. In Figure 4.1, the three distinct input types in Blockly are vividly demonstrated, with the corresponding JavaScript code on the left and the resultant visual blocks on the right.

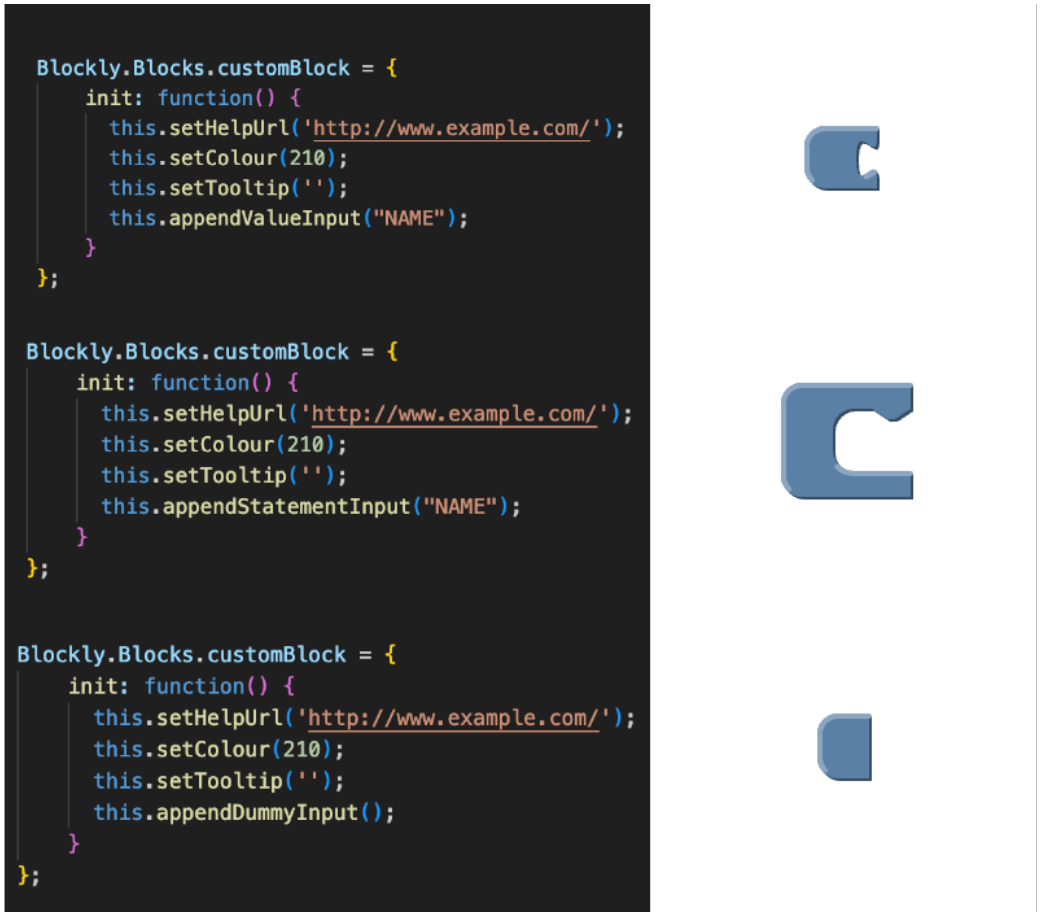


Figure 4.1: Illustration of Input Types in Blockly with Corresponding JavaScript Code and Visual Blocks.

Expanding our examination of *Blockly*'s capabilities, we come across another pivotal facet: fields. Fields play a crucial role in determining the specifics of a block's visual representation, ensuring the block communicates its function and potential values to the end-user. In *Blockly*, fields primarily revolve around user-modifiable values, text displays, images, and dropdowns. You can add multiple fields within each of the inputs we previously explored.

The `"text field"` is one of the simplest forms, and as the name suggests, it's used to present textual content on a block, primarily for explanatory or descriptive purposes. This can be added with `this.appendDummyInput().appendField("YourLabel")`. An example that illustrates how this code line

is converted to a SVG can be shown in the figure 4.2 This field is used in the `"math_number"` block in order to allow users input any number. The `"text input field"` lets users input values directly. For instance, to capture numerical or string data, one might use a `FieldTextInput`, added by `.appendField(new Blockly.FieldTextInput("default"), "FIELDNAME")`, where `"default"` is the default value and `"FIELDNAME"` is a unique identifier (figure 4.2). This field type is employed in the variable declaration block, allowing users to name their variables as per their preference. Another versatile field is the *dropdown field*. It provides users with a predefined set of options. This can be achieved via `this.appendDummyInput().appendField(newBlockly.FieldDropdown([["Option1","OPTION1_VALUE"],["Option2","OPTION2_VALUE"]]),"FIELDNAME")`. Here, the array lists the display name and corresponding values for the dropdown options (figure 4.2). This particular fields was used in the variable declaration block, to let users pick the variable's type (like int, float, char). Lastly, to make blocks more visually engaging or intuitive, one can integrate *field images*. This type of field wasn't used in any block.

Figure 4.2: Illustration of Field Types in Blockly with Corresponding JavaScript Code and Visual Blocks.

The color of each block can be specified with the function `this.setColour()`, which receives a value from 0 to 360.

Additionally, the function can set specific connection points, which define the compatibility and potential interactivity with the parent blocks. The `setOutput` method, when employed, equips a block with a puzzle-like tab, indicating its ability to act as a value provider. This can be set using `this.setOutput(true, "Type")`, with "Type" specifying the data type of

the output. Once equipped with this tab, the block can be seamlessly connected to another block's 'value input'. Meanwhile, the `setNextStatement` method provides a notched bottom edge to a block, enabling it to attach to another block placed directly beneath. By invoking `this.setNextStatement(true, "Type")`, a logical sequence of commands or statements can be facilitated, where "Type" defines the type of blocks that can connect below. Conversely, the `setPreviousStatement` method introduces a notched upper edge, allowing a block to connect upwards to a block positioned directly above, achieved using the command `this.setPreviousStatement(true, "Type")`.

A critical detail to note is that a block cannot simultaneously have both an output connection (from setOutput) and top or bottom connections. This enforces clarity in block design, ensuring each block has a distinct purpose and function. All these connection types and their nuances are vividly captured in Figure 4.3.

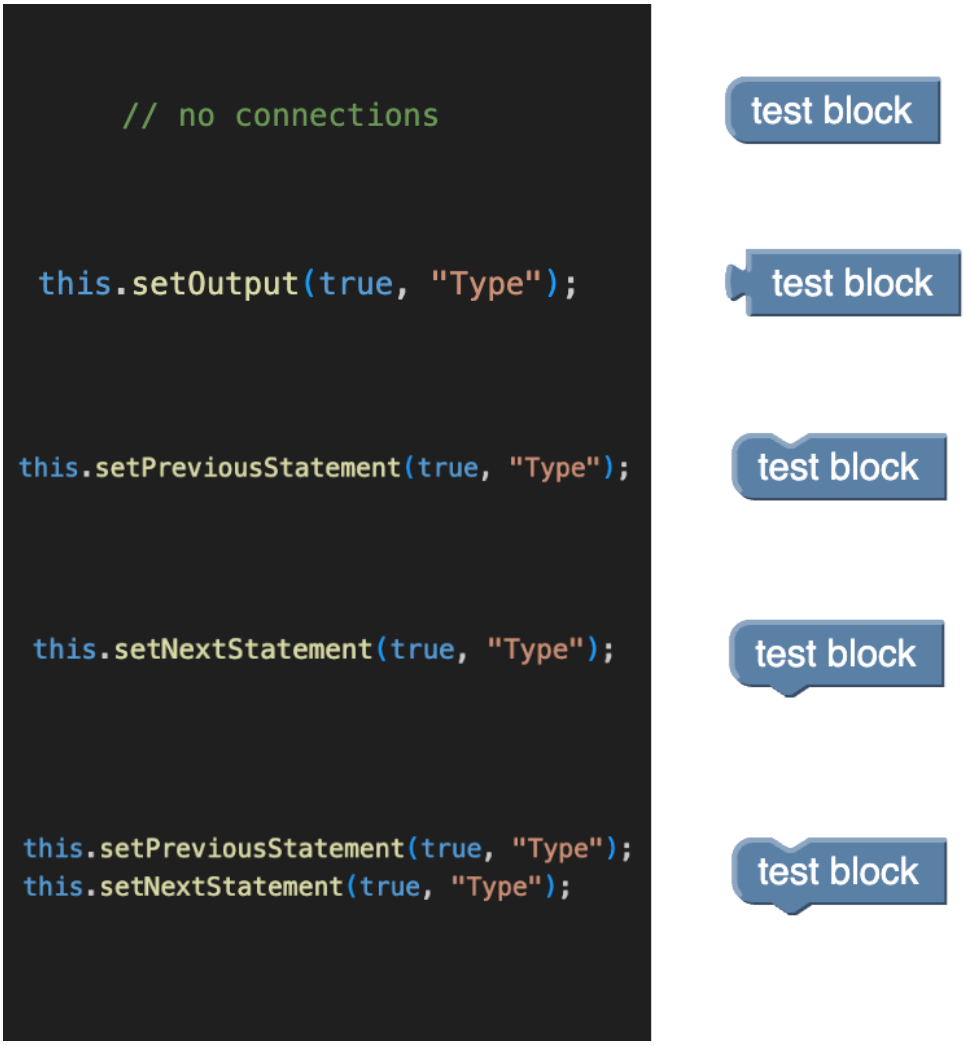


Figure 4.3: Illustration of Connection Types in Blockly with Corresponding JavaScript Code and Visual Blocks.

In addition to establishing the primary connections in *Blockly*, it's crucial to ensure that only the appropriate types of blocks can be connected together. *Blockly* provides a mechanism for this through 'checks'. Checks act as constraints, ensuring that a block connection adheres to specific types. When defining a value or statement input, you can set these checks to permit only specific types of blocks to connect. For instance, the code `this.appendValueInput("NAME").setCheck("Number");` allows only blocks

that return a number to be connected. Similarly, for statement inputs, one might use `this.appendStatementInput("NAME").setCheck("Loop");`, ensuring that only loop-type blocks can be nested within.

Further refinements, such as tooltips with `setTooltip` or help URLs with `setHelpUrl`, enhance the user experience, offering guidance or supplementary information.

Except for the `init` function, the `onChange` function also holds significant importance in the Blockly environment. Acting as an event listener, it's triggered whenever there's a change in the block or its surroundings. This change can be due to user interactions, such as connecting or disconnecting blocks, editing field values, or even programmatically modifying the workspace.

Within the `onChange` function, developers have the opportunity to define behaviors and responses tailored to the specific requirements of their block. As an illustration, consider the "variable declare" block that it was designed. The `onChange` function were used to dynamically adjust the validation rules for the block's value input. By doing so, the block can adapt to accept different types of child blocks, based on the variable's type: for instance, boolean versus int or float. This adaptability ensures that users maintain syntactical accuracy in C, reducing the likelihood of errors.

A further exemplification of the `onChange` function's versatility can be seen with the same "variable declare" block. It was intended for this block to seamlessly integrate both outside main functions and within them. This intention posed a challenge regarding the block's connection rules: when the block represents an external declaration, it shouldn't accommodate regular blocks like "while" or "if" conditions, which aren't syntactically valid outside a function in C. Instead, it should only connect with blocks like "main function", "custom functions", or "variable declaration" blocks. However, when placed inside a function, it should be compatible with standard blocks, including loops and conditions. To address this, the `onChange` function continuously updates the block's connection rules, ensuring it adheres to C's syntax norms depending on its placement context.

The `init` and `onChange` functions are the mandatory components for block creation in Blockly. Some basic blocks can be seamlessly defined using just these two functions. However, for intricate constructs like variable declarations, function definitions, and others, additional utility functions are essential. These functions provide the blocks with the specialized behaviors and capabilities necessary to meet their specific roles within the Blockly

ecosystem.

Now let's delve into the code of an actual block that was created in Blockly-C. In the figure 4.4 can be shown the code of the block "variable_declare". This block was previously mentioned and illustrated in figure 3.6. This block's code showcases the essential "init" and "onChange" functions. The "onChange" function, in particular, plays a crucial role; it adjusts the block's connection rules whenever the user alters the variable type, such as switching from integer to float. This dynamic adjustment is a key feature of Blockly-C, as it actively prevents users from making syntactical errors. Alongside these foundational functions, the "variable_declare" block also integrates several utility functions, essential for its unique behavior and functionality as discussed previously.

```
Blockly.Blocks.variables_declare = {
  init: function() {
    var a = [
      [Blockly.Msg.VARIABLES_SET_TYPE_INT, "int"],
      [Blockly.Msg.VARIABLES_SET_TYPE_UNSIGNED_INT, "unsigned int"],
      [Blockly.Msg.VARIABLES_SET_TYPE_FLOAT, "float"],
      [Blockly.Msg.VARIABLES_SET_TYPE_DOUBLE, "double"],
      [Blockly.Msg.VARIABLES_SET_TYPE_CHAR, "char"]
    ];
    this.setColour(350);
    var b = Blockly.Procedures.findLegalName(Blockly.Msg.VARIABLES_DECLARE_DEFAULT_NAME, this);
    this.interpolateMsg(
      'Variable -type: %1 -name: %2 -initial value: %3 ',
      ["TYPES", new Blockly.FieldDropdown(a, null, this)],
      ["VAR", new Blockly.FieldTextInput(b, Blockly.Procedures.rename)],
      ["VALUE", null, Blockly.ALIGN_RIGHT],
      Blockly.ALIGN_RIGHT
    );
    this.setPreviousStatement(!0, arrayPreviousStatementValues);
    this.setNextStatement(!0, arrayNextStatementValues);
    this.setTooltip(Blockly.Msg.VARIABLES_DECLARE_TOOLTIP);
    this.contextMenuMsg_ = Blockly.Msg.VARIABLES_SET_CREATE_GET;
    this.contextMenuType_ = "variables_get";
    this.tag = Blockly.Msg.TAG_VARIABLE_DECLARE
  },
  initVar: Blockly.Blocks.define_declare.initVar,
  getDist: function() {
    return "v"
  },
  getScope: function() {
    if (this.getSurroundParent()) return this.getSurroundParent().getName(this)
  },
  getSpec: function() {
    return null
  },
  getPos: function() {
    return this.getRelativeToSurfaceXY().y
  },
  getTypes: function() {
    return [this.getFieldValue("TYPES")]
  },
  getVars: function() {
    return [this.getFieldValue("VAR")]
  },
  getDeclare: function() {
    return [this.getFieldValue("VAR")]
  },
  renameVar: function(a, b) {
    Blockly.Names.equals(a, this.getFieldValue("VAR")) && this.setFieldValue(b, "VAR")
  },
  customContextMenu: Blockly.Blocks.variables_get.customContextMenu,
  onchange: function() {
    Blockly.Blocks.variablePlaceCheck(this); // this function checks for warning message (ex. Thi
    var a = this.getFieldValue("TYPES");
    0 == a && (a = "int");
    Blockly.Blocks.setCheckVariable(this, a, "VALUE")
  }
};
```


Figure 4.4: Illustration of code for block "variable_declare".

The "controls_for_2" block provides another illustrative example, with its code and user interface depiction presented in Figure 4.5. Unlike the "variable_declare" block, "controls_for_2" incorporates fewer utility functions, primarily featuring the "getName" function. This particular function generates a unique string identifier, incorporating the unique ID assigned to the block upon its initialization within the workspace. This distinctive naming convention is instrumental in differentiating variables, taking into account their scope and placement within the program's structure. Within the "init" function of the block, one can observe the comprehensive set of 'setCheck' conditions applied to each input value of the 'for' block. These conditions are meticulously aligned with the actual syntax rules of the C programming language, ensuring that input values adhere to the correct syntax constraints.


```
Blockly.Blocks.controls_for_2 = {
  init: function() {
      this.setColour(220);
      this.interpolateMsg("for (exp_stat/declaration: %1 exp_stat: %2 exp: %3)",
          ["INIT", "exp_variable_declare expressions for_set_var Variable exp_pointer_set "+
          "Pointer exp_array_set Array exp_printf exp_scanf exp_free exp_exit Boolean Number "+
          "Aster Address Macro Variable INT FLOAT DOUBLE CHAR UNINT NEGATIVE VAR_INT VAR_UNINT "+
          "VAR_FLOAT VAR_DOUBLE VAR_CHAR PTR_INT PTR_UNINT PTR_FLOAT PTR_DOUBLE PTR_CHAR DBPTR_INT "+
          "DBPTR_UNINT DBPTR_FLOAT DBPTR_DOUBLE DBPTR_CHAR".split(" "), Blockly.ALIGN_RIGHT],
          ["COND", "expressions for_set_var Variable exp_pointer_set Pointer exp_array_set Array "+
          "exp_printf exp_scanf exp_free exp_exit Boolean Number Aster Address Macro Variable INT FLOAT "+
          "DOUBLE CHAR UNINT NEGATIVE VAR_INT VAR_UNINT VAR_FLOAT VAR_DOUBLE VAR_CHAR PTR_INT PTR_UNINT "+
          "PTR_FLOAT PTR_DOUBLE PTR_CHAR DBPTR_INT DBPTR_UNINT DBPTR_FLOAT DBPTR_DOUBLE DBPTR_CHAR"
          .split(" "), Blockly.ALIGN_RIGHT],
          ["UPDT", "expressions for_set_var Variable exp_pointer_set Pointer exp_array_set Array "+
          "exp_printf exp_scanf exp_free exp_exit Boolean Number Aster Address Macro Variable INT FLOAT"+
          " DOUBLE CHAR UNINT NEGATIVE VAR_INT VAR_UNINT VAR_FLOAT VAR_DOUBLE VAR_CHAR PTR_INT PTR_UNINT"+
          " PTR_FLOAT PTR_DOUBLE PTR_CHAR DBPTR_INT DBPTR_UNINT DBPTR_FLOAT DBPTR_DOUBLE DBPTR_CHAR"
          .split(" "), Blockly.ALIGN_RIGHT],
          Blockly.ALIGN_RIGHT);
      this.appendStatementInput("DO").appendField(Blockly.Msg.CONTROLS_FOR_INPUT_DO);
      this.setPreviousStatement(!0, arrayPreviousStatementValues);
      this.setNextStatement(!0, arrayNextStatementValues);
      this.setInputsInline(!0);
      this.tag = Blockly.Msg.TAG_LOOP_FOR;
      this.setTooltip("");
  },
  getName: function() {
      return ['ForLoop_2-' + this.id];
  },
  onchange: Blockly.Blocks.requireInFunction
};
```

Figure 4.5: Illustration of code for block "controls_for_2".

Transitioning to the topic of mutable blocks in Blockly, it's imperative to understand their foundational principles. In Blockly, crafting a mutable block involves a set of procedures distinct from those of a standard block. A mutable block is dynamic in nature; its composition can be adjusted interactively by the user to accommodate varying numbers of inputs or fields. This flexibility sets mutable blocks apart, as they can be tailored to specific needs during runtime. Besides the mandatory `init` and `onChange` functions, constructing a mutable block necessitates the inclusion of the `mutationToDom`,

`domToMutation`, `decompose`, `compose` and `saveConnections` functions.

The `mutationToDom` function serves a critical role in preserving the state of a mutable block by saving it in an XML format. Whenever Blockly's environment calls for the serialization of the workspace—be it for saving or exporting—the `mutationToDom` method is invoked for each mutable block present. This function then duly returns an XML element, encapsulating any mutations or alterations the block might have experienced since its initial creation.

Acting as the inverse to texttttmutationToDom, the `domToMutation` function is tasked with restoring a block to its saved state by interpreting its XML representation. When Blockly initializes and loads a serialized workspace, it leverages the `domToMutation` method to comprehend the XML data, originally produced by `mutationToDom`, and reinstates each mutable block to its preserved state.

The `decompose` function plays a pivotal role in the mutable block's adaptability. As previously highlighted, when the mutable icon on a block is selected, a distinct "mutator window" emerges. This interface features a "Workspace Area" on the right, complemented by a "Toolbox" on the left, presenting available blocks for modification. The `decompose` function's responsibility is to translate the block's present state into a set of representative blocks, subsequently positioning them within the "Workspace Area" of the "mutator window". By doing so, it offers users the capacity to adjust the block's "mutational state" to their requirements, enabling intuitive addition, removal, or rearrangement of block components. Basicaly, the `decompose` function instructs Blockly on populating the contents of the 'mutator window' whenever users express the intent to modify a mutable block.

The `compose` function is the opposite of `decompose`. It reconstructs the mutable block according to the alterations made by users in the "mutator UI". Once the users conclude their edits in this interface and proceed to close it, the compose function comes into action. It meticulously scans the state of the blocks present within the mutator workspace and updates the original block, ensuring it mirrors the newly introduced changes.

With the mutable nature of these blocks, maintaining the continuity of connections between the primary block and its associated child blocks during the mutation process is paramount. This is where the `saveConnections` function finds its significance. As users navigate and modify the structure within the "mutator UI", potential disruptions to the original block's connections can emerge. By meticulously mapping each child block within the

mutator UI to its respective component in the main block, this function preserves the integrity of these connections. Subsequently, when the mutator interface is terminated and the compose function updates the primary block, the saved connections are leveraged to ensure the block's structure remains coherent and undisturbed.

Together, these functions provide a robust system for creating and editing mutable blocks, allowing Blockly to offer dynamic, adaptable block structures while ensuring the integrity and consistency of user creations.

An example of a mutable block in Blockly-C is demonstrated in figures 4.6, 4.7, and 4.8. These figures showcase the "library_stdio_printf" block, representing the "printf" function in C. This block's mutable design allows users to add multiple value inputs as needed, accommodating scenarios where printing a variable alongside a string or combining several variables with multiple strings is required. This flexibility reflects the diverse usage of the "printf" function in C programming.

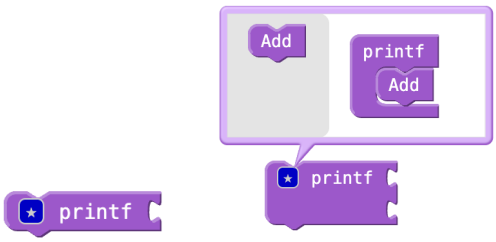


Figure 4.6: Illustration of code for block "library_stdio_printf" (part1).

```
Blockly.Blocks.library_stdio_printf = {
  init: function() {
      this.setColour(280);
      this.appendValueInput("VAR0")
        .setCheck(null).appendField(Blockly.Msg.STDIO_PRINTF_TITLE);
      this.setPreviousStatement(!0, arrayPreviousStatementValues);
      this.setNextStatement(!0, arrayNextStatementValues);
      this.setMutator(new Blockly.Mutator(["library_stdio_printf_add"]));
      this.setTooltip(Blockly.Msg.TEXT_PRINT_TOOLTIP);
      this.tag = Blockly.Msg.TAG_STDIO_PRINTF;
      this.printAddCount_ = 0
  },
  mutationToDom: function() {
      if (!this.printAddCount_) return null;
      var a = document.createElement("mutation");
      this.printAddCount_ &&
          a.setAttribute("printadd", this.printAddCount_);
      return a
  },
  domToMutation: function(a) {
      this.printAddCount_ = parseInt(a.getAttribute("printadd"), 10);
      for (a = 1; a <= this.printAddCount_; a++)
        this.appendValueInput("VAR" + a).setCheck(null).appendField("")
  },
  decompose: function(a) {
      var b = Blockly.Block.obtain(a, "library_stdio_printf_printf");
      b.initSvg();
      for (var c = b.getInput("STACK").connection, e = 1; e <= this.printAddCount_; e++) {
          var d = Blockly.Block.obtain(a, "library_stdio_printf_add");
          d.initSvg();
          c.connect(d.previousConnection);
          c = d.nextConnection
      }
      return b
  },
```

Figure 4.7: Illustration of code for block "library_stdio_printf" (part2).

```
  compose: function(a) {
      for (var b = this.printAddCount_; 0 < b; b--) this.removeInput("VAR" + b);
      this.printAddCount_ = 0;
      for (a = a.getInputTargetBlock("STACK"); a;) {
          switch (a.type) {
              case "library_stdio_printf_add":
                  this.printAddCount_++;
                  b = this.appendValueInput("VAR" + this.printAddCount_)
                    .setCheck(null).appendField("");
                  a.valueConnection_ && b.connection.connect(a.valueConnection_);
                  break;
              default:
                  throw "Unknown block type.";
          }
          a = a.nextConnection && a.nextConnection.targetBlock()
      }
  },
  saveConnections: function(a) {
      a =
          a.getInputTargetBlock("STACK");
      for (var b = 1; a;) {
          switch (a.type) {
              case "library_stdio_printf_add":
                  var c = this.getInput("VAR" + b);
                  a.valueConnection_ = c && c.connection.targetConnection;
                  a.statementConnection_ = b++;
                  break;
              default:
                  throw "Unknown block type.";
          }
          a = a.nextConnection && a.nextConnection.targetBlock()
      }
  },
  onchange: Blockly.Blocks.requireInFunction
};
```

Figure 4.8: Illustration of code for block "library_stdio_printf" (part3).

The "variables_declare_2" block in Blockly-C is a more intricate example of a mutable block, specifically designed to enhance user experience and compatibility with C syntax. Unlike the initial simple variable declaration blocks that allowed the declaration of only one variable per command, the

"variables_declare_2" block enables users to declare multiple variables in a single line, separated by commas, aligning with the syntactical allowances of C. While the original blocks were fully functional for basic C coding, the development of "variables_declare_2" was crucial to encompass the full spectrum of C syntax, particularly for the reverse conversion from plain text C to a block-based visual representation. This advanced block's complexity was necessary to ensure full adherence to C syntax, making it a significant addition to Blockly-C. The detailed code and UI implementation of this block are illustrated in figures from 4.9 until 4.15.

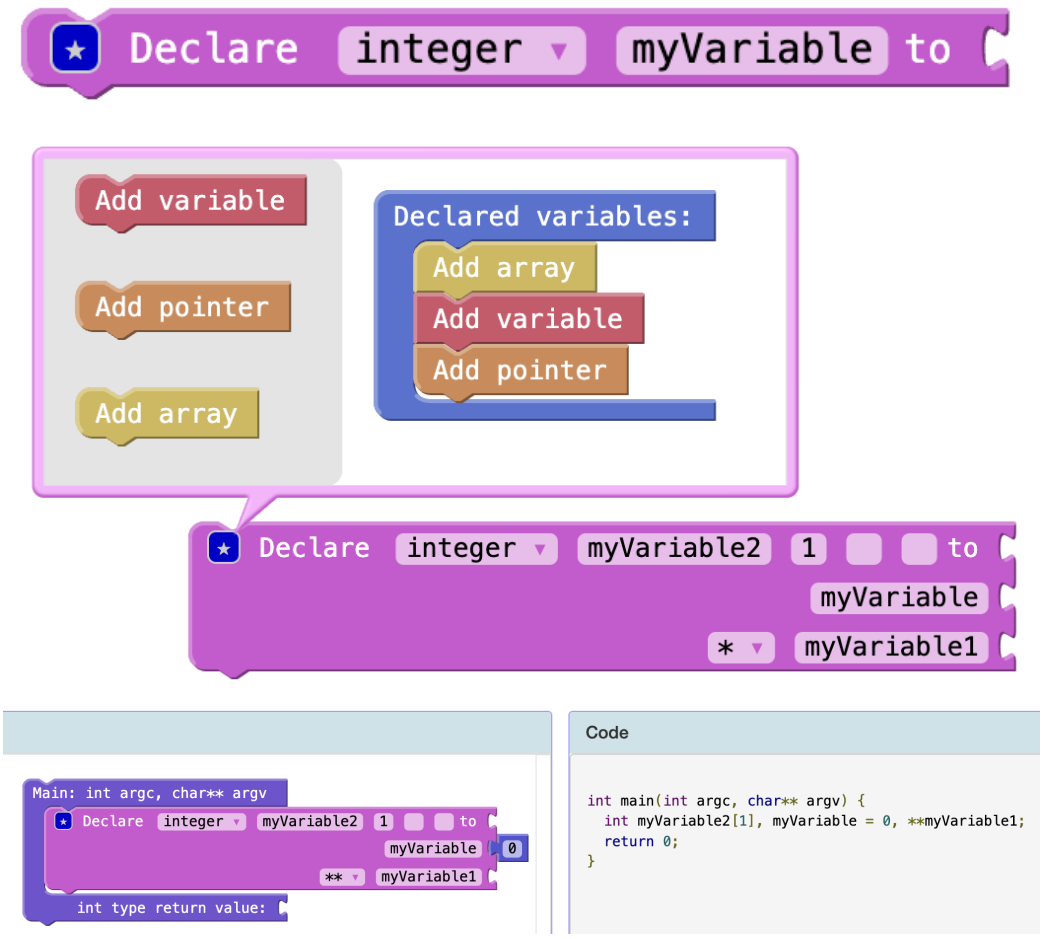
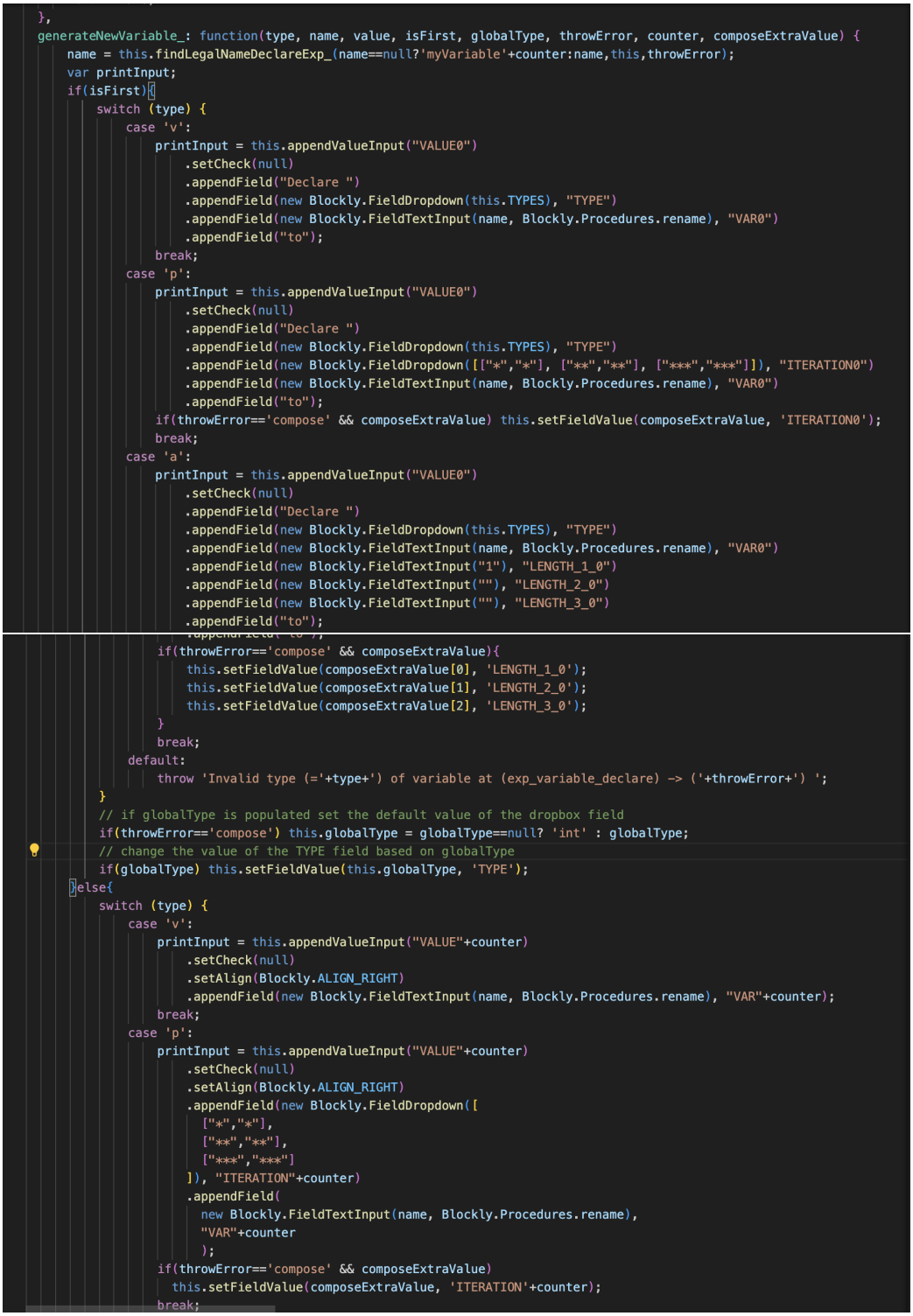


Figure 4.9: Illustration of "variables_declare_2" block (part 1).

```
Blockly.Blocks.variables_declare_2 = {
  init:function() {
      this.TYPES =
          [['integer', 'int'],
           ['unsigned int', 'unsigned int'],
           ['real number(float)', 'float'],
           ['real number(double)', 'double'],
           ['character', 'char']
          ];
      var name = Blockly.Procedures.findLegalName('myVariable', this);
      this.setColour(300);
      this.appendValueInput("VALUE0")
          .setCheck(null)
          .appendField("Declare ")
          .appendField(new Blockly.FieldDropdown(this.TYPES), "TYPE")
          .appendField(new Blockly.FieldTextInput(name, Blockly.Procedures.rename), "VAR0")
          .appendField("to");
      this.setPreviousStatement(!0,arrayPreviousStatementValues);
      this.setNextStatement(!0,arrayNextStatementValues);
      this.setMutator(new Blockly.Mutator(
        ['exp_variable_declare_mutation_add_block',
        'exp_variable_declare_mutation_add_pointer',
        'exp_variable_declare_mutation_add_array']));
      this.setTooltip('');
      this.contextMenuMsg_ = "Create 'get %1'";
      this.contextMenuType_ = 'variables_get';
      this.tag = ['variables', 'declare', 'define', 'declaration', 'definition'];

      // my variables
      this.myVariables = [[name,'v']];
      this.globalType = 'int';
      this.positionOfName = 0;
      this.positionOfType = 1; // 'v' || 'p' || 'a'
      this.mutationToDom();
  },
  mutationToDom:function() {
      if (!this.myVariables) return null;
      var container = document.createElement('mutation');

      if (this.myVariables) container.setAttribute('myvariables', JSON.stringify(this.myVariables));
      if (this.globalType) container.setAttribute('globaltype', this.globalType);
      if (this.positionOfName) container.setAttribute('positionofname', this.positionOfName);
      if (this.positionOfType) container.setAttribute('positionoftype', this.positionOfType);
      return container;
  },
  domToMutation: function(xmlElement) {
      var retrievedVariables = JSON.parse(xmlElement.getAttribute('myvariables'));
      if(retrievedVariables && retrievedVariables?.length && retrievedVariables?.length >= 1)
          this.myVariables = retrievedVariables;
      this.globalType = xmlElement.getAttribute('globaltype');
      this.positionOfName = 0;
      this.positionOfType = parseInt(xmlElement.getAttribute('positionoftype'));
      this.rebuildShape_(); // rebuild completely from 0 as it was before
  },
  findLegalNameDeclareExp_: function(name, block, throwError){
      // check the rest of the blocks
      name = Blockly.Procedures.findLegalName(name, block);
      // check the own block
      if(throwError=='compose'){
          for(var j=0;j<this.myVariables.length;j++){
              if(Blockly.Names.equals(this.myVariables[j][0], name)){
                  var r = name.match(/^(.*?)(\d+)$/);
                  if (!r) {
                      name += '2';
                  } else {
                      name = r[1] + (parseInt(r[2], 10) + 1);
                  }
                  return this.findLegalNameDeclareExp_(name, block, throwError);
              }
          }
      }
      return name;
```


Figure 4.10: Illustration of "variables_declare_2" block (part 2).



```
},
generateNewVariable_: function(type, name, value, isFirst, globalType, throwError, counter, composeExtraValue) {
    name = this.findLegalNameDeclareExp_(name==null?'myVariable'+counter:name,this,throwError);
    var printInput;
    if(isFirst){
        switch (type) {
            case 'v':
                printInput = this.appendValueInput("VALUE0")
                    .setCheck(null)
                    .appendField("Declare ")
                    .appendField(new Blockly.FieldDropdown(this.TYPES), "TYPE")
                    .appendField(new Blockly.FieldTextInput(name, Blockly.Procedures.rename), "VAR0")
                    .appendField("to");
                break;
            case 'p':
                printInput = this.appendValueInput("VALUE0")
                    .setCheck(null)
                    .appendField("Declare ")
                    .appendField(new Blockly.FieldDropdown(this.TYPES), "TYPE")
                    .appendField(new Blockly.FieldDropdown([["*","*"], ["**","**"], ["***","***"]]), "ITERATION0")
                    .appendField(new Blockly.FieldTextInput(name, Blockly.Procedures.rename), "VAR0")
                    .appendField("to");
                if(throwError=='compose' && composeExtraValue) this.setFieldValue(composeExtraValue, 'ITERATION0');
                break;
            case 'a':
                printInput = this.appendValueInput("VALUE0")
                    .setCheck(null)
                    .appendField("Declare ")
                    .appendField(new Blockly.FieldDropdown(this.TYPES), "TYPE")
                    .appendField(new Blockly.FieldTextInput(name, Blockly.Procedures.rename), "VAR0")
                    .appendField(new Blockly.FieldTextInput("1"), "LENGTH_1_0")
                    .appendField(new Blockly.FieldTextInput(""), "LENGTH_2_0")
                    .appendField(new Blockly.FieldTextInput(""), "LENGTH_3_0")
                    .appendField("to");
                if(throwError=='compose' && composeExtraValue){
                    this.setFieldValue(composeExtraValue[0], 'LENGTH_1_0');
                    this.setFieldValue(composeExtraValue[1], 'LENGTH_2_0');
                    this.setFieldValue(composeExtraValue[2], 'LENGTH_3_0');
                }
                break;
            default:
                throw 'Invalid type (='+type+') of variable at (exp_variable_declare) -> ('+throwError+') ';
        }
        // if globalType is populated set the default value of the dropbox field
        if(throwError=='compose') this.globalType = globalType==null? 'int' : globalType;
        // change the value of the TYPE field based on globalType
        if(globalType) this.setFieldValue(this.globalType, 'TYPE');
    }else{
        switch (type) {
            case 'v':
                printInput = this.appendValueInput("VALUE"+counter)
                    .setCheck(null)
                    .setAlign(Blockly.ALIGN_RIGHT)
                    .appendField(new Blockly.FieldTextInput(name, Blockly.Procedures.rename), "VAR"+counter);
                break;
            case 'p':
                printInput = this.appendValueInput("VALUE"+counter)
                    .setCheck(null)
                    .setAlign(Blockly.ALIGN_RIGHT)
                    .appendField(new Blockly.FieldDropdown([
                      ["*","*"],
                      ["**","**"],
                      ["***","***"]
                    ]), "ITERATION"+counter)
                    .appendField(
                      new Blockly.FieldTextInput(name, Blockly.Procedures.rename),
                      "VAR"+counter
                      );
                if(throwError=='compose' && composeExtraValue)
                  this.setFieldValue(composeExtraValue, 'ITERATION'+counter);
                break;
```

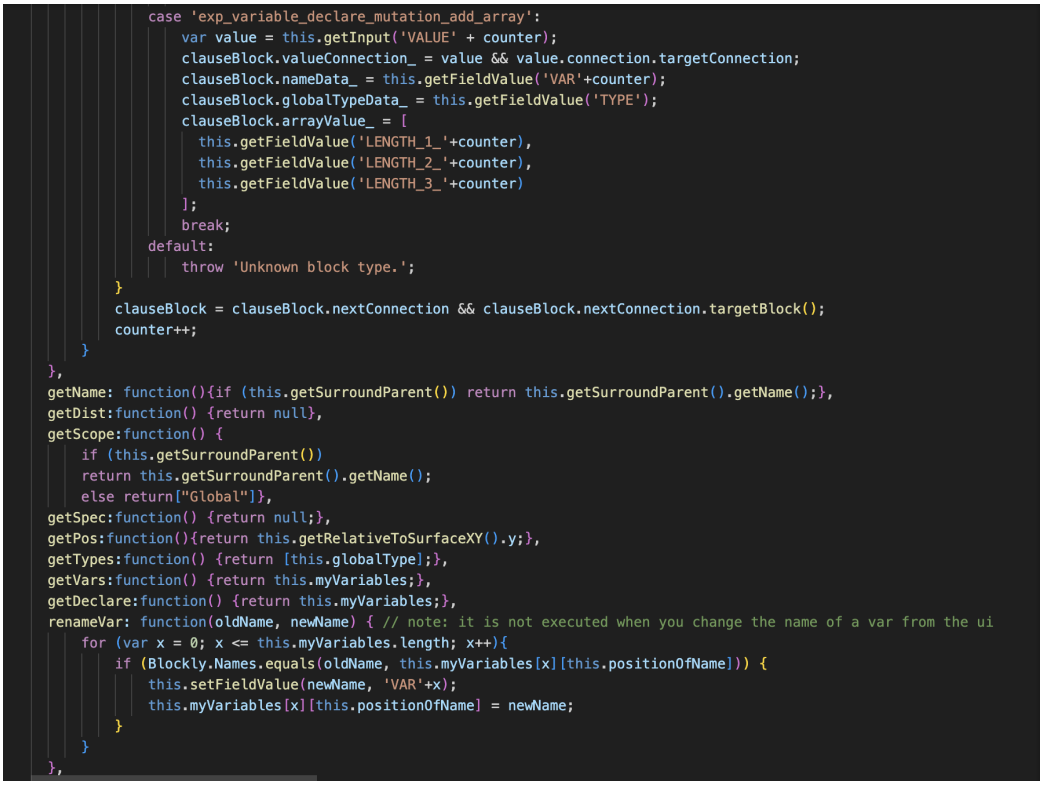
Figure 4.11: Illustration of "variables_declare_2" block (part 3).

```
                case 'a':
                    printInput = this.appendValueInput("VALUE"+counter)
                        .setCheck(null)
                        .setAlign(Blockly.ALIGN_RIGHT)
                        .appendField(new Blockly.FieldTextInput(name, Blockly.Procedures.rename), "VAR"+counter)
                        .appendField(new Blockly.FieldTextInput("1"), "LENGTH_1_"+counter)
                        .appendField(new Blockly.FieldTextInput(""), "LENGTH_2_"+counter)
                        .appendField(new Blockly.FieldTextInput(""), "LENGTH_3_"+counter);
                    if(throwError=='compose' && composeExtraValue){
                        this.setFieldValue(composeExtraValue[0], 'LENGTH_1_'+counter);
                        this.setFieldValue(composeExtraValue[1], 'LENGTH_2_'+counter);
                        this.setFieldValue(composeExtraValue[2], 'LENGTH_3_'+counter);
                    }
                    break;
                default:
                    throw 'Invalid type (='+type+') of variable at (exp_variable_declare) -> ('+throwError+') ';
            }
        }
        if(value) printInput.connection.connect(value);

        if(throwError=='compose') this.myVariables.push([name,type]);
    },
    rebuildShape_: function() {
        for (var x = this.myVariables.length; x >= 0; x--){
            if(this.getInput('VALUE' + x)) {
                this.removeInput('VALUE' + x);
            }
        }
        for (var x = 0; x < this.myVariables.length; x++){
            const curVar = this.myVariables[x];
            const name = curVar[this.positionOfName];
            const type = curVar[this.positionOfType];
            const valueConnection = this.getInput('VALUE' + x) ? this.getInput('VALUE' + x)
                .connection.targetConnection : null;
            if(x==0){
                const curGlobalType = this.getInput('TYPE') || this.globalType || null;
                this.generateNewVariable_(type, name, valueConnection, true, curGlobalType,
                  'domToMutation->rebuildShape_', x, null);
            }else{
                this.generateNewVariable_(type, name, valueConnection, false, null,
                  'domToMutation->rebuildShape_', x, null);
            }
        }
    },
    decompose:function(workspace) {
        var containerBlock = Blockly.Block.obtain(workspace, 'exp_variable_declare_mutation_block');
        containerBlock.initSvg();
        var connection = containerBlock.getInput('STACK').connection;
        for (var x = 0; x < this.myVariables.length; x++) {
            const curVar = this.myVariables[x];
            var expAddBlock = null;
            switch (curVar[this.positionOfType]) {
                case 'v':
                    expAddBlock = Blockly.Block.obtain(workspace, 'exp_variable_declare_mutation_add_block');
                    break;
                case 'p':
                    expAddBlock = Blockly.Block.obtain(workspace, 'exp_variable_declare_mutation_add_pointer');
                    break;
                case 'a':
                    expAddBlock = Blockly.Block.obtain(workspace, 'exp_variable_declare_mutation_add_array');
                    break;
                default:
                    throw 'Invalid type (='+curVar[2]+') of variable at (exp_variable_declare) -> (decompose) ';
            }
            expAddBlock.initSvg();
            connection.connect(expAddBlock.previousConnection);
            connection = expAddBlock.nextConnection;
        }
        return containerBlock;
    },
```

Figure 4.12: Illustration of "variables_declare_2" block (part 4).

```
compose: function(containerBlock) {
    // re-create completely the block based on the mutator block
    // all the information are stored inside "clauseBlock"
    // it also stores all the child blocks (the "clauseBlock")
    // we have to re-build the whole block and after that to connect all the child blocks again

    // delete everything
    for (var x = this.myVariables.length; x >= 0; x--){
        if(this.getInput('VALUE' + x)) {
            this.removeInput('VALUE' + x);
        }
    }

    // initialize all the variables
    this.myVariables = [];
    this.globalType = null;
    this.positionOfName = 0;
    this.positionOfType = 1;

    var isFirstBlock = true;
    var clauseBlock = containerBlock.getInputTargetBlock('STACK');
    while (clauseBlock) {
        const curName = clauseBlock.nameData_ || null;
        var curVariableType = null;
        var extraValue = null;
        if(clauseBlock.type=='exp_variable_declare_mutation_add_block') curVariableType = 'v';
        else if(clauseBlock.type=='exp_variable_declare_mutation_add_pointer'){
            extraValue = clauseBlock.pointerValue_;
            curVariableType = 'p';
        }else if(clauseBlock.type=='exp_variable_declare_mutation_add_array'){
            extraValue = clauseBlock.arrayValue_;
            curVariableType = 'a';
        }

        if(isFirstBlock){
            // take as value the current or the next one if exists (in case that we change the 1st block)
            const curGlobalType = clauseBlock.globalTypeData_ ||
            (clauseBlock.nextConnection && clauseBlock.nextConnection.targetBlock()
            ? clauseBlock.nextConnection.targetBlock().globalTypeData_  : null);
            this.generateNewVariable_(curVariableType, curName, clauseBlock.valueConnection_,
              true, curGlobalType, 'compose',this.myVariables.length, extraValue);
            isFirstBlock = false;
        }else {
            this.generateNewVariable_(curVariableType, curName, clauseBlock.valueConnection_,
              false, null, 'compose',this.myVariables.length, extraValue);
        }
        clauseBlock = clauseBlock.nextConnection && clauseBlock.nextConnection.targetBlock();
    }
},
saveConnections: function(containerBlock) {
    var clauseBlock = containerBlock.getInputTargetBlock('STACK');
    var counter = 0;
    while (clauseBlock) {
        switch (clauseBlock.type) {
            case 'exp_variable_declare_mutation_add_block':
                var value = this.getInput('VALUE' + counter);
                clauseBlock.valueConnection_ = value && value.connection.targetConnection;
                clauseBlock.nameData_ = this.getFieldValue('VAR'+counter);
                clauseBlock.globalTypeData_ = this.getFieldValue('TYPE');
                break;
            case 'exp_variable_declare_mutation_add_pointer':
                var value = this.getInput('VALUE' + counter);
                clauseBlock.valueConnection_ = value && value.connection.targetConnection;
                clauseBlock.nameData_ = this.getFieldValue('VAR'+counter);
                clauseBlock.globalTypeData_ = this.getFieldValue('TYPE');
                clauseBlock.pointerValue_ = this.getFieldValue('ITERATION'+counter);
                break;
```

Figure 4.13: Illustration of "variables_declare_2" block (part 5).



```
            case 'exp_variable_declare_mutation_add_array':
                var value = this.getInput('VALUE' + counter);
                clauseBlock.valueConnection_ = value && value.connection.targetConnection;
                clauseBlock.nameData_ = this.getFieldValue('VAR'+counter);
                clauseBlock.globalTypeData_ = this.getFieldValue('TYPE');
                clauseBlock.arrayValue_ = [
                  this.getFieldValue('LENGTH_1_'+counter),
                  this.getFieldValue('LENGTH_2_'+counter),
                  this.getFieldValue('LENGTH_3_'+counter)
                ];
                break;
            default:
                throw 'Unknown block type.';
        }
        clauseBlock = clauseBlock.nextConnection && clauseBlock.nextConnection.targetBlock();
        counter++;
    }
},
getName: function(){if (this.getSurroundParent()) return this.getSurroundParent().getName();},
getDist:function() {return null},
getScope:function() {
    if (this.getSurroundParent())
    return this.getSurroundParent().getName();
    else return["Global"]},
getSpec:function() {return null;},
getPos:function(){return this.getRelativeToSurfaceXY().y;},
getTypes:function() {return [this.globalType];},
getVars:function() {return this.myVariables;},
getDeclare:function() {return this.myVariables;},
renameVar: function(oldName, newName) { // note: it is not executed when you change the name of a var from the ui
    for (var x = 0; x <= this.myVariables.length; x++){
        if (Blockly.Names.equals(oldName, this.myVariables[x][this.positionOfName])) {
            this.setFieldValue(newName, 'VAR'+x);
            this.myVariables[x][this.positionOfName] = newName;
        }
    }
},
```

Figure 4.14: Illustration of "variables_declare_2" block (part 6).

```
onchange:function() {
    // show warnings
    if(this.getSurroundParent()==null){
        this.nextConnection.setCheck(['external_declaration']);
        this.previousConnection.setCheck(['external_declaration']);
    }else{
        this.nextConnection.setCheck(arrayNextStatementValues);
        this.previousConnection.setCheck(arrayPreviousStatementValues);
    }
    //keep updated the "this.myVariables"
    for(var x = 0; x < this.myVariables.length; x++){
        let currentName = this.getFieldValue('VAR'+x);
        if(currentName && currentName!=this.myVariables[x][0]){
            this.myVariables[x][0] = currentName;
        }
    }

    // keep updated the type of the block
    var a=this.getFieldValue("TYPE");
    0==a&&(a="int");
    if(this.globalType!=a){
        this.globalType = a;
    }

    // change set checks
    for (var x = 0; x < this.myVariables.length; x++){
        var type = this.myVariables[x][this.positionOfType];
        switch (type) {
            case 'v':
                Blockly.Blocks.setCheckVariable(this,a,"VALUE"+x);
                break;
            case 'p':
                "*"==this.getFieldValue("ITERATION"+x)
                ? Blockly.Blocks.setCheckPointer(this,a,"VALUE"+x) :
                "**"==this.getFieldValue("ITERATION"+x)&&
                Blockly.Blocks.setCheckPointer(this,"db"+a,"VALUE"+x)
                break;
            case 'a':
                break;
            default:
                throw 'Invalid type (='+type+') of variable at (exp_variable_declare) -> (onChnage) ';
        }
    }
}
};
```

Figure 4.15: Illustration of "variables_declare_2" block (part 7).

Once the block is defined, the next step is to map the block's structure to the desired code syntax for the targeted programming language. This translation is facilitated by the generator stub. It processes the information encapsulated within the block and generates it to the corresponding code structure. For every block that's been crafted in Blockly, there exists an equally essential counterpart: the code generator. The code generator is function in javascript that contains all the logic in order to translate the current state of a block into a plain text string. The generator's tasks don't end at the immediate block it translates. It also needs to account all the encapsulated or value connection blocks that are connected to the current

one. When processing the main block, the generator identifies these associated blocks. For each of them, it calls the corresponding code conversion function, ensuring that the entire structure of interconnected blocks is effectively translated. By taking into consideration every connected block and its specific function, the generator ensures a comprehensive and precise code output. Consequently, each of the 88 blocks that were developed required its own dedicated code generator function. Each of those functions were designed to product valid C code syntax. This detailed procedure was essential not only to guarantee the correct translation of the block's logic into code but also to prevent any potential syntax or grammatical errors in the output. To construct such a function, one can use the notation `Blockly.c.block_name = function(block){...}`. Each of these functions has as input the block object that has to convert and at the end it returns a string.

Figure 4.16 showcases a C code generator for a Blockly block, specifically for the "variables_declare_2" block. In this example, the generator function takes an instance of the block (represented by the variable "a") as its input. The function then performs a series of string manipulations to ensure that the block is accurately translated into the corresponding C syntax. After processing, it generates the C code as a string and appends a newline character `"\n"` at the end, thereby producing a syntactically correct and readable output in C language format.

```
Blockly.c.variables_declare_2=function(a){
  var n,type = a.getFieldValue('TYPE'),code=type,curVar;
  for (n = 0; n < a.myVariables.length; n++){
      curVar = a.myVariables[n];
      code += (n==0?' ':', ');
      var varName = Blockly.c.variableDB_.getName(curVar[0], Blockly.Variables.NAME_TYPE);
      if (Blockly.Blocks.checkLegalName(Blockly.Msg.VARIABLES_ILLEGALNAME, varName) == -1){
          this.initVar();
      }
      switch(curVar[1]){
          case 'v':
              code += curVar[0]
              break;
          case 'p':
              code += a.getFieldValue("ITERATION"+n) + curVar[0]
              break;
          case 'a':
              code += curVar[0]
              var len1 = a.getFieldValue("LENGTH_1_"+n),
                  len2 = a.getFieldValue("LENGTH_2_"+n),
                  len3 = a.getFieldValue("LENGTH_3_"+n),
                  len1 = 1 * len1,
                  len2 = 1 * len2,
                  len3 = 1 * len3;
              if (0 != len1 && 0 == len2 && 0 == len3) code += '['+len1+']'
              if (0 != len1 && 0 != len2 && 0 == len3) code += '['+len1+']' + '['+len2+']'
              if (0 != len1 && 0 != len2 && 0 != len3) code += '['+len1+']' + '['+len2+']' + '['+len3+']'
              break;
          default:
              break;
      }

      var argument = Blockly.c.valueToCode(a, 'VALUE'+n, Blockly.c.ORDER_ASSIGNMENT) || 'BLOCKLY_NO_VALUE';
      code += ((argument==='BLOCKLY_NO_VALUE')? '' : (' = '+argument));
  }
  return code+';\n';
};
```

Figure 4.16: Illustration of C generator for "variables_declare_2" block.

In addition to developing functions for each individual block, it was obligatory to create the generator for C as well in order for all this to work seamlessly. The procedure employed mirrored that of previously established generators. The C generator was initiated with `Blockly.c = new Blockly.Generator('c')`. Following this initialization, essential functions like `Blockly.c.init`, `Blockly.c.finish`, `Blockly.c.finishFull`, `Blockly.c.scrubNakedValue`, `Blockly.c.quote_`, and `Blockly.c.scrub_` were implemented to ensure that the generator behaves correctly for C. These functions handle everything from variable initializations, comment processing, to the overall structure and sequencing of the generated code. Moreover, to prevent naming conflicts and to guarantee accuracy in the code generation process, comprehensive lists of reserved words were provided, along with de-

fined operator precedence specific to C. The generator was also designed to manage special scenarios, such as naked values. These are top-level blocks that aren't connected to any other blocks, and they are appropriately appended with a semicolon to maintain code legality. Additionally, the aspect of string encoding was considered to ensure proper character escapes suitable for C. The generator even captures and assimilates comments from the *Blockly* interface into the resultant C code, ensuring a holistic and efficient transformation from *Blockly* blocks to C language syntax.

In the development of the generator, particular attention was also given to the inclusion of libraries in C. Recognizing the importance of streamlining the user experience, the generator was engineered to automatically incorporate the necessary "Include" libraries on top of the generated code. Whenever users employ one of the functions associated with a particular library within the Blockly interface, the corresponding library is automatically integrated into the generated C code. This alleviates the users from the manual burden of remembering and adding requisite libraries, ensuring that the transition from Blockly blocks to C code remains both efficient and user-friendly.

Lastly, to visualize the newly created blocks, they must be added to the Blockly toolbox XML, ensuring it becomes accessible from the Blockly workspace.

Following the examination on block creation, it is imperative to delve into the additional modifications undertaken within Blockly's core framework. This entailed alterations across several foundational files which form the basis of the system's operation. The files that subjected to modifications are the xml.js, blocks.js, field_dropdown.js, variables.js, procedures.js, connection.js, and block.js. Each of these files had specific changes tailored to address particular needs and to enhance the overall functionality and compatibility of the system with the C language.

The modifications regarding to the "xml.js" file will be examined in the subsequent section, where the conversion of C code into graphical blocks will be explored.

In the "blocks.js" file were added some util functions regarding the new blocks that were created. Within this file, developed functions that manage array indexing, like getIndexArray, which processes an array list to retrieve specific index values. The arrayTestFunction is a validation function that ensures array lengths are sequential and raises warnings otherwise. In terms of searching capabilities, functions such as search, searchTag, checkResult,

and showResult provide robust tools to search for blocks based on tags and render these results within the main workspace. Additionally, name validity is ascertained using checkLegalName, ensuring adherence to the C naming format. The file also brings in enhanced functionalities like setCheckVariable and setCheckPointer, which set the types of blocks or values permissible for certain input fields based on variable or pointer types. Lastly, the checkUnselect function serves as a filter to identify and modify content that matches predefined "unselected" labels, ensuring a seamless user experience.

The field_dropdown.js file has been meticulously updated to align better with C language conventions, enhancing Blockly's dropdown functionalities. The key modifications are evident in the listCreate function, which has been carefully redefined to curate a dropdown list of variables for set and get blocks pertinent to variables, pointers, arrays, and defines. Upon invoking this function, the list of all declared variables is fetched from the allVariables function. It ensures a precise and context-aware list by applying two critical filtering mechanisms. Firstly, it scrutinizes the temporal sequence of variable declarations relative to the block's position, ensuring only variables declared prior to the current block are included. This check prevents potential semantic errors where a variable might be invoked before its declaration. Secondly, the list undergoes a type-based filter. Variables are carefully sorted based on their type - define, regular variable, pointer, or array, represented by the numbers 0 through 3, respectively. For instance, when a user interacts with a 'get' block for 'define', the dropdown list will judiciously exclude names of pointers or regular variables, preserving the integrity of the C language semantics. The other functions, getTypefromVars and getParentType, though pre-existing, have also been modified to accommodate these C language-centric enhancements. Together, these refinements ensure that Blockly's interface presents only valid and contextually accurate dropdown menu options, improving user experience and code reliability.

Important changes made inside the variables.js file, to enhance the Blockly environment for C programming. Inside this file is the function that extracts and stores all the declared variables from the main Workspace. This is the function "allVariables". Delving into the function's logic reveals the utilization of a map variable with name "variableHash". Each variable is indexed in this map using a distinct string identifier as its key. The unique key is formulated by joining the variable's name with the scope derived from its associated parent block. In this context, the term "scope" refers to a fusion of the block's type and its unique identifier. Within Blockly's ecosystem, each

block possesses a type, defining its role or functionality. Additionally, upon its creation and initialization within the workspace by the user, each block carries a unique identifier (ID), ensuring it stands apart from other blocks, even those sharing the same type.

```
        }
    }else{
        variableHash[varName.toLowerCase()+"."+varScope.toLowerCase()] =
            [varType,varDist,varName,varScope,varPos,varSpec];
    }
}
```

Figure 4.17: Unique Storage Mechanism for Variables in the variableHash map variable.

In Figure 4.17, the code demonstrates the methodology for storing each variable within the 'variableHash'. Each variable's key is a concatenation of the variable's name and its scope, separated by a dot. This ensures that two variables with the same name but in different scopes (i.e., different blocks or contexts) can coexist without conflict. Each variable is cataloged with its distinct attributes: variableType, variableDist, variableName, variableScope, variablePosition, and VariableSpec. This meticulous documentation ensures complete knowledge about each variable.

Significantly, the detailed modifications made to the allVariables function within the variables.js are intricately linked with the previously discussed listCreate function in field_dropdown.js. This association emphasizes the integrated approach adopted to enhance the Blockly environment, guaranteeing that users have a seamless and error-free experience, from the declaration of a variable to its selection from a dropdown.

Complementing the transformations in variables.js is the .generateUniqueName functionality. This feature is pivotal when new variables or functions are declared, ensuring that each name is unique. This function is called inside blocks that declare new variables and new functions ensuring the assignment of unique names. It's an instrumental n safeguarding against syntax errors, especially for those new to programming. Additionally, modifications were made to another essential function, the flyoutCategory.

In the procedures.js file, the isLegalName function underwent modifications to ensure unique naming across different Blockly blocks. Previously focused on checking procedure names, its scope has broadened to include

blocks like 'structure_define', custom functions and all the type of variable declarations. The function scans all blocks in the workspace and determines the type of name-checking based on the block's type. It then checks either the procedure definitions, structure names, or general declarations accordingly. If a match is found, the function returns false, indicating the name is not unique; if no matches are found, it returns true, signifying the name's uniqueness within the workspace.

Within the block.js file, modifications were made to the onMouseMove_ function, specifically tailored to extend the dynamic adaptability we discussed in relation to the onChange function. For the very same need to facilitate blocks to be placed both inside and outside main functions while adhering to C's syntax, the onMouseMove_ function was enhanced. As a user drags a block across the workspace, the onMouseMove_ function is executed. This gives the flexibility to dynamically adjust the rules for its potential connection points, ensuring the connections remain syntactically correct for C programming. By doing so, the block's behavior remains consistent with its intended purpose, whether it's placed as an external declaration or nested within a function, offering an intuitive and error-free experience for the user. To address this dynamic connectivity requirement, an added conditional check was incorporated to discern if the block in transit belongs to the categories of 'variable declare', 'custom function', 'defined variable declaration', or 'structure declaration'. These particular blocks are earmarked for their ability to connect both internally and externally. When any of these block types are identified, their next and previous connections are adeptly modified to match the intended placement. With that final adjustment, the discussion on the modifications made to Blockly's core is concluded.

In wrapping up this chapter, it's worth noting that Blockly offers an efficient code compression system. This system allows the compression of all the functionalities of blockly into just a few files. In our case, we only need 3 files that were generated in order to run our refined blockly-c platform. The core mechanics of Blockly are placed within the "blockly_compressed.js" file. Similarly, the custom C blocks that were developed are neatly packed into "blocks_compressed.js", and the C code generator finds its home in "c_compressed.js". This streamlined structure ensures both efficiency and manageability for the platform."

4.3 C-to-Block Representation

This chapter illuminates the procedures and challenges associated with converting textual C code into Blockly's graphical blocks. The primary objective is to empower users with the capability to effortlessly import code – whether self-written or sourced from the internet – into the web application, which would then autonomously transmute it into blocks. This feature is especially advantageous for novices, facilitating a smoother transition into the realm of coding by presenting intricate lines of code in a more comprehensible, visual format. Concurrently, seasoned programmers stand to benefit from this functionality as well. The graphical representation of C commands streamlines the process of deciphering unfamiliar code, thereby enhancing productivity. The visually intuitive interface not only accelerates understanding but also injects an element of engagement into code analysis.

By default, Blockly supported the transformation of the main workspace into an XML file and vice versa. This feature was primarily designed to provide a mechanism for preserving ongoing work, allowing for subsequent retrieval and continuation. However, this existing functionality was tailored specifically for Blockly's native blocks. As highlighted in the preceding chapter, it was necessary to delete the pre-existing blocks and develop new ones that align with the C programming language. Consequently, there was a need to reconstruct the block-to-XML conversion process from scratch. For each of the newly crafted mutable blocks, the functions `mutationToDom` and `domToMutation` were implemented as it is already discussed. These ensured the capability to individual mutable blocks to XML conversion and retrieval, preserving their state for potential reinstatement. For the regular blocks there is no need to implement such function during declaration, because the Blockly's core handles their conversion. Beyond these block-specific XML conversions, modifications were also made to the xml.js file, an integral component of Blockly's core.

The xml.js file focus on the functionality required for serializing and deserializing Blockly blocks using XML and it's an integral part of the Blockly framework. This file contains the function `workspaceToDom`, which starts the serialization process, converting all the blocks in a Blockly workspace into an XML DOM. The logic accommodates for variables and constructs an XML representation based on block type, fields, values, mutations, etc. The function `blockToDom` provides a similar function but focused on an individual regular blocks. If a block is a mutable one then instead it uses the

`mutationToDom` function. The `blockToDom` function is used by the workspace-level function (`workspaceToDom`) to recursively handle every block and its children. It creates an XML node for the block, then iterates through all inputs, fields, and values to generate the full XML representation.

On the flip side, deserialization is handled by functions like `domToWorkspace` and `domToBlock`. The `domToWorkspace` function initializes or fills a workspace with blocks represented by the given XML DOM. It begins by clearing the workspace, and then translates each XML block into actual Blockly blocks. The `domToBlock` is more granular, converting an XML representation of a block into a Blockly block within a workspace. Like its serialization counterpart, this function dives deep into the block's attributes and child nodes to rebuild its Blockly representation.

Among all those various functions, the `domToBlock` function needed modification to accommodate unique conversions from XML to blocks, specifically when dealing with pointer or array declaration blocks. Both these block types have distinct fields that set them apart from standard blocks, necessitating special input handling. For instance, the pointer declaration block includes a field to indicate the number of asterisks in the pointer, such as * or **. Similarly, the array variable declaration block has a dedicated field to capture the array's dimensions, like [2][4][3].

After implementing the features mentioned earlier, Blockly now supports conversions from blocks to XML and reverse. The desired goal was to support the transformation of plain C language code into blocks. It was envisioned a solution to create a C compiler, which would be used to convert syntactically correct code C into XML, which could then be transformed into blocks by using Blockly's `domToWorkspace` function.

However, the challenge lay in ensuring this solution was web-based. This necessitated the entire process to be crafted in JavaScript. To achieve this, it was used the JISON library.

As it is already discussed, JISON is a JavaScript parser generator inspired by the Bison parser generator, which is commonly used in C and C++ development. JISON takes a grammar specification in Bison's syntax and produces a JavaScript parser for that grammar. With JISON, developers can define grammars for new languages or data formats, and then generate JavaScript code that can parse strings of those languages/formats into a structured format or even execute code based on them. Consequently, a JISON grammar was designed specifically for the C language. This grammar

accepts a string containing plain C code and transforms it into a specific XML format that Blockly's `domToWorkspace` function recognizes.

The process of creating a parser for a language involves two distinct phases. The initial phase, called lexical analysis, details how the input is segmented or tokenized into meaningful symbols or tokens. The programmer specifies patterns for tokens they're interested in, using regular expressions. As the input is read, these tools recognize sequences of characters that match these patterns and subsequently convert them into tokens. This streamlining ensures that the parser doesn't grapple with individual characters but rather deals with higher-level symbolic representations. For this conversion, the most famous tools to use is the Lex and Flex.

Following lexical analysis, the next step is grammar analysis or parsing. This phase revolves around identifying the structure of the token sequence and determining if it adheres to the rules of the language. Bison and Yacc are very common tools for this purpose. The developer defines a context-free grammar, which encapsulates the syntactical rules of the target language. Each rule defines how sequences of tokens can be grouped together to form higher-level constructs. For example, in a programming language, rules might dictate how sequences of tokens represent valid statements, expressions, or function declarations. When Bison or Yacc processes the grammar, they construct a table-driven parser. As this parser processes the stream of tokens, it consults these tables to decide which grammar rule to apply. If the input adheres to the grammar, the parser can construct a parse tree or an abstract syntax tree representing the input's structure. Moreover, developers can embed actions within the grammar rules in Bison and Yacc. This means that when a specific rule is recognized during parsing, associated actions, like generating intermediate code or updating a symbol table, can be executed. This technique was used to individually convert each line of C code into its corresponding XML format, representing the specific block to which that line of code should be translated.

In essence, through the joint use of Lex/Flex for token generation and Yacc/Bison for grammar-based parsing, developers can effectively design and implement interpreters or compilers for new languages. It's a systematic method where raw text is first reduced to tokens and then structured according to predefined syntactic rules, facilitating subsequent language processing or interpretation.

Jison operates on same principles. It allows developers to define a parser by using the same syntax as Flex/Bison except for some minor differences.

To generate a JavaScript parser using Jison, it needs to be created a .jison file that encapsulates both the lexical grammar (Flex) and the language grammar (Bison). Jison employs the format that is showing in figure 4.18.

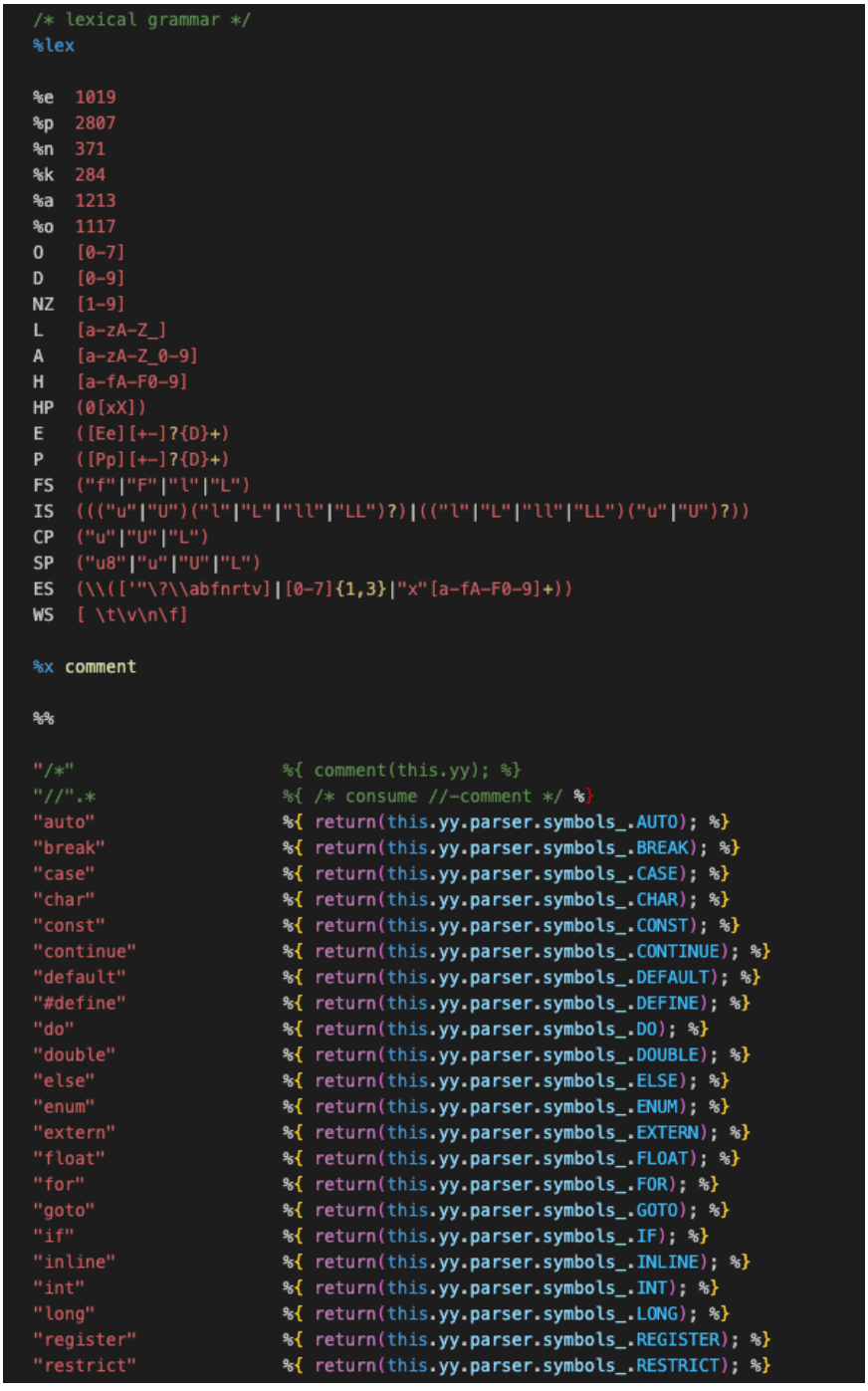
```
/* lexical grammar */
%lex

...

/lex

/* operator associations and precedence */

%token EXAMPLE_TOKEN_1 EXAMPLE_TOKEN_2 ...
%start total_code

%%
/* language grammar */

...
```

Figure 4.18: Structure and Flow of a Jison Parser Definition File.

The section of lexical grammar is placed at the top of the file, delineated by **%lex** at the beginning and **/lex** at the end. Following the lexical grammar, there is a section dedicated to declaring tokens and specifying operator precedence. This is where you'd define tokens that your grammar will recognize and use. It provides a bridge between the lexical and syntactical phases, ensuring both operate with a consistent set of symbols. Post token definitions, the file houses the language grammar segment, demarcated by %%. This is the position of the parser, defining how sequences of tokens form valid constructs in the target language. The rules are articulated in the form of context-free grammar productions.

Another distinction between Flex/Bison and Jison syntaxes lies in the lexical grammar of Jison. Specifically, exact string patterns should be enclosed in quotes, **"**, which isn't always the case with Flex. For instance, the Flex expression **float print(yytext)** would be reformulated in Jison as **"float" print(yytext);**. Furthermore, multi-line actions in Jison should be encapsulated within braces, as in **"float" %{ print(yytext); return 'FLOAT'; %}**.

The lexical grammar that was developed for the C language can be shown in the figures 4.19, 4.20 and 4.21.



```
/* lexical grammar */
%lex

%e  1019
%p  2807
%n  371
%k  284
%a  1213
%o  1117
O   [0-7]
D   [0-9]
NZ  [1-9]
L   [a-zA-Z_]
A   [a-zA-Z_0-9]
H   [a-fA-F0-9]
HP  (0[xX])
E   ([Ee][+-]?{D}+)
P   ([Pp][+-]?{D}+)
FS  ("f"|"F"|"l"|"L")
IS  ((("u"|"U")("l"|"L"|"ll"|"LL")?)|(("l"|"L"|"ll"|"LL")("u"|"U")?))
CP  ("u"|"U"|"L")
SP  ("u8"|"u"|"U"|"L")
ES  (\\(['"\?\\abfnrtv]|[0-7]{1,3}|"x"[a-fA-F0-9]+))
WS  [ \t\v\n\f]

%x comment

%%

"/*"                    %{ comment(this.yy); %}
"//".*                  %{ /* consume //-comment */ %}
"auto"                  %{ return(this.yy.parser.symbols_.AUTO); %}
"break"                 %{ return(this.yy.parser.symbols_.BREAK); %}
"case"                  %{ return(this.yy.parser.symbols_.CASE); %}
"char"                  %{ return(this.yy.parser.symbols_.CHAR); %}
"const"                 %{ return(this.yy.parser.symbols_.CONST); %}
"continue"              %{ return(this.yy.parser.symbols_.CONTINUE); %}
"default"               %{ return(this.yy.parser.symbols_.DEFAULT); %}
"#define"               %{ return(this.yy.parser.symbols_.DEFINE); %}
"do"                    %{ return(this.yy.parser.symbols_.DO); %}
"double"                %{ return(this.yy.parser.symbols_.DOUBLE); %}
"else"                  %{ return(this.yy.parser.symbols_.ELSE); %}
"enum"                  %{ return(this.yy.parser.symbols_.ENUM); %}
"extern"                %{ return(this.yy.parser.symbols_.EXTERN); %}
"float"                 %{ return(this.yy.parser.symbols_.FLOAT); %}
"for"                   %{ return(this.yy.parser.symbols_.FOR); %}
"goto"                  %{ return(this.yy.parser.symbols_.GOTO); %}
"if"                    %{ return(this.yy.parser.symbols_.IF); %}
"inline"                %{ return(this.yy.parser.symbols_.INLINE); %}
"int"                   %{ return(this.yy.parser.symbols_.INT); %}
"long"                  %{ return(this.yy.parser.symbols_.LONG); %}
"register"              %{ return(this.yy.parser.symbols_.REGISTER); %}
"restrict"              %{ return(this.yy.parser.symbols_.RESTRICT); %}
```

Figure 4.19: Jison's lexical grammar for C language (part 1).

Figure 4.20: Jison's lexical grammar for C language (part 2).

```
"||"                    %{ return this.yy.parser.symbols_.OR_OP; %}
"<="                    %{ return this.yy.parser.symbols_.LE_OP; %}
">="                    %{ return this.yy.parser.symbols_.GE_OP; %}
"=="                    %{ return this.yy.parser.symbols_.EQ_OP; %}
"!="                    %{ return this.yy.parser.symbols_.NE_OP; %}
";"                     %{ return ';'; %}
("{"|"<%")              %{ return '{'; %}
("}"|"%>")              %{ return '}'; %}
","                     %{ return ','; %}
":"                     %{ return ':'; %}
"="                     %{ return '='; %}
"("                     %{ return '('; %}
")"                     %{ return ')'; %}
("["|"<:")              %{ return '['; %}
("]"|":>")              %{ return ']'; %}
"."                     %{ return '.'; %}
"&"                     %{ return '&'; %}
"!"                     %{ return '!'; %}
"~"                     %{ return '~'; %}
"-"                     %{ return '-'; %}
"+"                     %{ return '+'; %}
"*"                     %{ return '*'; %}
"/"                     %{ return '/'; %}
"%"                     %{ return '%'; %}
"<"                     %{ return '<'; %}
">"                     %{ return '>'; %}
"^"                     %{ return '^'; %}
"|"                     %{ return '|'; %}
"?"                     %{ return '?'; %}
[ \r\t]                 %{ /* skip whitespace */ %}
\n                      %{ /* whitespace separates tokens */ %}
{WS}+                   %{ /* whitespace separates tokens */ %}
<<EOF>>                 %{ return 'EOF'; %}
.                       %{ /* discard bad characters */ %}

%%
function comment(yy){
    var c,c1;
    while(true){
        if((c1 = yy.lexer.input()) != '/' && c != undefined){
            yy.lexer.unput(c1)
            continue;
        }
        break;
    }
}

/lex
```

Figure 4.21: Jison's lexical grammar for C language (part 3).

This code provides the lexical grammar for a Jison parser tailored for the C language. The lexical grammar serves as the primary mechanism for tokenizing a C source code into meaningful components (tokens). It begins with the definition of patterns using regular expressions. For instance, patterns are defined for digits, hex digits, and alphabets. These patterns simplify subsequent token definitions by making them more concise. The main section, demarcated by %%, identifies specific tokens and associates them with corresponding C keywords, symbols, and constructs. For instance, the keyword "break" is associated with its respective symbol in the parser. Similarly, patterns for integer constants, floating-point constants, string literals, and various C operators are defined. Comments in the C code, which begin with /* and end with */, are handled by the comment function. This function essentially consumes and ignores the comment content, allowing the parser to disregard it in subsequent stages. In addition, the single line comments, which begin with // are ignored too. Whitespace, including spaces, tabs, and newlines, are skipped, ensuring that it doesn't interfere with meaningful token extraction. Any character not explicitly accounted for is discarded, ensuring robustness against unexpected input. Finally, the token 'EOF' represents the end of the file. The above code ensures that a given piece of C code is effectively broken down into its basic constructs, allowing for more detailed syntactic analysis in subsequent stages of parsing.

After the lexical grammar, there is the section dedicated to declaring tokens and specifying operator precedence. The code that was placed there can be shown in the figure 4.22

```
/* operator associations and precedence */

%token  IDENTIFIER I_CONSTANT F_CONSTANT STRING_LITERAL FUNC_NAME SIZEOF
%token  PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token  AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token  SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token  XOR_ASSIGN OR_ASSIGN
%token  TYPEDEF_NAME ENUMERATION_CONSTANT

%token  TYPEDEF EXTERN STATIC AUTO REGISTER INLINE
%token  CONST RESTRICT VOLATILE
%token  BOOL CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE VOID
%token  COMPLEX IMAGINARY
%token  STRUCT UNION ENUM ELLIPSIS

%token  CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%token  ALIGNAS ALIGNOF ATOMIC GENERIC NORETURN STATIC_ASSERT THREAD_LOCAL

%token DEFINE

%start total_code
```

Figure 4.22: Token Declarations and Operator Precedence in Jison for C Language Parsing.

This code defines tokens for the Jison grammar tailored for the C programming language. These tokens represent keywords, operators, data types, and other lexical units found in C. For instance, tokens like **IDENTIFIER** and **I_CONSTANT** correspond to variable names and integer constants respectively, while others like **IF**, **ELSE**, and **FOR** signify control flow constructs. Additionally, the grammar recognizes advanced C language features, such as storage class specifiers and type qualifiers. The **%start** directive indicates that the primary rule or starting point for the parser is **total_code**, suggesting that the overall structure of a C program will be captured in this rule. In essence, this segment of the Jison grammar prepares the groundwork for tokenizing and parsing C source code.

The next part of the .jison file is the lexical grammar. The grammar

analysis of Jison has exactly the same syntax with Bison. The only difference between them is that within the embedded actions of each grammar rule, developers must use JavaScript code instead of C code. This aligns with the objective of producing a final parser in the JavaScript language, which was the initial goal of choosing Jison. A small part of the grammar that was developed for the C language can be shown in the figure 4.23.

```
total_code
    : translation_unit EOF  {{ return($1); }}
    ;

translation_unit
    : external_declaration
    | translation_unit external_declaration {{
        if(!$1) $$ = $2
        else if(!$2) $$ = $1
        else {
            // both S1 and S2 are not empty.
            let pos = String($1).lastIndexOf('<next>')+6
            $$ = [String($1).slice(0, pos),String($2), String($1).slice(pos)].join('')
        }
        }}
    ;

external_declaration
    : function_definition
    | declaration {{
        if( checkIfItStartsWith($1,'<block type="structure_define">') ){
            $$ = $1;
        }else if( checkIfItStartsWith($1,'<block type="exp_variable_declare"') ){
            $1 = String($1).replace('<block type="exp_variable_declare"', '<block type="variables_declare_2"')
            let a,b;
            [a,b] = splitStringIntoTwoPartsFindLast_AfterString($1, '</block>')
            $$ = a.concat('<next></next>').concat(b)
        }else {
            throw new Error('Exception, uknown type in external_declaration.declaration')
        }
        }}
    | define_all
    ;
```

Figure 4.23: Snippet from the C Transpiler Grammar Implementation.

The main ideas and structure of this grammar will be highlighted. While some essential parts will be discussed, not everything can be covered due to its extensive length, spanning over 1300 lines of code. This expansive nature is primarily attributed to the fact that it is designed to encompass the entire grammar of the C language, which is intrinsically vast and intricate. This grammar essentially sets out the syntactic boundaries that dictate how valid programs can be framed within this language subset. Each rule within the grammar delineates a non-terminal symbol, and this is succeeded by a

sequence of terminals (tokens) and/or non-terminal symbols. The vertical bar (|) is utilized to segregate different plausible expansions or derivations of a non-terminal.

The initiation of this grammar is rooted in the rule named **total_code**. The **total_code** represents the entire code of a program in C that can be inserted into the parser. The above code denotes that a valid program is composed by a **translation_unit**, subsequently followed by an end-of-file (EOF). The **translation_unit**, according to its definition, can either be an **external_declaration** or, in a recursive twist, of another **translation_unit** coupled with an **external_declaration**. This implies that a **translation_unit** can consist of one or multiple **external_declaration** units. Thus, a valid program is composed by one or a series connections of **external_declarations**.

An **external_declaration**, can take the form of a function declaration, a variable declaration, or a define declaration. For instance, the familiar "main" function of a C program is an example of a function declaration. Besides the "main", a "function declaration" can also be any other custom function that a developer can declare. According to C syntax, it's permissible for programmers to declare variables or define values outside the "main" function or any other custom function. This adherence to C syntax is why the **declaration** and **define_all** components are integral to the **external_declaration** unit.

As a graphical illustration of the developed grammar, the parsed tree of this simple code in C **"int main(){return 0;}"** can be shown in figure 4.24.

Figure 4.24: Parse tree for example code in C.

As it is shown in the above figure, there are many encapsulations in order for constant "0" to arrive in the **jump_statement** unit which represents the single line of code **"return 0;"**. This complexity arises because the "return" command, along with many others, can accept an expression as its argument. Among all C syntax units, the expression unit is the most encapsulated.

The expression unit in the C language is a versatile and multifaceted component, pivotal to the language's flexibility and power. At its core, an expression represents a value, which can be a simple constant, a variable, or a more complex combination of variables and operators that yield a result. In C, expressions can range from the straightforward, like a + b, to the complex, such as function calls with nested operations: x = sqrt(y * (a + b)). The parse tree of the code **"x = sqrt(y * (a + b))"** can be shown in the figure 4.25.

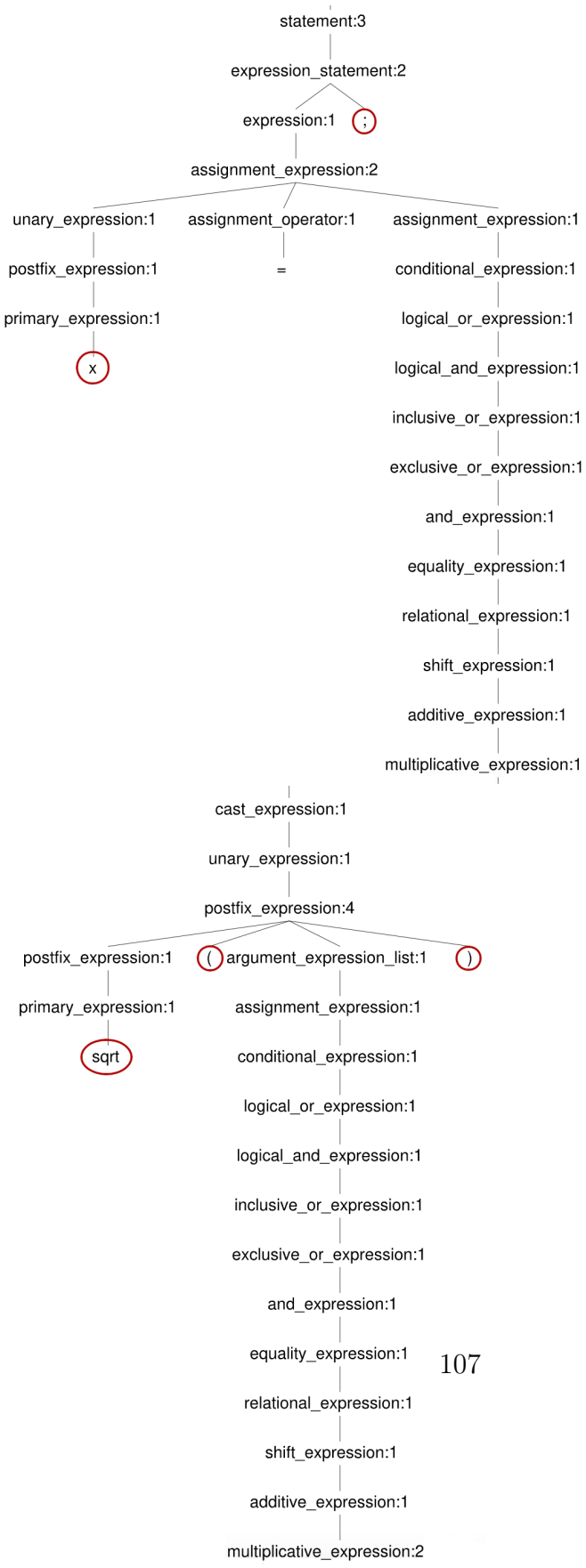

statement:3
expression_statement:2
expression:1
;
assignment_expression:2
unary_expression:1
assignment_operator:1
assignment_expression:1
postfix_expression:1
=
conditional_expression:1
primary_expression:1
logical_or_expression:1
x
logical_and_expression:1
inclusive_or_expression:1
exclusive_or_expression:1
and_expression:1
equality_expression:1
relational_expression:1
shift_expression:1
additive_expression:1
multiplicative_expression:1
cast_expression:1
unary_expression:1
postfix_expression:4
postfix_expression:1
(
argument_expression_list:1
)
primary_expression:1
assignment_expression:1
sqrt
conditional_expression:1
logical_or_expression:1
logical_and_expression:1
inclusive_or_expression:1
exclusive_or_expression:1
and_expression:1
equality_expression:1
relational_expression:1
shift_expression:1
additive_expression:1
multiplicative_expression:2

Figure 4.26: Parse tree for expression (part 2).

Several elements contribute to the depth of C's expression unit. First, there are the basic arithmetic operations – addition, subtraction, multiplication, and division – which can be layered and combined in various ways. On top of these, C offers a rich set of relational and logical operators, enabling comparisons (<, >, ==, !=) and logical decisions (&&, ||, !). Additionally, bitwise operations provide a means to directly manipulate individual bits within data, a feature that underscores C's low-level capabilities.

Another crucial aspect of C's expression syntax is the use of parentheses to determine precedence and grouping. While operators have inherent precedence, parentheses can override or clarify the order of operations, ensuring that expressions are evaluated correctly.

Function calls also fall within the realm of expressions. When a function is called, it can return a value, making the function call itself an expression. For instance, in y = foo(a, b), the function foo is expected to return a value that gets assigned to y.

Lastly, the C language provides a variety of unary operators and compound assignment operations, further enriching the possibilities within an expression. For example, the increment (++) and decrement (–) operators adjust a variable's value, while compound assignments like += and *= combine arithmetic with assignment in a single step.

In conclusion, the expression unit is a testament to the C language's depth and versatility. It encapsulates a wide array of operations and possibilities, allowing developers to concisely represent complex logic and computations.

The following analysis centers on the embedded actions inherent to each grammar rule. The primary function of the parser is to intake a string containing C code and, ultimately, produce and emit another string. This resultant string transforms the input code into an XML format recognizable by Blockly. However, before delving into the mechanics of this XML string conversion, it's crucial to first understand the specific XML format accepted by Blockly.

In Blockly, every block is denoted by the <block> tag. As per XML conventions, each tag must have a corresponding opening and closing. Furthermore, every <block> element mandatorily carries a type attribute, defining what kind of block it corresponds to. An illustrative example of such a block in XML is presented in Figure 4.27.

```
<block type="controls_if"></block>
```

Figure 4.27: XML Representation of a Blockly Block.

Most the blocks have fields that can be text, dropdown menus, checkboxes, etc. Each field is signified by a <field> element nested within the block. It has a name attribute which tells Blockly what the field represents. The actual value or text of the field lies between the opening and closing tags of the <field> element (figure 4.28).

```
<block type="math_number">
  <field name="NUM">123</field>
</block>
```

Figure 4.28: XML representation of a block with a field.

Blocks in Blockly can have inputs, which can either be value inputs (usually accepting a single block) or statement inputs (which can accept multiple stacked blocks). These inputs are denoted by <value> and <statement> elements, respectively. Both types of input elements will have a name attribute that identifies their purpose or function (figure 4.29).

```
<block type="controls_if">
  <value name="IF0">
    ...
  </value>
  <statement name="DO0">
    ...
  </statement>
</block>
```

Figure 4.29: XML representation of a block with value and statement inputs.

As it is already discussed, some blocks have a special output connection that allows them to act only like a value. Those blocks are connected into "value inputs" of other blocks. An example of this kind of block is the "math_number" block which can be shown in figure 3.6. In the XML format, this connection is denoted by nesting the block within the appropriate value input tag of the corresponding block (figure 4.30).

```
<block type="test_block">
    <value name="VALUE">
        <block type="math_number">
            <field name="NUM">123</field>
        </block>
    </value>
</block>
```

Figure 4.30: XML representation of a value input nested block structure.

In Blockly, both value connections and statement connections follow a similar XML representation pattern. However, statement connections differ in that they can encompass not just a single block, but a sequence of blocks.

Consider the main function of a program: it holds a series of code statements, each flowing in a particular order. This sequence can be lengthy, potentially containing numerous lines that must be executed in their given sequence. Blockly represents such sequences in its XML format using the "<next>" tag. This tag is a clear indicator that a block can be sequentially connected to other blocks. For the sake of clarity, almost all blocks in our context, except those serving only as value inputs (like "math_number"), contain this <next> tag. This includes the block representing the "main" function, as it's an "external_declaration" unit, allowing subsequent "external_declaration" units to connect to it.

Now, delving deeper into the statement connection structure, the first block in the sequence is nested within the "<statement>" tag of the preceding block. Before this block is closed off with its ending tag, it encapsulates the subsequent block within its "<next>" tag. This nested pattern continues, ensuring each block in the sequence is correctly nested within the "<next>" tag of the block before it.

To aid in understanding, consider an XML example that showcases three blocks connected in sequence, which is depicted in figure 4.31. This example provides a visual representation of the nested nature of blocks and how sequences are maintained using the "<next>" tag in Blockly's XML format.

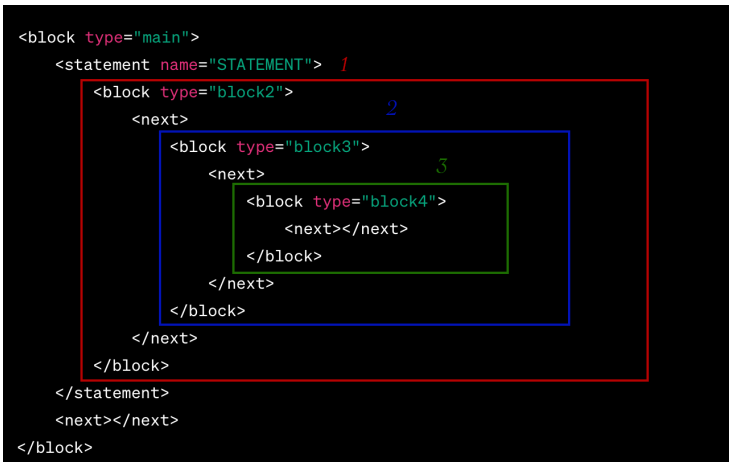


Figure 4.31: XML representation of statement connection structure.

Having explored all the topics of Blockly's XML format, the following illustration will present an actual XML of a block. In figure 4.32 it can be

shown the while loop block which is called "controls_whileUntil". This block contains a "dropdown field", a "value input" for the condition and "statement input" for all the code encapsulated within it. In the left section of the figure, the UI representation is displayed, while the right section showcases its XML counterpart.

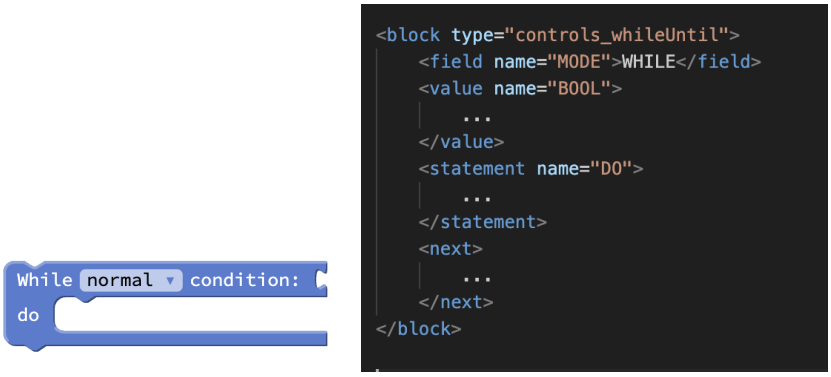


Figure 4.32: Comparison of UI representation and corresponding XML format for the while loop block.

As it is already discussed, in Blockly certain blocks can change their structure dynamically which are called "mutable". To represent these dynamic changes in the XML format, Blockly uses the "mutation" tag. A "mutation" tag doesn't typically encode the block's current state by itself. Instead, it provides additional data or attributes that the Blockly library uses to reconstruct the block's dynamic structure when loaded. Inside the "mutation" tag, custom attributes can be set to represent different states or configurations of the block. Blockly doesn't handle automatically the storage and reconstruct of those attributes. The functions **domToMutation** and **mutationToDom** of each mutable block cover that functionality. For instance, the mutable block "variables_declare_2" has 3 attributes which uses for storing the current state of the block. Those attributes are the inner variables which are called "myvariables", "globaltype" and "positionoftype". All those attributes are stored within the mutation tag. Its XML representation along with its corresponding UI can be shown in figure 4.33.


```
<block type="variables_declare_2">
    <mutation
        myvariables="[
                        ['myVariable','v'],
                        ['myVariable1','p'],
                        ['myVariable2','a'],
                        ['myVariable3','p']
                    ]"
        globaltype="int"
        positionoftype="1"
    ></mutation>
    <field name="TYPE">int</field>
    <field name="VAR0">myVariable</field>
    <field name="ITERATION1">*</field>
    <field name="VAR1">myVariable1</field>
    <field name="VAR2">myVariable2</field>
    <field name="LENGTH_1_2">1</field>
    <field name="LENGTH_2_2"></field>
    <field name="LENGTH_3_2"></field>
    <field name="ITERATION3">*</field>
    <field name="VAR3">myVariable3</field>
    <value name="VALUE0">
        ...
    </value>
    <value name="VALUE1">
        ...
    </value>
    <value name="VALUE2">
        ...
    </value>
    <value name="VALUE3">
    </value>
    <next>
        ...
    </next>
</block>
```

Figure 4.33: Comparison of UI representation and corresponding XML format for the "variables_declare_2" block.

Having explored the XML format of the blocks, the discussion now shifts to elucidating the embedded code associated with the grammar rules. The majority of the developed grammar rules have corresponding embedded code which it get executed whenever those grammar rules match with the parsed string. These embedded scripts are designed to translate smaller parts of the parsed string into XML, corresponding to Blockly's format. Then, through continuous layers of encapsulation, these XML snippets are combined, resulting in the 'total_code' unit at the topmost layer. Thereafter, the 'total_code' unit receives as input the entire converted code in an extensive XML document and returns it as a result with the code **"return($1);"** (figure 4.23).

However, some grammar rules does not have embedded code assigned to

them. There are various reasons for that. One reason is that some grammar shouldn't be converted to XML, such as EOF. Other rules are in a early encapsulation stage which there isn't yet a corresponding block to be converted to. For instance, an early stage grammar rule is the "type_specifier" unit which it can be shown in figure 4.34.

```
type_specifier
        : VOID
        | CHAR
        | SHORT
        | INT
        | LONG
        | FLOAT
        | DOUBLE
        | SIGNED
        | UNSIGNED
        | BOOL
        | COMPLEX
        | IMAGINARY     /* non-mandated extension */
        | atomic_type_specifier
        | struct_or_union_specifier
        | enum_specifier
        | TYPEDEF_NAME  /* after it has been defined as such */
        ;
```

Figure 4.34: Grammar Declaration for "type_specifier" unit.

As observed, none of the potential values within this stage have embedded code since it represents a preliminary phase that merely enumerates the valid variable types in C. At this point, there is no corresponding block for conversion.

Finally, one last reason is that some rules are already converted to their corresponding XML and they does not require further conversion. An example for that case is the "total_code" unit. As it can be seen in figure 4.23 the "total_code" unit has only one grammar rule, which is the **"transaltion_unit EOF"**. The embedded code of this grammar rule doesn't further modify the string that comes from the "translation_unit" because it is already in the desired format.

Those embedded codes within the Jison grammar perform XML string manipulations with the help of specialized variables. Those special variables are the **"$$"**, which represents the current grammar symbol's value, and indexed variables such as **"$1"**, **"$2"**, etc., which refer to the values of the

symbols on the right-hand side of the rule. In practice, **"$$"** is used to assign a value to a nonterminal symbol after the rule's conditions are met. Basically by using **"$$ = ..."** inside the embedded code, developers can set a value as output of this grammar rule. The indexed variables (**"$1"**, **"$2"** and so on) allow access to the matched input elements, providing the values that were recognized by the parser in each position of the rule's definition. For example, if a rule in the grammar matches a pattern that includes two different elements, the first matched element is accessible within the embedded code as **"$1"**, and the second as **"$2"**. The embedded codes that were implemented manipulate these values by concatenating strings of XML to construct the elements that represent the parsed code in the desired format.

Before dive into specific code implementations, it should be mentioned at this point that during the individual string manipulations, concatenations and conversions, some procedures were constantly repeated. So to enhance code clarity and usability, several utility functions were developed. The util functions can be shown in figures 4.35, 4.36 and 4.37.

```
function splitStringIntoThreeParts(totalString, preString, afterString){
    let a,b,c,index1,index2;
    index1 = totalString.indexOf(preString) + preString.length; // we add the length in order
    index2 = totalString.indexOf(afterString); // we don't add the length because we want to g
    if(index1-preString.length ==-1 || index2==-1) return [totalString,'',''];
    a = totalString.slice( 0 , index1 );
    b = totalString.slice( index1 , index2 );
    c = totalString.slice ( index2 );
    return [a,b,c];
}

function splitStringIntoTwoParts(totalString, preString){
    let a,b,index;
    index = totalString.indexOf(preString) + preString.length;
    if(index-preString.length ==-1) return [totalString,''];
    a = totalString.slice( 0 , index );
    b = totalString.slice ( index );
    return [a,b];
}

function splitStringIntoTwoPartsFindLast_PreString(totalString, preString){
    let a,b,index;
    index = totalString.lastIndexOf(preString);
    if(index==-1) return [totalString,''];
    index += preString.length;
    a = totalString.slice( 0 , index );
    b = totalString.slice( index );
    return [a,b];
}
```

Figure 4.35: Developed Util Functions in JavaScript for repetitive procedures (part 1).

```
function splitStringIntoTwoPartsFindLast_AfterString(totalString, afterString){
    let a,b,index;
    index = totalString.lastIndexOf(afterString);
    if(index==-1) return [totalString,''];
    a = totalString.slice( 0 , index );
    b = totalString.slice( index );
    return [a,b];
}

function checkIfItStartsWith(totalString, compareString){
    if(totalString.slice(0,compareString.length) == compareString){
        return true;
    }
    return false;
}

function convertToVariableGet(identifier){
    return '<block type="variables_get">'+
                '<field name="VAR">'+identifier+'</field>'+
            '</block>';
}

function convertToVariableSetExpression(identifier,operation,value){
    return '<block type="for_variables_set">'+
                '<field name="VAR">'+identifier+'</field>'+
                '<field name="OP">'+operation+'</field>'+
                '<value name="VALUE">'+value+'</value>'+
            '</block>';
}

function convertToArrayGet(string){ // string has format "identifier[number]"
    let temp = string.split('[');
    let varName = temp[0];
    temp.shift();
    temp = temp.map(elem => elem.replace("]", ""));
    let arrayLength = ['','',''];
    for(let i=0;i<temp.length;i++){
        if(temp[i]) arrayLength[i] = temp[i];
    }
    return '<block type="variables_array_get">'+
                '<field name="VAR">'+varName+'</field>'+
                '<field name="LENGTH_1">'+arrayLength[0]+'</field>'+
                '<field name="LENGTH_2">'+arrayLength[1]+'</field>'+
                '<field name="LENGTH_3">'+arrayLength[2]+'</field>'+
            '</block>';
}
```

Figure 4.36: Developed Util Functions in JavaScript for repetitive procedures (part 2).

```
function convertToArraySetExpression(string,operation,value){
    let temp = String(string).split('[');
    let varName = temp[0];
    temp.shift();
    temp = temp.map(elem => elem.replace("]", ""));
    let arrayLength = ['','',''];
    for(let i=0;i<temp.length;i++){
        if(temp[i]) arrayLength[i] = temp[i];
    }
    return '<block type="exp_array_set">'+
                '<field name="VAR">'+varName+'</field>'+
                '<field name="LENGTH_1">'+arrayLength[0]+'</field>'+
                '<field name="LENGTH_2">'+arrayLength[1]+'</field>'+
                '<field name="LENGTH_3">'+arrayLength[2]+'</field>'+
                '<field name="OP">'+operation+'</field>'+
                '<value name="VALUE">'+value+'</value>'+
            '</block>';
}

function extractArrayInfo(arrayString){
    let temp = String(arrayString).split('[')
    let varName = temp[0]
    temp.shift()
    temp = temp.map(elem => elem.replace("]", ""))
    let arrayLength = ['','','']
    for(let i=0;i<temp.length;i++){
        if(temp[i]) arrayLength[i] = temp[i]
    }
    return [varName,arrayLength[0],arrayLength[1],arrayLength[2]];
}

function extractPointerInfo(pointerString){
    let temp = String(pointerString).split('*')
    let stars = ''
    for(let i=0;i<temp.length-1;i++){
        stars = stars + '*'
    }
    return [temp[temp.length-1],stars];
}

function encapsulateStatements(totalStringXML, newStringXML){
    let index = String(totalStringXML).lastIndexOf('<next>')+6
    let newTotal = [String(totalStringXML).slice(0, index),String(newStringXML), String(totalStringXML).slice(index)].join('')
    return newTotal
}
```

Figure 4.37: Developed Util Functions in JavaScript for repetitive procedures (part 3).

The splitStringIntoThreeParts function is designed to dissect a string into three segments based on predefined substrings, facilitating granular control over string manipulation. Similarly, splitStringIntoTwoParts cuts a string into two at the first occurrence of a specified substring, while its variants, splitStringIntoTwoPartsFindLast_PreString and splitStringIntoTwoPartsFindLast_AfterString, serve to locate and split the string at the last occurrence of given substrings. These functions are indispensable for parsing and reorganizing strings in a precise manner.

The checkIfItStartsWith function plays a critical role in syntax checking by validating whether a string begins with a specific sequence of characters.

For the creation of XML blocks that reflect Blockly's visual elements, convertToVariableGet transforms a variable identifier into its XML representation for the "variables_get" block, and convertToVariableSetExpression constructs the "for_variables_set" XML block representation. The convertToVariableSetExpression functions has as inputs the name of the variable, the operation symbol (for example "=" or "+=" etc.) and variable's value.

When dealing with arrays, convertToArrayGet and convertToArraySetExpression adeptly handle the generation of XML blocks for array access and assignment, even for multi-dimensional arrays, by interpreting and integrating index values. The extractArrayInfo and extractPointerInfo functions are tailored to extract critical information from strings that denote arrays and pointers, breaking them down into component parts such as variable names and array dimensions or pointer levels.

A key function in the suite is encapsulateStatements, which is essential for the sequential composition of code blocks. It embeds a new XML string within an existing one by inserting it within a <next> tag, ensuring the correct order of execution in the generated code. Collectively, these utility functions are engineered to reduce the intricacies of direct string operations, thereby providing a simplified, reliable method for generating the XML structures that underpin Blockly's visual code representation in the system.

Now that we have covered the utility functions too let's explain what the embedded code does in the figure 4.23. Building upon the description of utility functions, let's delve into the logic embedded within the grammar rules in Figure 4.23. It is already discussed the logic behind the "total_code" unit. For the translation_unit rule, it serves as the root and can either be a single external_declaration or a combination of multiple declarations. In the case of a single declaration, the resulting XML is straightforwardly returned. However, when there are multiple declarations, the embedded code takes on the task of combining them into a cohesive XML document. It does so by finding the last occurrence of the <next> tag in the current XML string and then merging the new declaration XML string at this point. This ensures that each code block is correctly sequenced, reflecting the order of declarations in the source code.

Within the external_declaration unit, there are three grammar rules. Either contains a function_definition, a declaration or a define declaration. In case of the function, the resulting XML is straightforwardly returned. In Jison, the default action for a grammar rule without an explicitly defined

embedded code is to pass through the first element unmodified. In other words, if no specific actions are provided, Jison will implicitly execute **"$$ = $1"**, assigning the value of the first element to the output.

When encountering a declaration rule, two distinct block types may be inputted. if the declaration is a "structure_define" block then it is directly returned without modification. For the "exp_variable_declare" block case, a <next> tag is appended to allow for sequential execution in the Blockly environment. The reasons for the "exp_variable_declare" block lacking an embedded <next> tag initially will be examined subsequently. Additionally, the embedded code handles exceptions with a safety net. If an unrecognized declaration is encountered, an error is thrown, flagging an unexpected or unhandled case. This ensures that only known and correctly formatted XML is constructed, preserving the integrity of the resulting Blockly code structure. In addition it provides a usefull message to the user indicating the source and the position of the error.

Another importance aspect of the grammar is the **"function_definition"** unit. This unit plays a crucial role in generating the blocks corresponding to the main function as well as all other custom function declarations. There are two distinct types of blocks for declaring custom functions: **"procedures_defnoreturn"** and **"procedures_defreturn"**. This distinction stems from the fact that a function in C can be a void (not returning any value) or can return a specific type of variable. For functions that return a value, the corresponding block includes additional fields to select the variable to be returned at the function's conclusion. Figures 4.38 and 4.39 illustrate the visual and XML representations of these blocks. Additionally, both block types are designed to be mutable, accommodating scenarios where a function may not require input variables or might need multiple input variables as defined by the programmer.

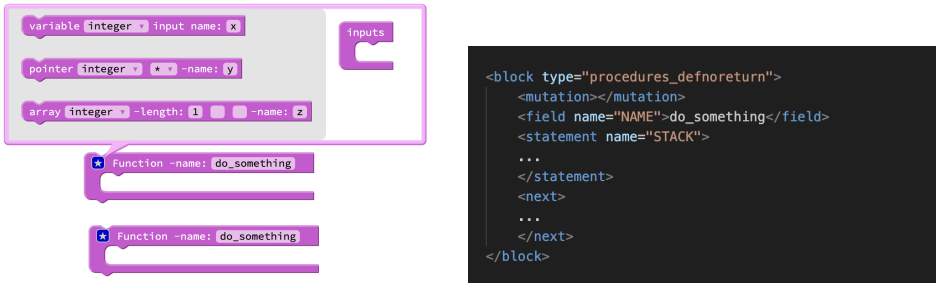


Figure 4.38: Comparison of UI representation and corresponding XML format for "procedures_defnoreturn" block.

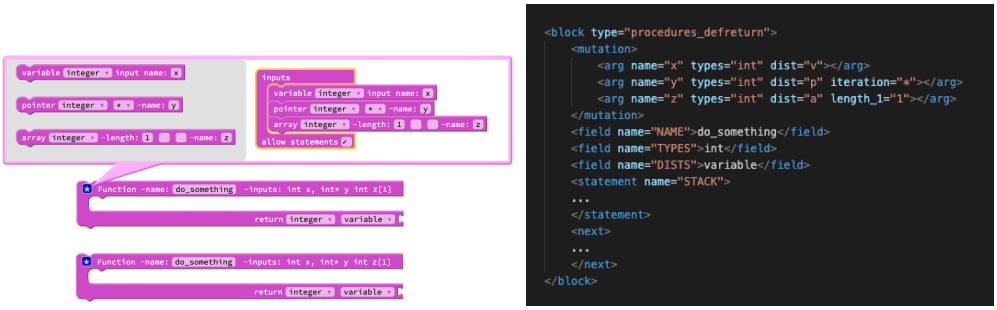


Figure 4.39: Comparison of UI representation and corresponding XML format for "procedures_defreturn" block.

The code for the **"function_definition"** unit can be seen in figure 4.40.

```
function_definition
    : declaration_specifiers declarator declaration_list compound_statement {{
        throw new Error('old-style parameter declarations in prototyped function definition');
        }}
    | declaration_specifiers declarator compound_statement {{
            if($2 == 'main'){
                $$ = '<block type="main_block" id="1" inline="false" deletable="false" editable="false">
                        <statement name="STACK">'+$3+'</statement>
                        <value name="RETURN"></value>
                        <next></next>
                    </block>';
            }else{


                // remove arguments from $2
                let arguments = ''
                if(String($2).includes('<arg')){
                    arguments = String($2).slice(String($2).indexOf('<arg'))
                    $2 = String($2).slice(0,String($2).indexOf('<arg'))
                }

                if($1 == 'void'){
                    $$ = '<block type="procedures_defnoreturn">
                                <mutation>'+arguments+'</mutation>
                                <field name="NAME">'+$2+'</field>
                                <statement name="STACK">'+$3+'</statement>
                                <next></next>
                            </block>'
                }else{
                    if(String($2).includes('*')){
                        let funcName,stars
                        [funcName,stars] = extractPointerInfo($2)
                        $$ = '<block type="procedures_defreturn">
                                    <mutation>'+arguments+'</mutation>
                                    <field name="NAME">'+funcName+'</field>
                                    <field name="TYPES">'+$1+'</field>
                                    <field name="DISTS">pointer</field>
                                    <field name="PSPECS">'+stars+'</field>
                                    <statement name="STACK">'+$3+'</statement>
                                    <value name="RETURN"></value>
                                    <next></next>
                                </block>'
                    }else if(String($2).includes('[')){ // it can't have array as a type
                        throw new Error('error: '+$2+' declared as function returning an array');
                    }else{
                        $$ = '<block type="procedures_defreturn">
                                    <mutation>'+arguments+'</mutation>
                                    <field name="NAME">'+$2+'</field>
                                    <field name="TYPES">'+$1+'</field>
                                    <field name="DISTS">variable</field>
                                    <statement name="STACK">'+$3+'</statement>
                                    <value name="RETURN"></value>
                                    <next></next>
                                </block>'
                    }
                }
            }
        }}
    ;
```


Figure 4.40: Grammar rules and embedded code for "function_definition" unit.

The code distinguishes between the main function and custom functions by evaluating whether the "declarator" is equivalent to "main". This approach effectively differentiates the main function from user-defined functions in the input. When the parsed string is identified as the main function, the code generates a specific block of type main_block. This block is marked as non-deletable and non-editable, signifying its essential role in any C program. The compound_statement ($3 in the code) from the function definition is placed within a statement tag named "STACK", indicating the main body of the function.

For custom functions, the code differentiates between void functions and those returning a value. This is discerned from the declaration_specifiers segments of the function definition by checking if it equal with "void" or not. If the function is void, a procedures_defnoreturn block is created. It includes a mutation tag to handle function arguments, a field tag for the function name, and the function body within the statement tag.

If the function returns a value, a procedures_defreturn block is created. This block also contains a mutation tag for arguments, field tags for the function name and return type, and tags for STACK and RETURN to accommodate the function body and return value, respectively. If the function with return value returns a pointer, the code extracts the number of stars and alternates the XML format accordingly.

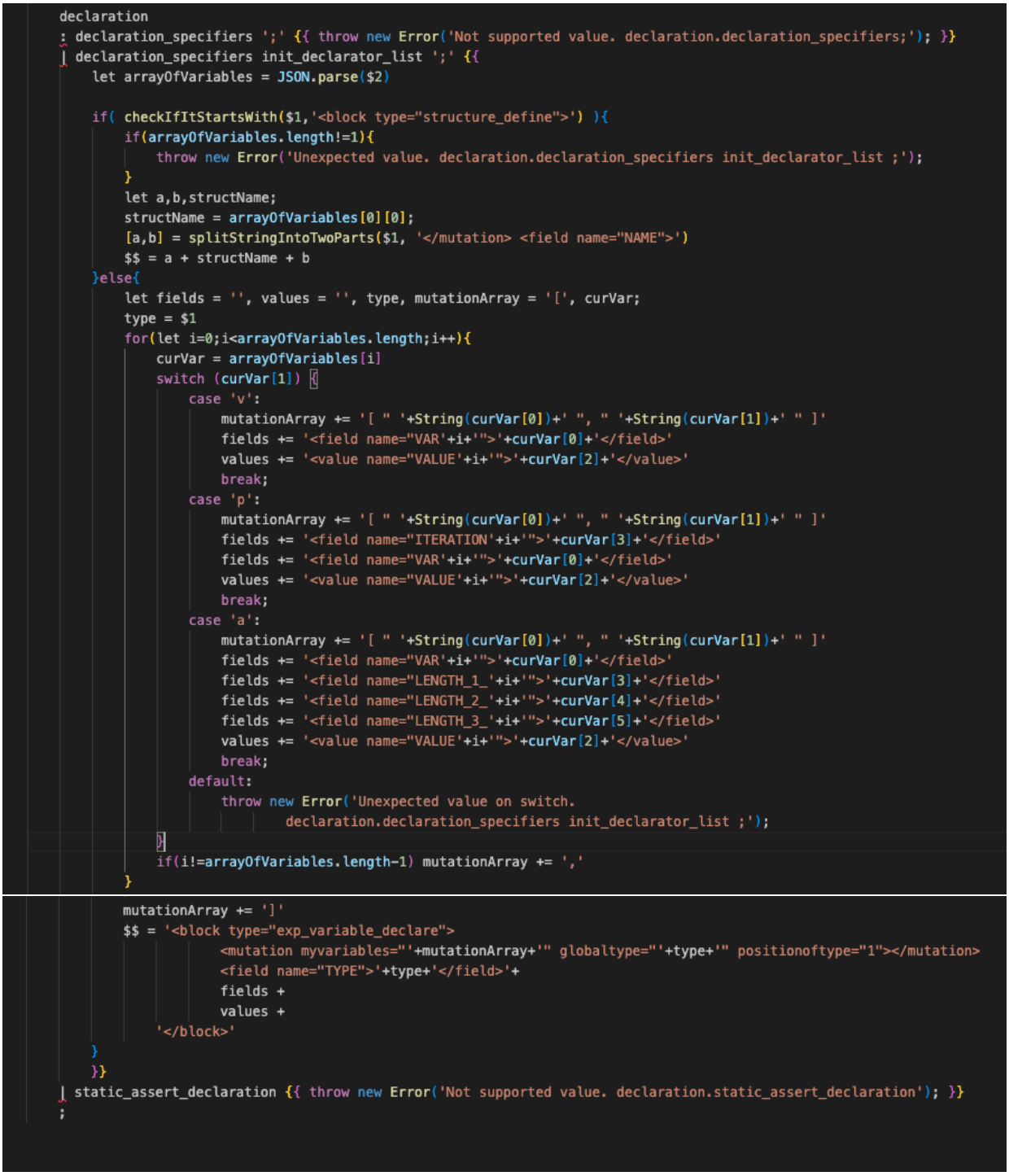
When the "declarator" segment input represents a custom function with arguments, it's already formatted into specific XML through prior encapsulations. Consequently, no further conversion is necessary, except for separating the function's name from its arguments. This separation is efficiently managed using JavaScript's string manipulation functions. Subsequently, these arguments are inserted within the 'mutation' tag. In cases where there are no arguments, an empty string is appropriately positioned inside the tag.

The code includes safeguards against incorrect function definitions, such as functions declared to return an array, which is syntactically incorrect in C. In such cases, an error is thrown to alert the user or the system.

The block "exp_variable_declare" stands as a pivotal and intricate component in Blockly's structure, as depicted in Figure 4.33. Both its user interface and XML representations are showcased in this figure. Characterized by its mutability, this block is adept at declaring multiple variables within a single code line, with commas separating each declaration. An example of its usage could be: **int myVar1 = 1, *myVar2, myVar3[1][2][3], **my-**

Var4;.

The creation of the "exp_variable_declare" block is handled within the "declaration" unit of the grammar. This unit, along with its associated embedded code, is detailed in Figure 4.41. It plays a crucial role in parsing and transforming C language declarations into their corresponding Blockly XML format, thereby enabling the dynamic creation of variable declaration blocks within the Blockly framework.



```
declaration
: declaration_specifiers ';' {{ throw new Error('Not supported value. declaration.declaration_specifiers;'); }}
| declaration_specifiers init_declarator_list ';' {{
    let arrayOfVariables = JSON.parse($2)

    if( checkIfItStartsWith($1,'<block type="structure_define">') ){
        if(arrayOfVariables.length!=1){
            throw new Error('Unexpected value. declaration.declaration_specifiers init_declarator_list ;');
        }
        let a,b,structName;
        structName = arrayOfVariables[0][0];
        [a,b] = splitStringIntoTwoParts($1, '</mutation> <field name="NAME">')
        $$ = a + structName + b
    }else{
        let fields = '', values = '', type, mutationArray = '[', curVar;
        type = $1
        for(let i=0;i<arrayOfVariables.length;i++){
            curVar = arrayOfVariables[i]
            switch (curVar[1]) {
                case 'v':
                    mutationArray += '[ " '+String(curVar[0])+' ", " '+String(curVar[1])+' " ]'
                    fields += '<field name="VAR'+i+'">'+curVar[0]+'</field>'
                    values += '<value name="VALUE'+i+'">'+curVar[2]+'</value>'
                    break;
                case 'p':
                    mutationArray += '[ " '+String(curVar[0])+' ", " '+String(curVar[1])+' " ]'
                    fields += '<field name="ITERATION'+i+'">'+curVar[3]+'</field>'
                    fields += '<field name="VAR'+i+'">'+curVar[0]+'</field>'
                    values += '<value name="VALUE'+i+'">'+curVar[2]+'</value>'
                    break;
                case 'a':
                    mutationArray += '[ " '+String(curVar[0])+' ", " '+String(curVar[1])+' " ]'
                    fields += '<field name="VAR'+i+'">'+curVar[0]+'</field>'
                    fields += '<field name="LENGTH_1_'+i+'">'+curVar[3]+'</field>'
                    fields += '<field name="LENGTH_2_'+i+'">'+curVar[4]+'</field>'
                    fields += '<field name="LENGTH_3_'+i+'">'+curVar[5]+'</field>'
                    values += '<value name="VALUE'+i+'">'+curVar[2]+'</value>'
                    break;
                default:
                    throw new Error('Unexpected value on switch.
                            declaration.declaration_specifiers init_declarator_list ;');
                }
                if(i!=arrayOfVariables.length-1) mutationArray += ','
            }
            mutationArray += ']'
            $$ = '<block type="exp_variable_declare">
                    <mutation myvariables="'+mutationArray+'" globaltype="'+type+'" positionoftype="1"></mutation>
                    <field name="TYPE">'+type+'</field>'+
                    fields +
                    values +
                '</block>'
        }
    }}
| static_assert_declaration {{ throw new Error('Not supported value. declaration.static_assert_declaration'); }}
;
```

Figure 4.41: Grammar rules and embedded code for "declaration" unit.

This unit is defined with three rules, among which the only valid one is the **"declaration_specifiers init_declarator_list ;"**. The other two rules, upon invocation, throw errors with descriptive messages indicating unsupported values. The primary rule handles two different outputs: a "structure_define" block representing a structure declaration, and the "exp_variable_declare" block for variable declarations.

The segment "declaration_specifiers" contains either the specifier of variables (like "int") or the XML for a "structure_define" block. These two cases are differentiated based on whether the "declaration_specifiers" starts with '<block type="structure_define">'. In instances where the declaration begins with a structure definition, the code expects only one variable in the "init_declarator_list," which is the structure's name. An error is raised if this is not met, and it extracts the structure's name and injects it into the existing XML format at the correct position, adjusting the XML content accordingly.the appropriate position.

For variable declarations, the "init_declarator_list" segment encompasses all details about the variables declared in that line. Here, a unique approach was used in earlier encapsulations of "init_declarator_list" to simplify the storage and retrieval of declared variables. At more primal stages of encapsulation, it's easier to identify whether a declared variable is a simple variable, a pointer, or an array, as each has distinct grammar rules. For example, the rule for a pointer is **": pointer direct_declarator ... "**. At these stages, essential details of each variable are stored in a JavaScript array. The format varies based on the variable type: simple variables use "[nameOfVariable, 'v', valueOfVariable]", pointers use "[nameOfVariable, 'p', valueOfVariable, stars]", and arrays use "[nameOfVariable, 'a', valueOfVariable, length1, length2, length3]". In these formats, 'nameOfVariable' represents the variable's identifier and 'valueOfVariable' is the assigned value to the variable. For simple variables, 'v' indicates a basic variable type. In the pointer format, 'p' denotes a pointer type and 'stars' reflects the pointer's level (e.g., *, **). For arrays, 'a' signals an array variable and 'length1', 'length2', 'length3' specify the dimensions of the array, accommodating up to three levels of depth. There are only three depths because three is the maximum dimensions that an array may have in Blockly.

The JSON format is employed for string conversions, utilizing JavaScript's "JSON.stringify()" and "JSON.parse()" functions for converting variables to and from strings.

To manage variable storage and retrieval across multiple encapsulation

layers within the embedded code, the JSON format supported by JavaScript was employed. This approach utilizes two key JavaScript functions: JSON.stringify() and JSON.parse(). The JSON.stringify() function takes any JavaScript variable as input, converts it into a JSON-formatted string, and outputs this string. Conversely, JSON.parse() performs the inverse operation. This methodology allows for the conversion of each variable, formatted as "[nameOfVariable, 'v', valueOfVariable]", into a JSON string at the end of each embedded code segment. This string is then passed to outer layers of encapsulation. For instance, a variable can be stringified and assigned using **let currentVariable = [nameOfVariable, 'v', valueOfVariable]; $$ = JSON.stringify(currentVariable);**. At the next level of encapsulation, this stringified variable can be reconstituted into its original form by using **let currentVariable = JSON.parse($1);**. This process ensures a seamless transition of data between different layers of the code.

The "init_declarator_list" unit can be shown in figure 4.42.

```
init_declarator_list
: init_declarator {{
    $$ = '['+$1+']'
}}
| init_declarator_list ',' init_declarator {{
    let array = JSON.parse($1)
    $3 = JSON.parse($3)
    array.push($3)
    $$ = JSON.stringify(array)
}}
;
```

Figure 4.42: Grammar rules and embedded code for "init_declarator_list" unit.

The 'init_declarator_list' unit gathers individual variable declarations, consolidating them into a single array that is then passed on for further processing in the 'declaration' unit. Within "declaration" unit, as depicted in figure 4.41, the array of variables is retrieved from the 'init_declarator_list' segment by using JSON.parse(). This process results in the formation of a local array, designated as 'arrayOfVariables'. The code then iteratively processes each variable within this array through a 'for' loop, applying the

necessary logic to construct the appropriate XML structure for each variable type. The code employs a switch statement to identify each variable's type, using 'v', 'p', or 'a' as markers for simple variables, pointers, or arrays, respectively. Within the for loop, it assembles the 'mutationArray', 'fields', and 'values' as string constructs. Ultimately, this process culminates in the formation of the XML structure for the 'exp_variable_declare' block. This structure incorporates the 'mutationArray' within its mutation tag and correctly positions the dynamically generated 'fields' and 'values' within the block's XML framework.

Delving further into the parsing structure, every line of code that it can be placed within the brackets of the main or a custom function is a "block_item" (figure 4.43). The "block_item" unit can either be a "declaration", a "statement" or a "define" variable. The "statement" unit can be shown in figure 4.44.

```
block_item
: declaration {{
    if( checkIfItStartsWith($1,'<block type="exp_variable_declare"') ){
        $1 = String($1).replace('<block type="exp_variable_declare"', '<block type="variables_declare_2"')
        let a,b;
        [a,b] = splitStringIntoTwoPartsFindLast_AfterString($1, '</block>')
        $$ = a.concat('<next></next>').concat(b)
    }else {
        throw new Error('Exception, uknown type in block_item.declaration')
    }
    }}
| statement
| define_all
;
```

Figure 4.43: Grammar rules and embedded code for "block_item" unit.

```
statement
: labeled_statement
| compound_statement
| expression_statement {{
    if($1==''){
        $$ = '<block type="expression_statement"><value name="VAR0"></value><next></next></block>'
    }else{
        if( checkIfItStartsWith($1, '<block type="expressions"') ){
            $1 = String($1).replace('<block type="expressions"', '<block type="expression_statement"')
            let a,b;
            [a,b] = splitStringIntoTwoPartsFindLast_AfterString($1, '</block>')
            $$ = a.concat('<next></next>').concat(b)
        }else{
            $$ = '<block type="expression_statement"><value name="VAR0">'+$1+'</value><next></next></block>'
        }
    }
    }}
| selection_statement
| iteration_statement
| jump_statement
;
```

Figure 4.44: Grammar rules and embedded code for "statement" unit.

The "statement" unit encompasses a variety of forms, including "la-

beled_statement", "compound_statement", "expression_statement", "selection_statement", "iteration_statement", and "jump_statement". Each of these categories represents a distinct type of statement within the programming structure. In C language, an expression statement is essentially a line of code that performs a specific action, such as assigning a value, calling a function, or manipulating data. It is an any type of expression that ends with a semicolon (;). For example, x = y + 2; is an expression statement. Selection statements in C, such as if, else if, else, and switch, are conditional constructs that steer the execution flow based on specific conditions (figure 4.45). Iteration statements, encompassing the while, do-while, and for loops, repeatedly execute code blocks based on defined criteria (figure 4.46). Lastly, jump statements like goto, continue, return, and break are used to alter the execution sequence, either by moving to different sections of code, continuing or exiting loops, or returning from functions (figure 4.47).

```
selection_statement
    : IF '(' expression ')' statement ELSE statement {{
        if( checkIfItStartsWith($7,'<block type="controls_if"') ){
            $7 = String($7);
            // find how many elseif we have and add one more
            let temp = $7.slice($7.indexOf('<mutation')+9);
            temp = temp.slice(0,temp.indexOf('>'));
            if(temp.includes('elseif="')){
                temp = temp.split(" ");
                temp = temp[1]
                temp = temp.split('"');
                temp = temp[1];
                temp =  parseInt(temp);
                temp = String(parseInt(temp+1));
                $7 = ($7.slice(0,$7.indexOf('elseif="')+8)).concat(temp).concat($7.slice($7.indexOf('" else="')));
            }else{
                temp = 1;
                $7 = ($7.slice(0,$7.indexOf(' else="'))).concat(' elseif="1"').concat($7.slice($7.indexOf(' else="')));
            }
            for(let i=temp;i>0;i--){
                $7 = $7.replace('IF'+parseInt(i-1),'IF'+i);
                $7 = $7.replace('DO'+parseInt(i-1),'DO'+i);
            }
            $7 = $7.slice(0,$7.indexOf('<value name="IF1">'))
                    .concat('<value name="IF0">'+$3+'</value> <statement name="DO0">'+$5+'</statement> ')
                    .concat($7.slice($7.indexOf('<value name="IF1">')));
            $$ = $7;
        }else{
            // first time that we have if-else
            $$ = '<block type="controls_if" inline="false">
                        <mutation else="1"></mutation>
                        <value name="IF0">'+$3+'</value>
                        <statement name="DO0">'+$5+'</statement>
                        <statement name="ELSE">'+$7+'</statement>
                        <next></next>
                    </block>'
        }
        }}
    | IF '(' expression ')' statement {{
        $$ = '<block type="controls_if" inline="false">
                    <value name="IF0">'+$3+'</value>
                    <statement name="DO0">'+$5+'</statement>
                    <next></next>
                </block>'
        }}
```

```
    | SWITCH '(' expression ')' statement {{
        $5 = '<removalTag>' + $5 + '</removalTag>';
        let tempXML = new DOMParser().parseFromString($5,"text/xml").childNodes[0];
        let case_number = -1;
        for (var x = 0, xmlChild; xmlChild = tempXML.childNodes[x]; x++) {
            if (xmlChild.nodeType == 3 && xmlChild.data.match(/^\s*$/)) {
                continue;
            }
            var name = xmlChild.getAttribute('name');
            switch (xmlChild.nodeName.toLowerCase()){
                case 'value':
                    if(name=='CASE0'){
                        case_number = case_number + 1;
                        tempXML.childNodes[x].setAttribute('name', 'CASE'+case_number);
                    }
                    break;
                case 'statement':
                    if(name=='DO0') tempXML.childNodes[x].setAttribute('name', 'DO'+case_number);
                    break;
                default:
                    // Unknown tag; ignore.  Same principle as HTML parsers.
            }
        }
        $5 = new XMLSerializer().serializeToString(tempXML);
        $5 = removeCollapsedTags($5);
        $5 = $5.replace('<removalTag>','');
        $5 = $5.replace('</removalTag>','');
        $$ = '<block type="controls_switch" inline="false">
                    <mutation case="'+case_number+'"></mutation>
                    <value name="SWITCH">'+$3+'</value> '+
                    $5 +
                    '<next></next>
                </block>';
        }}
    ;
```

Figure 4.45: Grammar rules and embedded code for "selection_statement" unit.

Figure 4.46: Grammar rules and embedded code for "iteration_statement" unit.

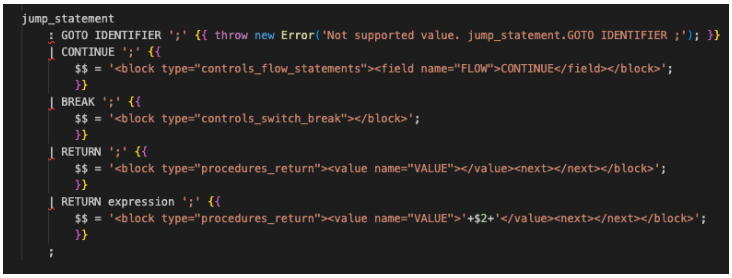


Figure 4.47: Grammar rules and embedded code for "jump_statement" unit.

The statements printf("Hello World") and scanf(“%d”, &var); are examples of expression statements, as are myCustomeFunction(); and x = myFunc(3);. Essentially, any function call in C, whether it’s a custom function or one from a library, is considered an expression statement. C offers a variety of libraries like stdio, stdlib, string, math, and time, each containing useful functions. Key utility functions from these libraries have been developed into blocks for Blockly, as illustrated in Figures 4.48 and 4.49. The grammar rules governing these function calls are encapsulated within the postfix_expression unit, detailed in Figure 4.50.

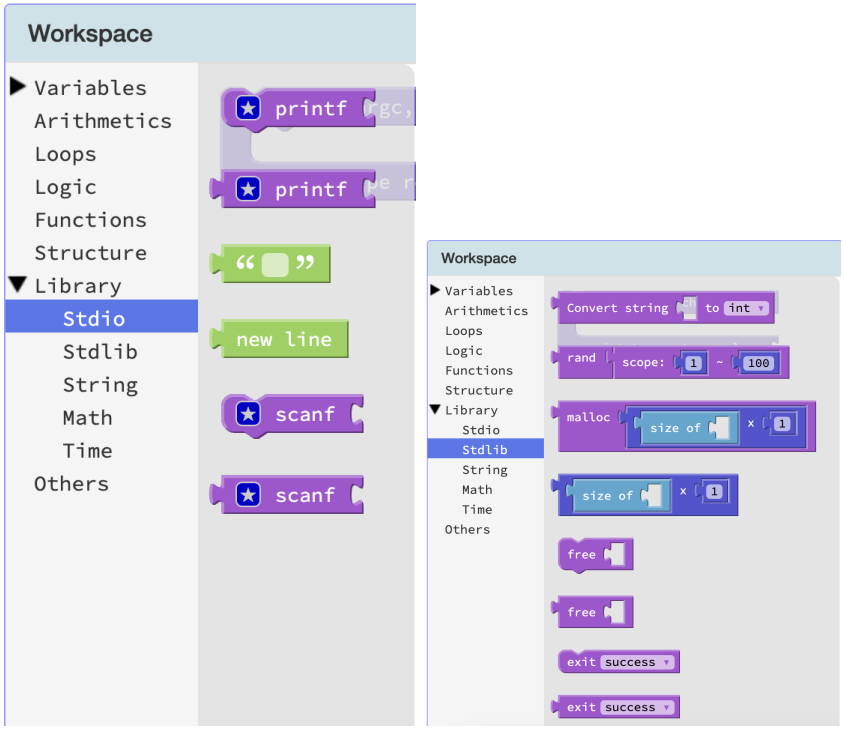


Figure 4.48: Block representations of Util functions from popular C libraries (part 1).

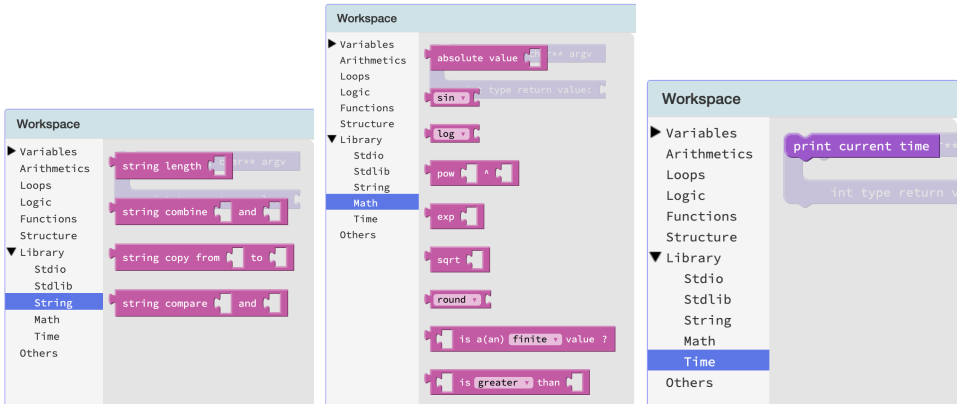


Figure 4.49: Block representations of Util functions from popular C libraries (part 2).

Figure 4.50: Grammar rules and embedded code for "postfix_expression" unit.

The grammar rules that are specific for functions calls are the **"postfix_expression ( )"** and **"postfix_expression (argument_expression_list)"**. Basically, those two rules are the same except that one has arguments within the parenthesis and the other does not. To streamline the code and enhance its usability, a utility function named createFunctionsAndLibraries was developed. This function is invoked for both those rules, taking the function name and the arguments as parameters. It then returns the appro-

priate XML block conversion. For instance, in the case of a function call, the assignment would be either $$ = createFunctionsAndLibraries($1, $3) or $$ = createFunctionsAndLibraries($1, null), depending on whether arguments are present. The implementation code of this utility function are shown in Figures 4.51, 4.52, 4.53, 4.54, 4.55, 4.56 and 4.57.

```
function createFunctionsAndLibraries(funcName,arguments){
    if( checkIfItStartsWith( funcName, '<block') ){
        throw new Error('Unknown type. Expecting not a block, we found = ',funcName,'. postfix_expression.functionAndLibraries');
    }

    if(arguments != null) arguments = String(arguments).split('<argument_expression_list_comma>')
    let argumentsString = '',printadd = '0',mutationScanf = ''

    switch (funcName) {
        case 'printf':
            for(let i=0; arguments != null && i<arguments.length; i++){
                if(arguments[i]){
                    if(checkIfItStartsWith(arguments[i],'<block type="library_stdio_text">')){
                        const formatSpecifierRegex = /%[a-z]/gi;
                        arguments[i] = arguments[i].replace(formatSpecifierRegex, '');
                    }
                    argumentsString += '<value name="VAR'+i+'">'+arguments[i]+'</value>';
                }
            }
            printadd = (arguments!= null&&arguments.length>0)? String(arguments.length-1) : '0';
            if(argumentsString=='')argumentsString = '<value name="VAR0"></value>'
            return '<block type="exp_printf">'+
                        '<mutation printadd="'+printadd+'"></mutation>'+
                        argumentsString+
                    '</block>'
        case 'scanf':
            for(let i=1; arguments != null && i<arguments.length; i++){
                if(arguments[i]){
                    if( checkIfItStartsWith(arguments[i], '<block type="variables_pointer_&"') ){
                        arguments[i] = arguments[i].replace('<block type="variables_pointer_&"><value name="VALUE">','')
                        arguments[i] = arguments[i].replace('</value></block>','')
                    }
                    argumentsString += '<value name="VAR'+(i-1)+'">'+arguments[i]+'</value>';
                }
            }
            printadd = (arguments!= null&&arguments.length>0)? String(arguments.length-1) : '0';
            if(argumentsString=='')argumentsString = '<value name="VAR0"></value>'
            if(arguments.length>2) mutationScanf = '<mutation scanadd="'+(printadd-1)+'"></mutation>'
            return '<block type="exp_scanf">'+ mutationScanf + argumentsString +'</block>'
        case 'atoi':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_stdlib_convert">'+
                                '<field name="OPERATORS">INT</field>'+
                                '<value name="VAR">'+arguments[0]+'</value>'+
                    '</block>'
```

Figure 4.51: Utility function "createFunctionsAndLibraries" (part 1).

```
        case 'atof':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                            'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_stdlib_convert">'+
                        '<field name="OPERATORS">DOUBLE</field>'+
                        '<value name="VAR">'+arguments[0]+'</value>'+
                    '</block>'
        case 'rand':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_stdlib_rand">'+
                        '<value name="VAR">'+arguments[0]+'</value>'+
                    '</block>'
        case 'malloc':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_stdlib_malloc">'+
                        '<value name="VAR">'+arguments[0]+'</value>'+
                    '</block>'
        case 'free':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="exp_free">'+
                        '<value name="VAR">'+arguments[0]+'</value>'+
                    '</block>'
        case 'exit':
            return '<block type="exp_exit">'+
                        '<field name="OPERATORS">SUCCESS</field>'+
                    '</block>'
        case 'strlen':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_string_strlen">'+
                        '<value name="VAR">'+arguments[0]+'</value>'+
                    '</block>'
        case 'strcat':
            if(arguments==null)arguments = ['','']
            if(arguments.length!=2) throw new Error('Invalid value. Expecting 2 arguments, '+
                                'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_string_strcat">'+
                        '<value name="STR1">'+arguments[0]+'</value>'+
                        '<value name="STR2">'+arguments[1]+'</value>'+
                    '</block>'
```

Figure 4.52: Utility function "createFunctionsAndLibraries" (part 2).

```
        case 'strcpy':
            if(arguments==null)arguments = ['',''']
            if(arguments.length!=2) throw new Error('Invalid value. Expecting 2 arguments, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_string_strcpy">'+
                        '<value name="STR1">'+arguments[0]+'</value>'+
                        '<value name="STR2">'+arguments[1]+'</value>'+
                    '</block>'
        case 'strcmp':
            if(arguments==null)arguments = ['',''']
            if(arguments.length!=2) throw new Error('Invalid value. Expecting 2 arguments, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_string_strcmp">'+
                        '<value name="STR1">'+arguments[0]+'</value>'+
                        '<value name="STR2">'+arguments[1]+'</value>'+
                    '</block>'
        case 'abs':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_abs">'+
                        '<value name="VAR">'+arguments[0]+'</value>'+
                    '</block>'
        case 'sin':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_trig">'+
                        '<field name="OP">SIN</field>'+
                        '<value name="NUM">'+arguments[0]+'</value>'+
                    '</block>'
        case 'cos':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_trig">'+
                        '<field name="OP">COS</field>'+
                        '<value name="NUM">'+arguments[0]+'</value>'+
                    '</block>'
        case 'tan':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_trig">'+
                        '<field name="OP">TAN</field>'+
                        '<value name="NUM">'+arguments[0]+'</value>'+
                    '</block>'
```

Figure 4.53: Utility function "createFunctionsAndLibraries" (part 3).

```
        case 'log':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_logs">'+
                        '<field name="OP">LOG</field>'+
                        '<value name="NUM">'+arguments[0]+'</value>'+
                    '</block>'
        case 'log10':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_logs">'+
                        '<field name="OP">LOG10</field>'+
                        '<value name="NUM">'+arguments[0]+'</value>'+
                    '</block>'
        case 'log2':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_logs">'+
                        '<field name="OP">LOG2</field>'+
                        '<value name="NUM">'+arguments[0]+'</value>'+
                    '</block>'
        case 'pow':
            if(arguments==null)arguments = ['','']
            if(arguments.length!=2) throw new Error('Invalid value. Expecting 2 arguments, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_pow">'+
                        '<value name="BASE">'+arguments[0]+'</value>'+
                        '<value name="EXPO">'+arguments[1]+'</value>'+
                    '</block>'
        case 'exp':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_exp">'+
                        '<value name="EXPO">'+arguments[0]+'</value>'+
                    '</block>'
        case 'sqrt':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_sqrt">'+
                        '<value name="VAR">'+arguments[0]+'</value>'+
                    '</block>'
```

Figure 4.54: Utility function "createFunctionsAndLibraries" (part 4).

```
        case 'round':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_round">'+
                        '<field name="OP">ROUND</field>'+
                        '<value name="NUM">'+arguments[0]+'</value>'+
                    '</block>'
        case 'ceil':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_round">'+
                        '<field name="OP">CEIL</field>'+
                        '<value name="NUM">'+arguments[0]+'</value>'+
                    '</block>'
        case 'floor':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_round">'+
                        '<field name="OP">FLOOR</field>'+
                        '<value name="NUM">'+arguments[0]+'</value>'+
                    '</block>'
        case 'trunc':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_round">'+
                        '<field name="OP">TRUNC</field>'+
                        '<value name="NUM">'+arguments[0]+'</value>'+
                    '</block>'
        case 'isfinite':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_numcheck">'+
                        '<field name="CONDITIONS">ISFINITE</field>'+
                        '<value name="VAR">'+arguments[0]+'</value>'+
                    '</block>'
        case 'isinf':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_numcheck">'+
                        '<field name="CONDITIONS">ISINF</field>'+
                        '<value name="VAR">'+arguments[0]+'</value>'+
                    '</block>'
```

Figure 4.55: Utility function "createFunctionsAndLibraries" (part 5).

```
        case 'signbit':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_numcheck">'+
                        '<field name="CONDITIONS">SIGNBIT</field>'+
                        '<value name="VAR">'+arguments[0]+'</value>'+
                    '</block>'
        case 'isnan':
            if(arguments==null)arguments = ['']
            if(arguments.length!=1) throw new Error('Invalid value. Expecting one argument, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_numcheck">'+
                        '<field name="CONDITIONS">ISNAN</field>'+
                        '<value name="VAR">'+arguments[0]+'</value>'+
                    '</block>'
        case 'isgreater':
            if(arguments==null)arguments = ['','']
            if(arguments.length!=2) throw new Error('Invalid value. Expecting 2 arguments, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_numcompare">'+
                        '<field name="CONDITIONS">ISGREATER</field>'+
                        '<value name="VAR1">'+arguments[0]+'</value>'+
                        '<value name="VAR2">'+arguments[1]+'</value>'+
                    '</block>'
        case 'isless':
            if(arguments==null)arguments = ['','']
            if(arguments.length!=2) throw new Error('Invalid value. Expecting 2 arguments, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_numcompare">'+
                        '<field name="CONDITIONS">ISLESS</field>'+
                        '<value name="VAR1">'+arguments[0]+'</value>'+
                        '<value name="VAR2">'+arguments[1]+'</value>'+
                    '</block>'
        case 'isgreaterequal':
            if(arguments==null)arguments = ['','']
            if(arguments.length!=2) throw new Error('Invalid value. Expecting 2 arguments, '+
                                    'we found = ',arguments,'. postfix_expression.functionAndLibraries')
            return '<block type="library_math_numcompare">'+
                        '<field name="CONDITIONS">ISGREQ</field>'+
                        '<value name="VAR1">'+arguments[0]+'</value>'+
                        '<value name="VAR2">'+arguments[1]+'</value>'+
                    '</block>'
```

Figure 4.56: Utility function "createFunctionsAndLibraries" (part 6).

Figure 4.57: Utility function "createFunctionsAndLibraries" (part 7).

In the createFunctionsAndLibraries function, a comprehensive switch statement is implemented, handling 37 distinct cases. This function operates by evaluating the function name it receives, determining if it matches one of the 36 predefined utility function names from various C libraries or if it should be classified as a custom function. The second parameter of the function is 'arguments', which are pre-converted into the corresponding XML format through earlier encapsulation stages. When multiple arguments are present, they are demarcated with the tag '<argument_expression_list_comma>'.

So with this technique, all the arguments can be extracted into a list of arguments by using `arguments=String(arguments).split('<argument_expression_list_comma>')`. The switch statement then tailors the XML block accordingly for each case, dynamically integrating the argument list for library functions or appending the function name for custom functions, resulting in a precise XML representation for each function call.

Overall, with the above explanations they are covered some of the main aspects of the comprehensive grammar developed for transforming C code into Blockly's XML format. With the completion of the .jison file, which encapsulates this grammar, the next step involves its conversion into JavaScript. This grammar file was named "c_parser.jison". This transformation is achieved using the Jison command jison c_parser.jison, executed in a terminal window—a process detailed in Jison's documentation. This command generates a JavaScript file named c_parser.js. Subsequently, this file is integrated into the web application's JavaScript code. The parser, as crafted by Jison, materializes as a substantial function named c_parser, spanning an extensive 1,939 lines, encompassing the intricate logic required for the conversion process.

Inside the web application an "Import" button was added into the "index.html" file which is the main screen. This button alogn with its action listener function can be seen in figure 4.58.

```
<input
    type="file"
    id="files"
    style="display:none"
    onchange="getContextOfFile(files)"
    accept=".c"/>
<button
    type="button"
    class="btn btn-default navbar-btn"
    onclick="selectFile(files);">
        <span class="glyphicon glyphicon-cloud-upload"></span> Import
</button>
```

```
function selectFile(importButton){
    sessionStorage.clear();
    importButton.click();
}


function getContextOfFile(files){
  if(files && files.length === 1){
    var reader = new FileReader();
    reader.onload = function(event) {
      try{
        transpileCode(event.target.result);
      }catch(error){
        window.alert(error.message);
      }
    };
    reader.readAsText(files[0]);
  }
}
```

Figure 4.58: HTML and JavaScript code for Import button of the web application.

The HTML portion includes an <input> element of type file and a <button>. The input element, with the ID files, is styled to be hidden (display:none) and is configured to accept only files with a .c extension (ac-

cept=".c"). The button is designed for user interaction, enabling file import with an accompanying icon. The "onclick" event on the button triggers the selectFile(files) JavaScript function. This function, when executed, first clears the sessionStorage to ensure no residual data from previous operations exists. It then programmatically triggers a click event on the hidden file input element (importButton.click()). This action prompts the user to select a file from their system. The "onchange" event of the file input element calls the getContextOfFile(files) function. This function checks if a file has been selected (if(files && files.length === 1)). If a file is present, it uses the FileReader API to read its content. Inside getContextOfFile, a FileReader object is created. Its onload event is set up to handle the file reading process. Once the file is successfully read, the result is passed to the "transpileCode" function. This function receives a string input with the code that was read from the .c file, it transpiles it and populates the main workspace of Blockly with the corresponding blocks. If any errors occur during the file reading or transpilation process, an alert is displayed to the user with the error message (window.alert(error.message)). This error handling method catches all the "throw errors" that was previously placed within the embedded codes of the parser. So in case that the user tries to insert invalid or unsupported code, then he gets notified with the corresponding message indicating the exact problem.

The "transpileCode" function plays a pivotal role in the web application, particularly in transforming the imported C code into Blockly's visual block format. As it is already mentioned this function is designed to accept a single string argument, which represents the C code to be processed. The implementation of this function is illustrated in figures 4.60, 4.61, 4.62, 4.63, 4.64 and 4.65. The process begins with the removal of 'Include' libraries from the C code, as Blockly was modified to automatically integrate these libraries without user intervention. This Blockly's functionality was discussed in the previous chapter. The removal of 'Include' libraries is achieved through the 'removeIncludes' utility function ('code = removeIncludes(code);'). This utility function can be seen in figure 4.59.

The core transformation is initiated by invoking the 'c_parser' function, which parses the C code and converts it into an XML format compatible with Blockly. This is executed with the command **let xmlString = c_parser.parse.apply(c_parser, [code]);**. The resulting XML string, however, requires further modification before it can be effectively translated

into Blockly blocks.

During the parsing C code for Blockly conversion, a recurring scenario involves encountering common commands like "x = 4", "myFunc(y, x, z)", and "y = x". A commonality among these commands is the presence of variable setters or getters. However, a challenge arises in identifying the variable types in these instances. Unlike variable declarations where the type is explicit, setters and getters do not inherently reveal the variable type.

Recognizing each variable's specific type is essential in Blockly's framework, as it utilizes unique blocks to differentiate between simple variables, pointers, and arrays. This distinction is also reflected in their corresponding XML representations. However, during parsing, distinguishing between these types for setters and getters poses a challenge, as the parser lacks information about the variable type. The adopted strategy treats all variables in setters and getters as "simple variables" during the initial parsing phase, converting them into their corresponding simple variable XML blocks. Consequently, these XML blocks need to be modified later to accurately reflect the actual types of the variables.

To adjust accordingly the incorrect XML representation of code, a two-step approach is implemented. The first 'for' loop traverses the code, identifying all variable declarations—both within and outside of functions, including function arguments. These variables are then stored within an object. In the object, we use unique identifiers to organize the declared variables based on their location in the code. For instance, variables declared within the main function are stored under 'functionObjects['main']'. This approach helps differentiate between variables based on where they are declared. For instance, a variable named 'myVar1' could be a pointer declared in the main function, while a different variable with the same name 'myVar1' could be an array declared within a custom function.

The second 'for' loop systematically searches the entire code for setters and getters. Utilizing the previously compiled object of declared variables, the loop adjusts the XML representations of these setters and getters to reflect their actual types. This meticulous process ensures that each variable is accurately represented in the Blockly environment, corresponding to its true nature as a simple variable, pointer, or array, as initially declared in the C code. Through this method, the conversion maintains the integrity and functionality of the original C code in its new visual Blockly format.

To perform such adjustments in the XML code the function "transpileCode" employs the DOMParser and XMLSerializer utilities. Initially, DOM-

Parser is used to convert the XML string into a DOM (Document Object Model) element. This conversion allows for iteration and manipulation of the XML structure through various loops. Once the processing is complete and the XML structure has been adjusted to meet Blockly's requirements, XMLSerializer is employed to reconvert the DOM element back into a string format. This string format is essential for integrating the XML with Blockly and rendering the corresponding blocks.

Inside those loops the code processes each XML block, examining its type and modifying its structure as needed. This includes handling different types of blocks such as 'main_block' and custom function declarations ('procedures_defreturn' and 'procedures_defnoreturn'). The function meticulously stores variable declarations and mutations, and adapts variable 'set' and 'get' operations to align with Blockly's format. To achieve the above adjustments several utility functions were constructed and can be illustrated in figures 4.66, 4.67 and 4.68.

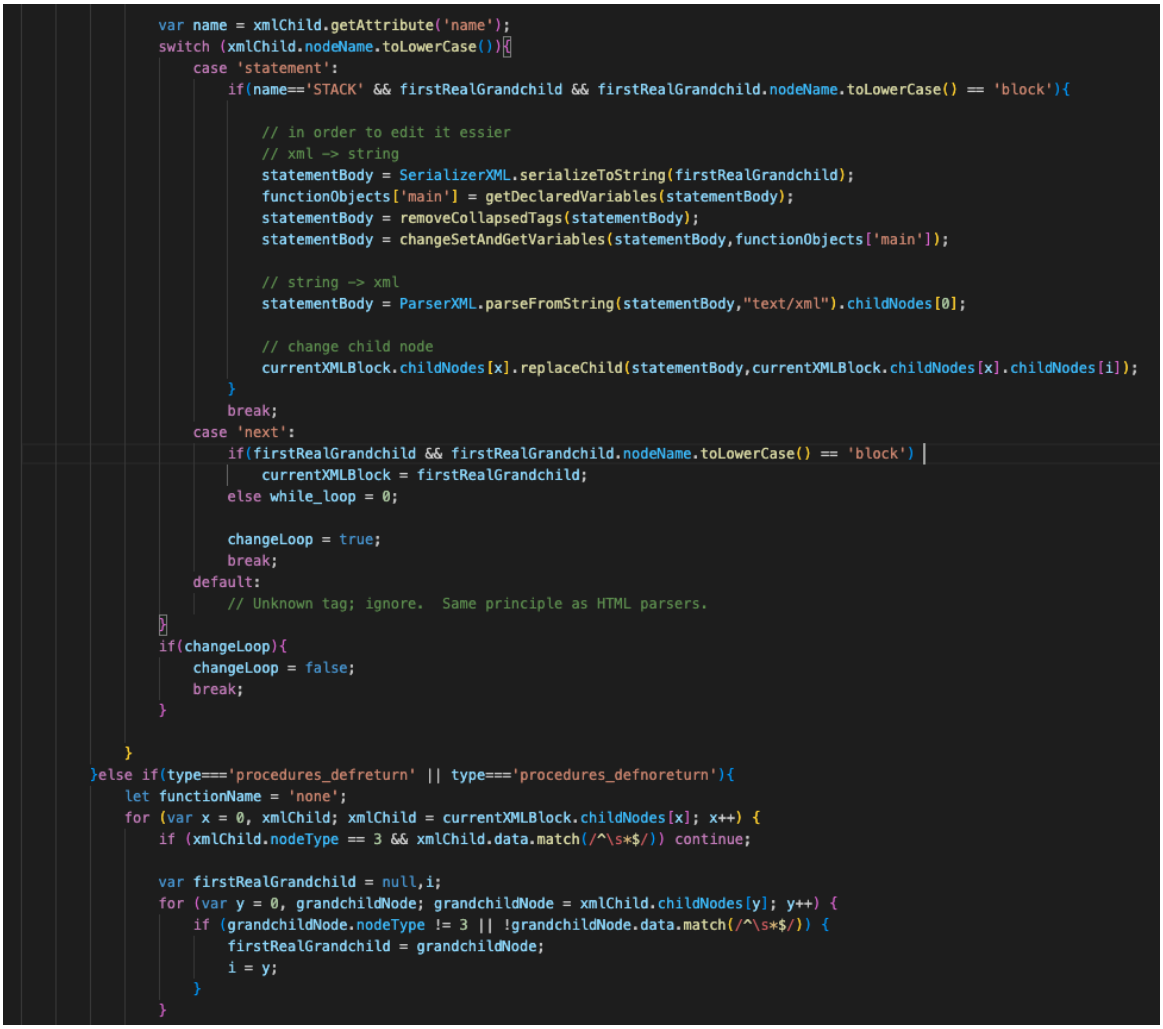
Once all modifications are complete, the XML string is encapsulated within a '<xml>' tag and the Blockly workspace is cleared to prepare for the new block configuration ("Blockly.mainWorkspace.clear();"). The final step involves using Blockly's 'domToWorkspace' method to convert the XML string into interactive Blockly blocks within the workspace, thus completing the transformation from text-based C code to a visual programming environment ("Blockly.Xml.domToWorkspace(Blockly.mainWorkspace,xmlDoc);").

```
function removeIncludes(code){
    const includePattern = /#include\s*<.*?>/g;
    const codeWithoutIncludes = code.replace(includePattern, '');
    return codeWithoutIncludes;
}
```

Figure 4.59: Utility function "removeIncludes".

```
function transpileCode(code){
    code = removeIncludes(code);
    let xmlString = c_parser.parse.apply(c_parser, [code]);


    const SerializerXML = new XMLSerializer();
    const ParserXML = new DOMParser();
    let currentXMLBlock = ParserXML.parseFromString(xmlString,"text/xml").childNodes[0];
    let totalXML = currentXMLBlock;

    let functionMutationArguments = {};
    let functionObjects = {};
    let arguments,statementBody;
    let while_loop = 1;

    // Run for all external blocks
    // store all declared variables (both mutations and ordinary)
    // mutation arguments -> functionMutationArguments
    // ordinary variables & mutation arguments -> functionObjects
    // change set & get variables if it is necessary
    while(while_loop){
        let type = currentXMLBlock.getAttribute('type');
        let changeLoop = false;
        if(type==='main_block'){
            for (var x = 0, xmlChild; xmlChild = currentXMLBlock.childNodes[x]; x++) {
                if (xmlChild.nodeType == 3 && xmlChild.data.match(/^\s*$/)) continue;

                var firstRealGrandchild = null,i;
                for (var y = 0, grandchildNode; grandchildNode = xmlChild.childNodes[y]; y++) {
                    if (grandchildNode.nodeType != 3 || !grandchildNode.data.match(/^\s*$/)) {
                        firstRealGrandchild = grandchildNode;
                        i = y;
                    }
                }
```

Figure 4.60: Code implementation for "transpileCode" (part 1).



```
            var name = xmlChild.getAttribute('name');
            switch (xmlChild.nodeName.toLowerCase()){
                case 'statement':
                    if(name=='STACK' && firstRealGrandchild && firstRealGrandchild.nodeName.toLowerCase() == 'block'){

                        // in order to edit it essier
                        // xml -> string
                        statementBody = SerializerXML.serializeToString(firstRealGrandchild);
                        functionObjects['main'] = getDeclaredVariables(statementBody);
                        statementBody = removeCollapsedTags(statementBody);
                        statementBody = changeSetAndGetVariables(statementBody,functionObjects['main']);

                        // string -> xml
                        statementBody = ParserXML.parseFromString(statementBody,"text/xml").childNodes[0];

                        // change child node
                        currentXMLBlock.childNodes[x].replaceChild(statementBody,currentXMLBlock.childNodes[x].childNodes[i]);
                    }
                    break;
                case 'next':
                    if(firstRealGrandchild && firstRealGrandchild.nodeName.toLowerCase() == 'block')
                        currentXMLBlock = firstRealGrandchild;
                    else while_loop = 0;

                    changeLoop = true;
                    break;
                default:
                    // Unknown tag; ignore.  Same principle as HTML parsers.
            }
            if(changeLoop){
                changeLoop = false;
                break;
            }

        }
    }else if(type==='procedures_defreturn' || type==='procedures_defnoreturn'){
        let functionName = 'none';
        for (var x = 0, xmlChild; xmlChild = currentXMLBlock.childNodes[x]; x++) {
            if (xmlChild.nodeType == 3 && xmlChild.data.match(/^\s*$/)) continue;

            var firstRealGrandchild = null,i;
            for (var y = 0, grandchildNode; grandchildNode = xmlChild.childNodes[y]; y++) {
                if (grandchildNode.nodeType != 3 || !grandchildNode.data.match(/^\s*$/)) {
                    firstRealGrandchild = grandchildNode;
                    i = y;
                }
            }
```

Figure 4.61: Code implementation for "transpileCode" (part 2).

```
var name = xmlChild.getAttribute('name');
switch (xmlChild.nodeName.toLowerCase()){
    case 'mutation':
        arguments = SerializerXML.serializeToString(xmlChild);
        arguments = arguments.replace('<mutation>','');
        arguments = arguments.replace('</mutation>','');
        break;
    case 'field':
        if(name=='NAME' && xmlChild?.childNodes[0].data) {
            functionName = xmlChild.childNodes[0].data;
            if(arguments) {
                // add arguments as they are on functionMutationArguments
                functionMutationArguments[functionName] = arguments;
                // change the form of the argumnets and add them on functionObjects
                let functionVariables = {};
                arguments = arguments.split('<arg');
                if(arguments.length>1){
                    // run for all arguments-variables of this functions
                    for(let i=1;i<arguments.length;i++){
                        let varName,curVar = {};
                        varName = arguments[i].slice(arguments[i].indexOf('name="')+6);
                        varName = varName.slice(0,varName.indexOf('"'));
                        arguments[i] = arguments[i].slice(arguments[i].indexOf('"'));

                        curVar['type'] = arguments[i].slice(arguments[i].indexOf('types="')+7);
                        curVar['type'] = curVar['type'].slice(0,curVar['type'].indexOf('"'));
                        arguments[i] = arguments[i].slice(arguments[i].indexOf('"'));
                        if(arguments[i].includes('dist="p"')){
                            arguments[i] = arguments[i].slice(arguments[i].indexOf('"'));
                            curVar['pointer'] = arguments[i].slice(arguments[i].indexOf('iteration="')+11);
                            curVar['pointer'] = curVar['pointer'].slice(0,curVar['pointer'].indexOf('"'));
                        }else if(arguments[i].includes('dist="a"')){
                            arguments[i] = arguments[i].slice(arguments[i].indexOf('"'));
                            curVar['arr_1'] = arguments[i].slice(arguments[i].indexOf('length_1="')+10);
                            curVar['arr_1'] = curVar['arr_1'].slice(0,curVar['arr_1'].indexOf('"'));
                            if(arguments[i].includes('length_2="')){
                                arguments[i] = arguments[i].slice(arguments[i].indexOf('"'));
                                curVar['arr_2'] = arguments[i].slice(arguments[i].indexOf('length_2="')+10);
                                curVar['arr_2'] = curVar['arr_2'].slice(0,curVar['arr_2'].indexOf('"'));
                                if(arguments[i].includes('length_3="')){
                                    arguments[i] = arguments[i].slice(arguments[i].indexOf('"'));
                                    curVar['arr_3'] = arguments[i].slice(arguments[i].indexOf('length_3="')+10);
                                    curVar['arr_3'] = curVar['arr_3'].slice(0,curVar['arr_3'].indexOf('"'));
                                }
                            }
                        }
                        functionVariables[varName] = curVar;
                    }
                    functionObjects[functionName] = functionVariables;
                }
            }
        }
```

Figure 4.62: Code implementation for "transpileCode" (part 3).

```
                    break;
                case 'statement':
                    if(name=='STACK' && firstRealGrandchild && firstRealGrandchild.nodeName.toLowerCase() == 'block'){
                        statementBody = SerializerXML.serializeToString(firstRealGrandchild);
                        if(functionObjects[functionName]){
                            // variables already exist on this function (from arguments)
                            let temp1=functionObjects[functionName], temp2=getDeclaredVariables(statementBody);
                            Object.keys(temp2).forEach(key => {temp1[key] = temp2[key];});
                            functionObjects[functionName] = temp1;
                        }
                        else functionObjects[functionName] = getDeclaredVariables(statementBody);

                        statementBody = removeCollapsedTags(statementBody);
                        statementBody = changeSetAndGetVariables(statementBody,functionObjects[functionName]);
                        statementBody = ParserXML.parseFromString(statementBody,"text/xml").childNodes[0];
                        currentXMLBlock.childNodes[x].replaceChild(statementBody,currentXMLBlock.childNodes[x].childNodes[i]);
                    }
                    break;
                case 'next':
                    if(firstRealGrandchild && firstRealGrandchild.nodeName.toLowerCase() == 'block') currentXMLBlock = firstRealGrandchild;
                    else while_loop = 0;

                    changeLoop = true;
                    break;
                default:
                    // Unknown tag; ignore.  Same principle as HTML parsers.
            }
            if(changeLoop){
                changeLoop = false;
                break;
            }
        }
    }else{
        // skip this block without doing anything
        for (var x = 0, xmlChild; xmlChild = currentXMLBlock.childNodes[x]; x++) {
            if (xmlChild.nodeType == 3 && xmlChild.data.match(/^\s*$/)) continue;

            var firstRealGrandchild = null,i;
            for (var y = 0, grandchildNode; grandchildNode = xmlChild.childNodes[y]; y++) {
                if (grandchildNode.nodeType != 3 || !grandchildNode.data.match(/^\s*$/)) {
                    firstRealGrandchild = grandchildNode;
                    i = y;
                }
            }

            var name = xmlChild.getAttribute('name');
            switch (xmlChild.nodeName.toLowerCase()){
                case 'next':
                    if(firstRealGrandchild && firstRealGrandchild.nodeName.toLowerCase() == 'block') currentXMLBlock = firstRealGrandchild;
                    else while_loop = 0;
```

Figure 4.63: Code implementation for "transpileCode" (part 4).

```
                        changeLoop = true;
                        break;
                    default:
                        // Unknown tag; ignore.  Same principle as HTML parsers.
                }
                if(changeLoop){
                    changeLoop = false;
                    break;
                }
            }
        }
    }
    xmlString = SerializerXML.serializeToString(totalXML);
    xmlString = removeCollapsedTags(xmlString);
    // --------------------------------------------------------


    // Add mutation argumnets on function calls
    let functions = xmlString.split('procedures_callreturn');
    if(functions.length>1){
        for(let i=1;i<functions.length;i++){
            functionName = functions[i].slice(functions[i].indexOf('<mutation name="'))
            functionName = functionName.slice(0,functionName.indexOf('">'))
            functionName = functionName.replace('<mutation name="','')
            if(functionMutationArguments[functionName]){
                let index = xmlString.indexOf('<mutation name="'+functionName+'">');
                let preString = xmlString.slice(0,index);
                let metaString = xmlString.slice(index);
                metaString = metaString.slice(metaString.indexOf('</mutation>'));
                preString = preString + '<mutation name="'+functionName+'">' + functionMutationArguments[functionName];
                xmlString = preString.concat(metaString);
            }
        }
    }
    functions = xmlString.split('procedures_callnoreturn')
    if(functions.length>1){
        for(let i=1;i<functions.length;i++){
            functionName = functions[i].slice(functions[i].indexOf('<mutation name="'))
            functionName = functionName.slice(0,functionName.indexOf('">'))
            functionName = functionName.replace('<mutation name="','')
            if(functionMutationArguments[functionName]){
                let index = xmlString.indexOf('<mutation name="'+functionName+'">');
                let preString = xmlString.slice(0,index);
                let metaString = xmlString.slice(index);
                metaString = metaString.slice(metaString.indexOf('</mutation>'));
                preString = preString + '<mutation name="'+functionName+'">' + functionMutationArguments[functionName];
                xmlString = preString.concat(metaString);
            }
        }
    }
    // --------------------------------------------------------
```

Figure 4.64: Code implementation for "transpileCode" (part 5).

```
    xmlString = '<xml style="display: none">'+xmlString+'</xml>';
    Blockly.mainWorkspace.clear();
    var xmlDoc = ParserXML.parseFromString(xmlString,"text/xml").childNodes[0];
    Blockly.Xml.domToWorkspace(Blockly.mainWorkspace,xmlDoc);
}
```

Figure 4.65: Code implementation for "transpileCode" (part 6).

```
function getDeclaredVariables(statementString){
    statementString = statementString.replaceAll('<field name="LENGTH_2"/>','<field name="LENGTH_2"></field>');
    statementString = statementString.replaceAll('<field name="LENGTH_3"/>','<field name="LENGTH_3"></field>');
    let desiredBlocks = ['variables_declare','variables_pointer_declare','variables_array_declare'];
    let tempString,functionVariables = {};
    for (const element of desiredBlocks) {
        tempString = statementString;
        tempString = tempString.split(element);
        if(tempString.length>1){
            for(let i=1;i<tempString.length;i++){
                let variable = {},variableName;
                variable['type'] = tempString[i].slice(tempString[i].indexOf('<field name="TYPES">')+20);
                variable['type'] = variable['type'].slice(0,variable['type'].indexOf('</field>'));
                tempString[i] = tempString[i].slice(tempString[i].indexOf('</field>')+8);

                variableName = tempString[i].slice(tempString[i].indexOf('<field name="VAR">')+18);
                variableName = variableName.slice(0,variableName.indexOf('</field>'));
                tempString[i] = tempString[i].slice(tempString[i].indexOf('</field>')+8);

                if(element=='variables_pointer_declare'){
                    variable['pointer'] = tempString[i].slice(tempString[i].indexOf('<field name="ITERATION">')+24);
                    variable['pointer'] = variable['pointer'].slice(0,variable['pointer'].indexOf('</field>'));
                }else if(element=='variables_array_declare'){
                    variable['arr_1'] = tempString[i].slice(tempString[i].indexOf('<field name="LENGTH_1">')+23);
                    variable['arr_1'] = variable['arr_1'].slice(0,variable['arr_1'].indexOf('</field>'));
                    tempString[i] = tempString[i].slice(tempString[i].indexOf('</field>')+8);

                    variable['arr_2'] = tempString[i].slice(tempString[i].indexOf('<field name="LENGTH_2">')+23);
                    variable['arr_2'] = variable['arr_2'].slice(0,variable['arr_2'].indexOf('</field>'));
                    tempString[i] = tempString[i].slice(tempString[i].indexOf('</field>')+8);
                    if(!variable['arr_2']) delete variable['arr_2'];

                    variable['arr_3'] = tempString[i].slice(tempString[i].indexOf('<field name="LENGTH_3">')+23);
                    variable['arr_3'] = variable['arr_3'].slice(0,variable['arr_3'].indexOf('</field>'));
                    if(!variable['arr_3']) delete variable['arr_3'];
                }

                functionVariables[variableName] = variable;
            }
        }
    }
    return functionVariables;
}
```

Figure 4.66: Utility function "getDeclaredVariables".

```
function changeSetAndGetVariables(statementString,variables){
    let tempString = statementString.split('variables_get');
    if(tempString.length>1){
        for(let i=1;i<tempString.length;i++){
            let curVarName = tempString[i].slice(tempString[i].indexOf('<field name="VAR">')+18);
            curVarName = curVarName.slice(0,curVarName.indexOf('</field>'));
            let curVar = variables[curVarName];
            if(curVar && (curVar.pointer || curVar.arr_1)){
                if(curVar.pointer){
                    tempString[i] = 'variables_pointer_get' + tempString[i];
                }else{
                    tempString[i] = 'variables_array_get' + tempString[i];
                    tempString[i] = tempString[i].replace('<field name="VAR">'+curVarName+'</field>',
                        '<field name="VAR">'+curVarName+'</field><field name="LENGTH_1"></field><field name="LENGTH_2"></field><field name="LENGTH_3"></field>');
                }
            }else{
                tempString[i] = 'variables_get' + tempString[i];
            }
        }
        statementString = tempString.join('');
    }
    tempString = statementString.split('variables_set');
    if(tempString.length>1){
        for(let i=1;i<tempString.length;i++){
            let curVarName = tempString[i].slice(tempString[i].indexOf('<field name="VAR">')+18);
            curVarName = curVarName.slice(0,curVarName.indexOf('</field>'));
            let curVar = variables[curVarName];
            if(curVar && (curVar.pointer || curVar.arr_1)){
                if(curVar.pointer){
                    tempString[i] = 'variables_pointer_set' + tempString[i];
                    tempString[i] = tempString[i].replace('<field name="VAR">'+curVarName+'</field>',
                        '<value name="VAR"><block type="variables_pointer_get"><field name="VAR">'+curVarName+'</field></block></value>');
                }else{
                    tempString[i] = 'variables_array_set' + tempString[i];
                    tempString[i] = tempString[i].replace('<field name="VAR">'+curVarName+'</field>',
                        '<field name="VAR">'+curVarName+'</field><field name="LENGTH_1"></field><field name="LENGTH_2"></field><field name="LENGTH_3"></field>');
                }
            }else{
                tempString[i] = 'variables_set' + tempString[i];
            }
        }
        statementString = tempString.join('');
    }
    return statementString;
}
```

Figure 4.67: Utility function "changeSetAndGetVariables".

```
function removeCollapsedTags(statementString){
    statementString = statementString.replace('xmlns="http://www.w3.org/1999/xhtml"','');
    let temporaryString = statementString.split('/>');
    let tagName;
    if(temporaryString.length>1){
        for(let i=0;i<temporaryString.length-1;i++){
            tagName = temporaryString[i].slice(temporaryString[i].lastIndexOf('<')+1);
            if(tagName.includes(' ')){
                tagName = tagName.split(' ')[0];
                temporaryString[i] = temporaryString[i] + '></'+tagName+'>';
            }else{
                temporaryString[i] = temporaryString[i] + '></'+tagName+'>';
            }
        }
    }
    return temporaryString.join('');
}
```

```
function stringToXML(string){
    const ParserXML = new DOMParser();
    let xml = ParserXML.parseFromString(string,"text/xml").childNodes[0];
    return xml;
}

function xmlToString(xml){
    const SerializerXML = new XMLSerializer();
    let string = SerializerXML.serializeToString(xml);
    string = removeCollapsedTags(string);
    return string;
}
```

Figure 4.68: Utility functions "removeCollapsedTags", "stringToXML" and "xmlToString".

As this chapter comes to a close, it's pertinent to reflect on the comprehensive process undertaken to translate C code into its Blockly visual block equivalent. This intricate endeavor involved meticulously parsing C code, managing various data types and structures, and ensuring accurate correspondence with Blockly's visual blocks. The implementation demanded a deep understanding of both C syntax and Blockly's framework, culminating in a sophisticated system that enables intuitive interaction with C programming concepts. This significant effort not only augments the learning experience for programming novices but also serves as a bridge, connecting the abstract world of coding with its tangible, visual representation. This

advancement in educational technology holds the promise of making programming more accessible and engaging for learners.

4.4 Code Execution and Memory Visualization

This chapter delves into the intricate processes of code execution and memory visualization within the context of visual programming. The core objective is to analyse the implementation of how code, once created into Blockly's visual blocks, is executed and how the underlying memory management and operations are visually represented. This representation is not only pivotal for understanding the execution flow but also crucial for grasping complex concepts such as memory allocation, pointer operations, and data structure manipulation.

When the user clicks the 'Run' button, its associated action listener triggers the execution of the **runCode()** function. Its primary function is to facilitate the transition from the code editing environment to the visualization interface. The function initially checks if the browser supports session storage, an essential feature for temporarily storing code and XML data across different pages of the web application. This check ensures compatibility and smooth functioning across various browsers and user environments. Then it retrieves the current state of the Blockly workspace, which includes all the visual blocks that represent the C code. This workspace is converted into an XML DOM structure using **Blockly.Xml.workspaceToDom(Blockly.mainWorkspace)**, and then into a string representation through the **xmlToString()** function. Concurrently, the function also extracts the textual representation of the code from the **codePanel** element. This extraction involves stripping HTML tags and converting HTML entities back to their character equivalents, ensuring the code is in its raw and executable form. The extracted code is then URL-encoded using **encodeURIComponent()**. This encoding is crucial for preserving the integrity of the code when passing it through the web environment, where certain characters might otherwise be misinterpreted. Both the encoded code and the stringified XML are then stored in the session storage using **sessionStorage.setItem()**. This storage step is key to transferring the state of the user's work to the visualization page without losing any data or context. Finally, the function redirects the user to the visualization page **(visualize.html)** using **window.open()**. This redirection is done in the same tab **("_self")**, ensuring a seamless transition from

the coding environment to the visualization interface. On this new page, the stored code and XML data are retrieved from the session storage, ready to be executed and visualized, providing the user with an interactive and educational insight into how their code translates into actions and memory utilization. This function is illustrated in figure 4.69.

```
function runCode() {
    if(sessionStorage){
        var code, xmlSes;
        xmlSes = Blockly.Xml.workspaceToDom(Blockly.mainWorkspace);
        xmlSes = xmlToString(xmlSes);
        code = codePanel.innerHTML;
        code = code.replace(/<\/?[^>]+(>|$)/g, "");
        code = code.replaceAll('&lt;','<');
        code = code.replaceAll('&gt;','>');
        code = encodeURIComponent(code);
        sessionStorage.setItem("code", code);
        sessionStorage.setItem("xml", xmlSes);
        window.open("visualize.html","_self");
    }
}
```

Figure 4.69: Implementation of function "runCode".

The **visualize.html** page within the web application is ingeniously designed to integrate the Python Tutor visualization tool, specifically for executing and visualizing C code. This integration is achieved through the use of an <iframe> element, which essentially acts as a window or portal within the web app, embedding content from an external source – in this case, Python Tutor. This iframe tag can be seen in figure 4.70. The <**iframe**> element, identified by **id="myFrame"**, is configured to take up the entire width (**width="100%"**) and a specific height (**height="800"**) of its container on the **visualize.html** page. This sizing ensures that the visualization tool occupies a substantial portion of the page, providing a clear and extensive view of the code execution process. The source for the <iframe> is not statically defined in the HTML. Instead, it is dynamically set based on the C code that the user intends to run and visualize. This dynamic setting is crucial for tailoring the visualization to the specific code snippet that the user has input.

```
<iframe id="myFrame" src="" width="100%" height="800" style="border:none;">
</iframe>
```

Figure 4.70: The "iframe" tag within visualize.html page.

The core functionality of the visualization process is encapsulated within a <**script**> tag at the end of the page's body. This script is illustrated in figure 4.71 This script begins by checking if the **sessionStorage** object exists and contains the key "code", which it was embedded from the previous page. On confirming the presence of code, the script retrieves this stored code and proceeds to construct a URL string for the Python Tutor service. This URL is not just a link to Python Tutor's homepage; it's a specialized address that includes several parameters. The most critical part of the URL is the inclusion of the user's C code, URL-encoded to ensure it's transmitted over the internet correctly and interpreted accurately by Python Tutor. The script retrieves the encoded C code from sessionStorage using **sessionStorage.getItem("code")**. The code has been pre-converted into a URL-encoded format compatible with Python Tutor on the previous page, prior to being securely stored within sessionStorage. Except from the code, the generated URL contains several other configuration parameters. These parameters dictate how Python Tutor should display the code. They control aspects like the visualization mode, the treatment of primitive values in memory, and the specific C compiler settings to emulate (such as cumulative, heapPrimitives, mode, origin, py, rawInputLstJSON, and textReferences).

After crafting the URL, the script sets this URL as the **src** attribute of the <**iframe**>. This action essentially tells the <iframe> to load the content from the Python Tutor URL, which now contains the user's specific code and the desired visualization settings. Once the <iframe> loads the URL, Python Tutor takes over. It interprets the URL-encoded C code and displays its execution within the <iframe>. This includes visual representations of the code's execution flow, the state of the stack, and the heap memory allocations. Users can step through their code line by line, gaining insights into how their C code operates at runtime. If a user accesses visualize.html directly without running any code (hence, sessionStorage would not have the "code" item), the script triggers a modal alert. This alert prevents users from reaching the visualization page without valid context, ensuring that the

tool is used as intended within the workflow of the web application. The previously mentioned actions, including URL encoding, storage operations and modal alert appearance, are executed using functions from the jQuery library, a widely-used JavaScript framework.

```
  </body>
  <script>
    if(sessionStorage && sessionStorage.getItem("code")){
        var code = sessionStorage.getItem("code");
        var url = 'https://pythontutor.com/visualize.html#code='+code+'&'+
                        'cumulative=false&curInstr=0&heapPrimitives=nevernest&'+
                        'mode=display&origin=opt-frontend.js&py=c_gcc9.3.0&'+
                        'rawInputLstJSON=%5B%5D&textReferences=false';
        const el = document.querySelector("#myFrame");
        el.setAttribute("src", url);
    }else{
        var mymodal = $('#errorModal');
        mymodal.find('.alert-text').text('You don\'t have access on this page!');
        mymodal.modal('show');
    }
</script>
</html>
```

Figure 4.71: The "script" code within visualize.html page.

In summary, the visualize.html page, through its <iframe> setup and intelligent scripting, cleverly embeds the Python Tutor visualization tool within the web app. This integration allows users to execute their C code and interactively visualize its memory and execution dynamics, providing an invaluable learning experience in understanding C programming concepts.

Chapter 5

Evaluation

5.1 Introduction

This chapter outlines the evaluation process undertaken to validate the effectiveness and functionality of the Blockly-C web application, as described in this thesis. The evaluation was conducted with a group of students from the Technical University of Crete in Chania, who graciously participated in testing and providing feedback on the application. The primary objective was to demonstrate the practicality and user-friendliness of Blockly-C and assess its potential as an educational tool for learning the C programming language.

5.2 Evaluation Methodology and Data Collection

The evaluation process began with a comprehensive demonstration of Blockly-C web application to the participating students. This demonstration included a step-by-step walkthrough of creating a simple program in C using the application, highlighting its key features and capabilities. The demonstration aimed to familiarize the students with the various functionalities of Blockly-C, from code creation and editing to program execution and memory visualization.

This presentation was conducted in an online meeting with the students. During this session, the students were provided with a thorough explanation

of the programming exercise, encompassing all the steps involved in the solution. The primary objective of this meeting was to introduce the tool to the students and familiarize them with its functionality, rather than assessing their proficiency in using it.

The exercise assigned to the students is depicted in Appendix files .1. It involved constructing a program in C featuring two nested for loops. The objective of these loops was to output a right-angled triangle made of asterisks ("*") in the console. An illustrative example of the expected triangle shape is presented in Figure 5.1.

Figure 5.1: Right-angled triangle made of asterisks ("*") that should be printed in console.

The students received a URL to access the Blockly-C website, providing them with the opportunity to explore and engage with the platform at their own pace. After the introduction of the exercise, they had the flexibility to independently experiment and attempt to solve it using Blockly-C. Alternatively, they could participate in a screen sharing session, where a live demonstration of the tool was presented, along with a detailed solution to the exercise. During the presentation, students were actively encouraged to ask any questions they had regarding the tool, ensuring a thorough understanding and interactive learning experience.

Following the demonstration, the students were invited to engage with the application, where they had the opportunity to explore its features and create their own C programs using the block-based interface. This hands-on experience allowed them to gain a practical understanding of the application's utility in learning and practicing C programming.

To gather structured feedback, the students were asked to complete a questionnaire after their interaction with Blockly-C. The questionnaire com-

prised questions from the User Experience Questionnaire (UEQ) and additional queries tailored to this specific evaluation. The inclusion of UEQ questions aimed to measure the application's user experience in a standardized manner, while the custom questions sought to gather insights specific to the educational aspects and usability of Blockly-C in a learning environment.

5.3 Analysis of Questionnaire Results

The results of this evaluation process are critical in assessing the feasibility of Blockly-C as an educational tool. They provide valuable insights into the application's strengths, areas for improvement, and its overall impact on the learning experience of students in the field of computer programming. As questionnaire it was used the User Experience Questionnaire (UEQ).

The User Experience Questionnaire (UEQ) [13] is a widely recognized tool used in the evaluation of user experience for various products and services, especially in the field of software and web applications. Developed to provide a quick, reliable, and standardized method of assessing user experience, the UEQ has become an essential tool for researchers, designers, and developers in gauging the effectiveness and appeal of their products.

One of the primary strengths of the UEQ is its comprehensive coverage of different aspects of user experience. It encompasses a broad range of attributes, including attractiveness, perspicuity (clarity and ease of understanding), efficiency, dependability, stimulation (the extent to which the product is exciting and motivating), and novelty (the ability to catch interest and innovate). This wide spectrum allows for a holistic assessment of a product from a user's perspective.

The questionnaire is structured as a set of bipolar scales, each representing different attributes of user experience. For example, an item might be presented as a scale from "boring" to "exciting" to gauge stimulation. Participants are asked to rate their experience based on these scales, offering insights into how they perceive various facets of the product or service. This format is not only user-friendly but also enables quick completion, typically requiring only a few minutes.

Another significant advantage of the UEQ is the availability of benchmark data, which allows for comparative analysis. The UEQ database contains data from hundreds of studies, spanning a diverse range of products. This benchmarking capability enables evaluators to compare their product's user

experience scores against those of similar products, providing a context for understanding the results.

Furthermore, the UEQ is versatile and adaptable, suitable for use in different stages of product development. It can be employed in early prototypes as well as in finished products, making it a valuable tool for continuous improvement and iterative design processes. Its adaptability extends to various contexts and user groups, making it applicable to a wide range of scenarios and target audiences.

A total of 27 students participated in the questionnaire survey. The questionnaire completed by the participants is presented in Figures 5.2, 5.3 and 5.4.

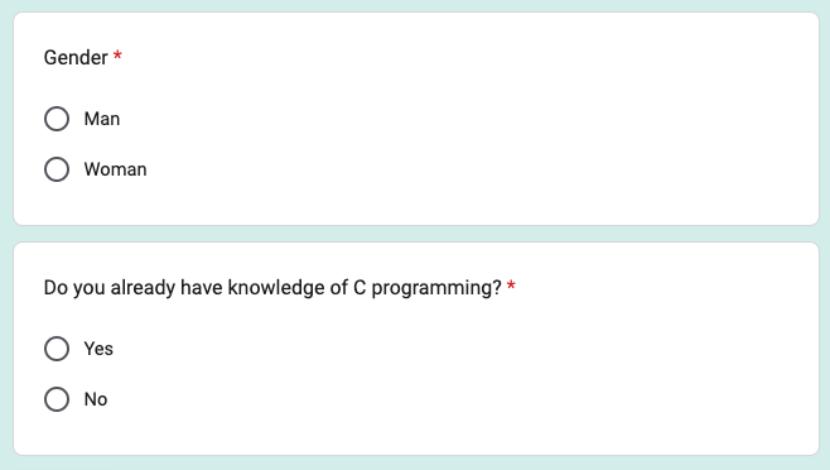


Figure 5.2: Evaluation questionnaire for Blockly-C (part 1).

	1	2	3	4	5	6	7		
annoying	○	○	○	○	○	○	○	enjoyable	1
not understandable	○	○	○	○	○	○	○	understandable	2
creative	○	○	○	○	○	○	○	dull	3
easy to learn	○	○	○	○	○	○	○	difficult to learn	4
valuable	○	○	○	○	○	○	○	inferior	5
boring	○	○	○	○	○	○	○	exciting	6
not interesting	○	○	○	○	○	○	○	interesting	7
unpredictable	○	○	○	○	○	○	○	predictable	8
fast	○	○	○	○	○	○	○	slow	9
inventive	○	○	○	○	○	○	○	conventional	10
obstructive	○	○	○	○	○	○	○	supportive	11
good	○	○	○	○	○	○	○	bad	12
complicated	○	○	○	○	○	○	○	easy	13
unlikable	○	○	○	○	○	○	○	pleasing	14
usual	○	○	○	○	○	○	○	leading edge	15
unpleasant	○	○	○	○	○	○	○	pleasant	16
secure	○	○	○	○	○	○	○	not secure	17
motivating	○	○	○	○	○	○	○	demotivating	18
meets expectations	○	○	○	○	○	○	○	does not meet expectations	19
inefficient	○	○	○	○	○	○	○	efficient	20
clear	○	○	○	○	○	○	○	confusing	21
impractical	○	○	○	○	○	○	○	practical	22
organized	○	○	○	○	○	○	○	cluttered	23
attractive	○	○	○	○	○	○	○	unattractive	24
friendly	○	○	○	○	○	○	○	unfriendly	25
conservative	○	○	○	○	○	○	○	innovative	26

Figure 5.3: Evaluation questionnaire for Blockly-C (part 2).

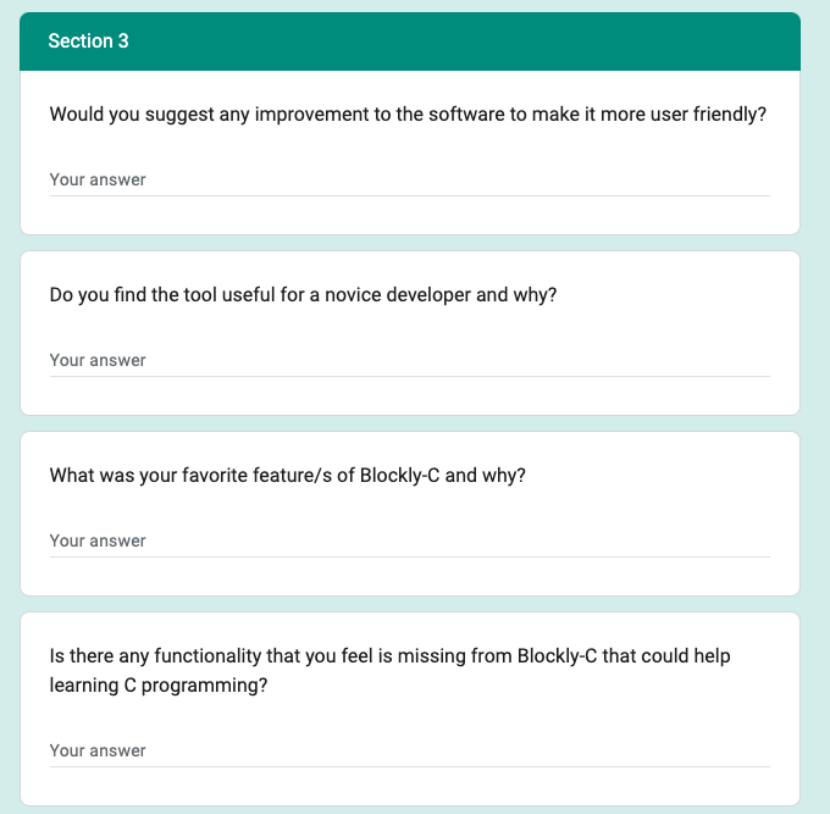


Figure 5.4: Evaluation questionnaire for Blockly-C (part 3).

The analysis was performed in the Amazon QuickSight visualization tool, based on the data taken from the UEQ. The numbers 1-7 in the data, show the score range for each item of evaluation (bad/good, unattractive/attractive, unfriendly/friendly etc). The items of evaluation are grouped in different attributes that are mentioned as scale in the analysis.

Of the 27 participants, 16 were men and 11 were women, representing a gender distribution of 60% and 40%, respectively, as illustrated in Figure 5.5."

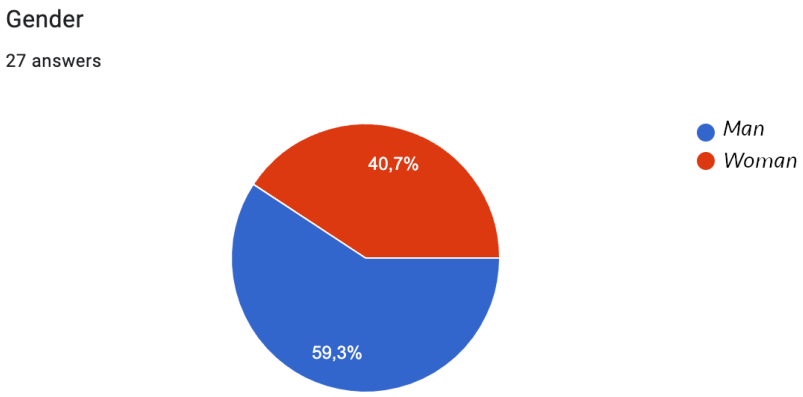


Figure 5.5: Gender Statistics.

Out of the participants, 14 reported having no knowledge of the C programming language, while 13 indicated familiarity with it, as detailed in Figure 5.6.

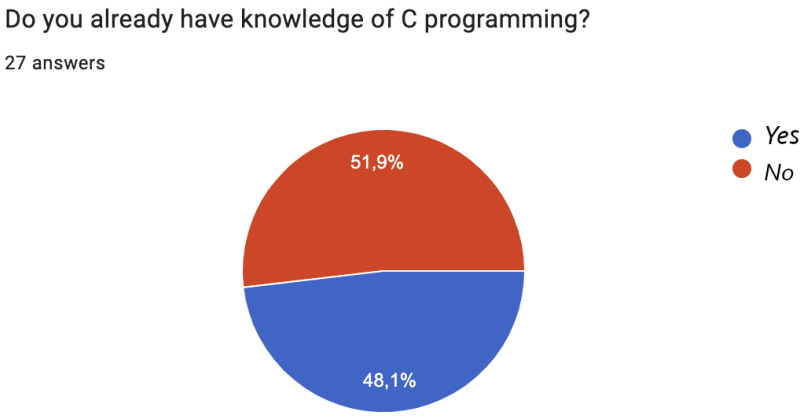


Figure 5.6: Knowledgeable of C programming language statistics.

The two vertical bar charts in Figures 5.7 and 5.8 show the amount of each score for every evaluation item and for the different attributes. The colours in these charts represent the range of the scores, where green is the most positive score and red the most negative. As it appears in the graphs the positive scores exceed the negatives. One can easily identify the items with the most positive feedback but also the areas that could be improved.

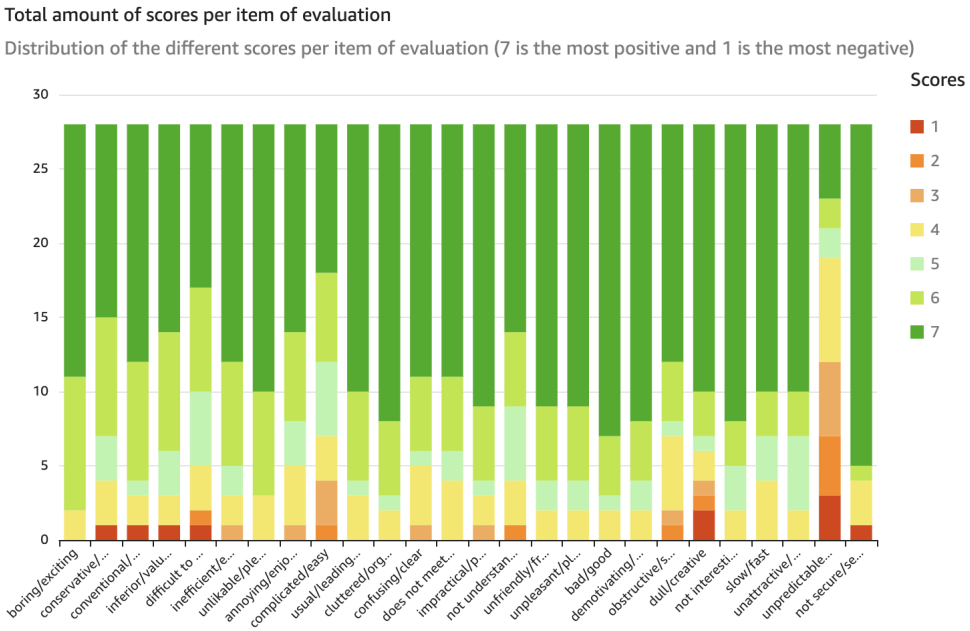


Figure 5.7: Total amount of scores per item of evaluation.

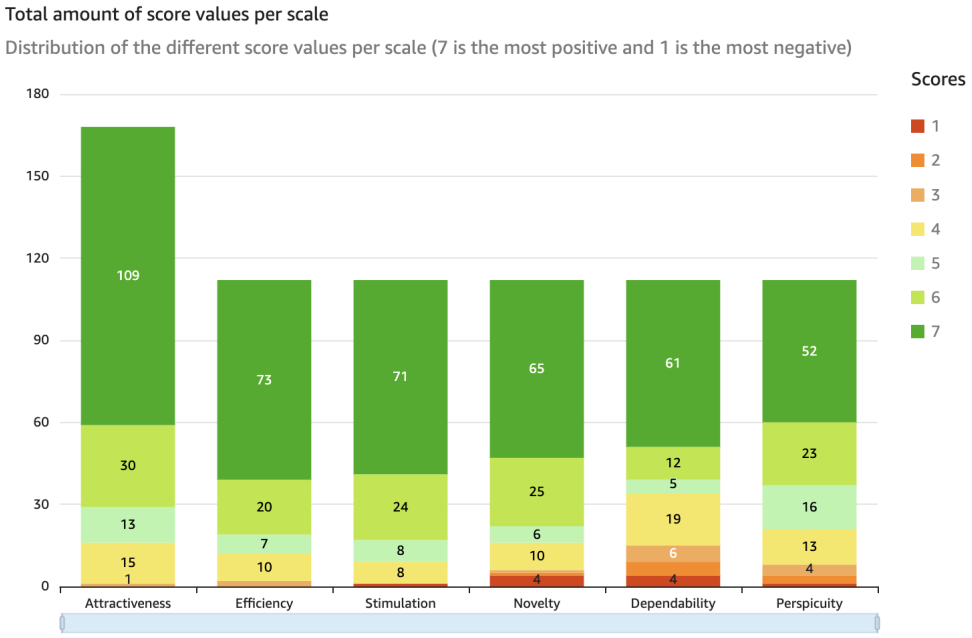


Figure 5.8: Total amount of score values per scale.

The Figure 5.9 represents the evaluation data, where the score values were grouped into positive, negative and neutral categories. The horizontal bars show the percentage of the group rates for each attribute. It is easily distinguished that the biggest strength of the Blockly-C, according to the participants, is its attractiveness. In addition this visual shows that for further development of the tool, the focus should be the dependability.

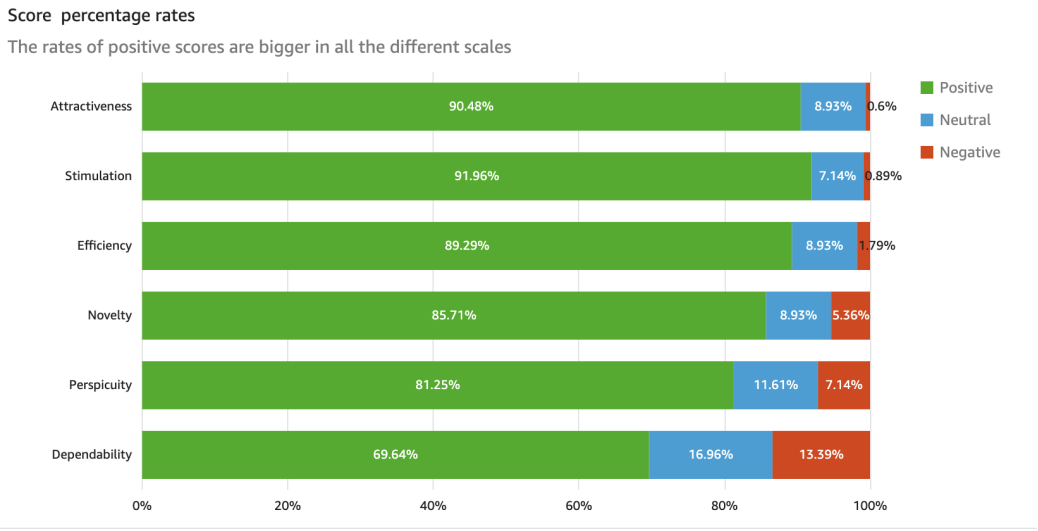


Figure 5.9: Score percentage rates.

A more detailed drill-down is displayed in the sankey diagram (Figure 5.10) where it can be seen how the evaluation items are arranged in the different attributes, organized by the size of the positive scores. Again here, as expected, the attractiveness had the most positive scores.

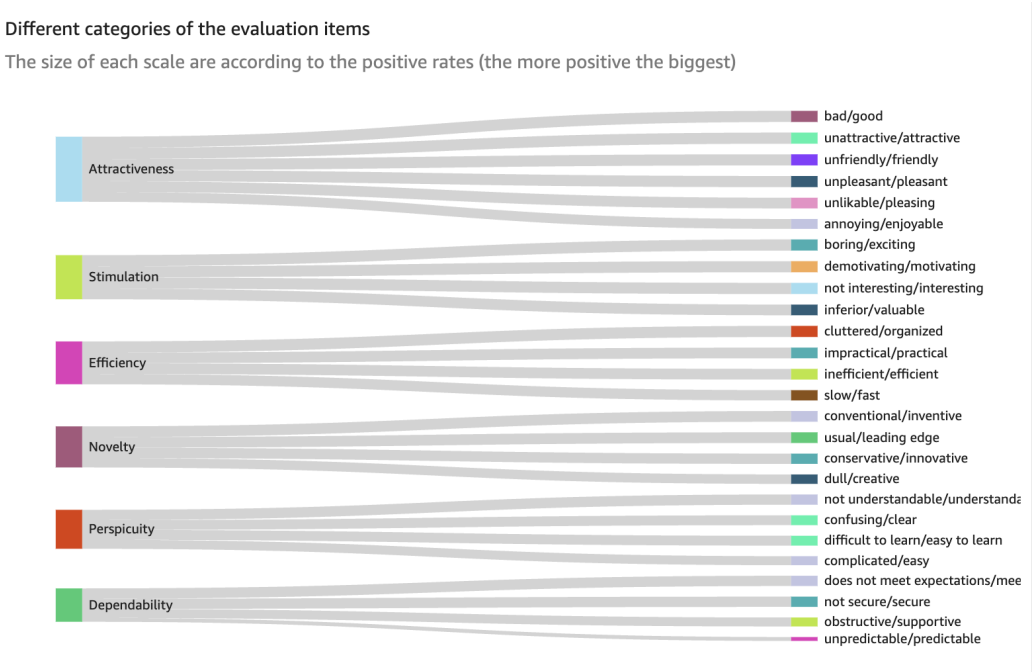


Figure 5.10: Different categories of the evaluation items.

The tables in Figure 5.11 shows the formation of the evaluation items into the attribute categories and the number of positive, negative and neutral votes. The total and subtotal are also displayed in the table for a more detailed view of the data.

Number of positive, negatine and neatral evaluations per category

Scale	Item	Positive	Neutral	Negative
⊟ Attractiveness	annoying/enjoyable	23	4	1
	bad/good	26	2	0
	unattractive/attractive	26	2	0
	unfriendly/friendly	26	2	0
	unlikable/pleasing	25	3	0
	unpleasant/pleasant	26	2	0
	Attractiveness Subtotal	152	15	1
⊟ Dependability	does not meet ...	24	4	0
	not secure/secure	24	3	1
	obstructive/supportive	21	5	2
	unpredictable/predictable	9	7	12
	Dependability Subtotal	78	19	15
⊟ Efficiency	cluttered/organized	26	2	0
	impractical/practical	25	2	1
	inefficient/efficient	25	2	1
	slow/fast	24	4	0
	Efficiency Subtotal	100	10	2
⊟ Novelty	conservative/innovative	24	3	1
	conventional/inventive	25	2	1
	dull/creative	22	2	4
	usual/leading edge	25	3	0
	Novelty Subtotal	96	10	6
⊟ Perspicuity	complicated/easy	21	3	4
	confusing/clear	23	4	1
	difficult to learn/easy to learn	23	3	2
	not ...	24	3	1
	Perspicuity Subtotal	91	13	8
⊟ Stimulation	boring/exciting	26	2	0
	demotivating/motivating	26	2	0
	inferior/valuable	25	2	1
	not interesting/interesting	26	2	0
	Stimulation Subtotal	103	8	1
Total		620	75	33

Figure 5.11: Number of positive, negative and neutral evaluations per category.

The benchmark results that the UEQ analysis tool gave can be shown in figures 5.12, 5.13 and 5.14.

Scale	Mean	Comparisson to benchmark	Interpretation
Attractiveness	2,38	**Excellent**	In the range of the 10% best results
Perspicuity	1,83	**Good**	10% of results better, 75% of results worse
Efficiency	2,36	**Excellent**	In the range of the 10% best results
Dependability	1,64	**Good**	10% of results better, 75% of results worse
Stimulation	2,38	**Excellent**	In the range of the 10% best results
Novelty	2,11	**Excellent**	In the range of the 10% best results

Figure 5.12: Benchmark results of UEQ (part 1).

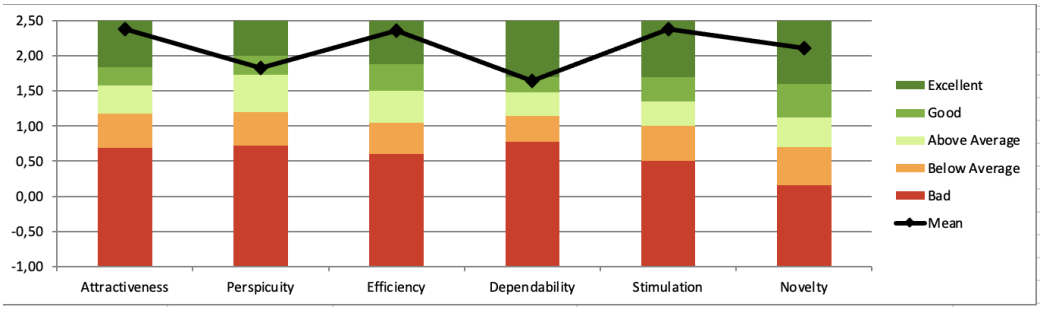


Figure 5.13: Benchmark results of UEQ (part 2).

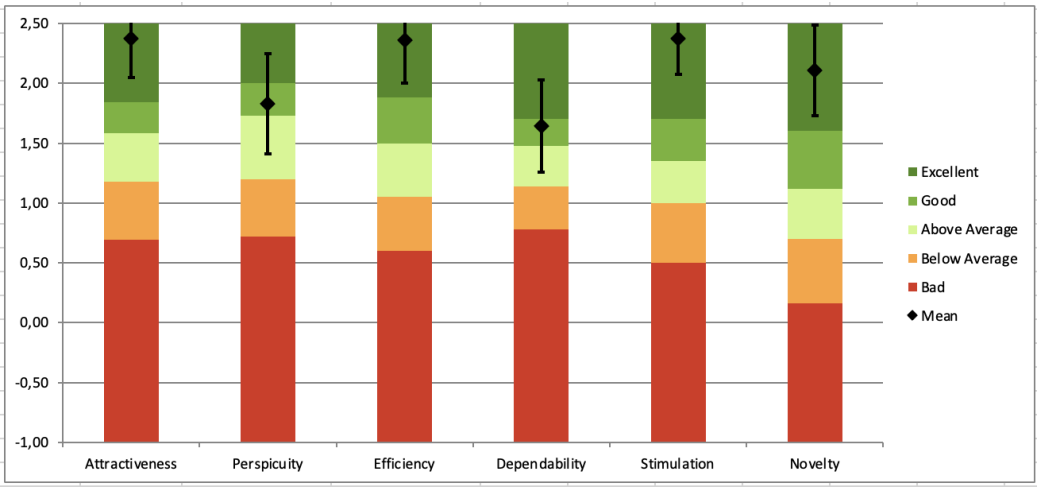


Figure 5.14: Benchmark results of UEQ (part 3).

The benchmark results from the User Experience Questionnaire (UEQ) showcase a comprehensive evaluation of the tool across various user experience dimensions. The tool's Attractiveness is highly praised, with a mean score of 2.38, earning an 'Excellent' classification. This score places it within the elite range of the top 10% of all results, suggesting that users find the tool exceptionally appealing and pleasing to use.

Perspicuity, which assesses how clear and easy to understand the tool is, received a 'Good' rating with a mean score of 1.83. This indicates that there is a smaller margin for improvement, as it surpasses 10% of other results but still falls behind 75%, indicating that most users found it relatively straightforward, yet there's potential for making it even clearer.

Efficiency is another strong suit for the tool, with a mean score of 2.36, it is rated as 'Excellent'. This high score positions the tool within the top 10% of results, reflecting that users perceive it to be highly efficient in its operation.

Dependability of the tool is also viewed positively, garnering a mean score of 1.64 and a 'Good' rating. While it outperforms 10% of results, a significant majority — 75% — are rated higher, suggesting that while the tool is considered reliable, there are better-performing tools in terms of dependability.

The tool excels in Stimulation with a mean score of 2.38, mirroring the high mark in Attractiveness. An 'Excellent' rating here indicates that users find the tool engaging and motivating, placing it in the highest echelon of the top 10% of results.

Lastly, Novelty achieves an 'Excellent' rating with a mean of 2.11. This score signifies that the tool is considered innovative and original, capturing the interest of users and holding a place among the most inventive 10% of tools evaluated.

Overall, the UEQ results paint a picture of a tool that is well-regarded in its user experience, with standout features in attractiveness, efficiency, stimulation, and novelty, while still acknowledging areas like perspicuity and dependability where there is room for enhancement.

A few short answer questions were included at the end of the questionnaire where the participants were asked to answer about further advancement of the tool in a user-friendly aspect and the utility in the learning process for a new starter, among other things. The feedback were again very positive where the majority of the participants believed that is already quite user-friendly and found it very useful for a new programmer. Most of the participants appreciated the fact that it was easy to use and they did not

find anything that is missing from the tool. Illustrative examples of these responses are displayed in Figures 3, 5, and 7.

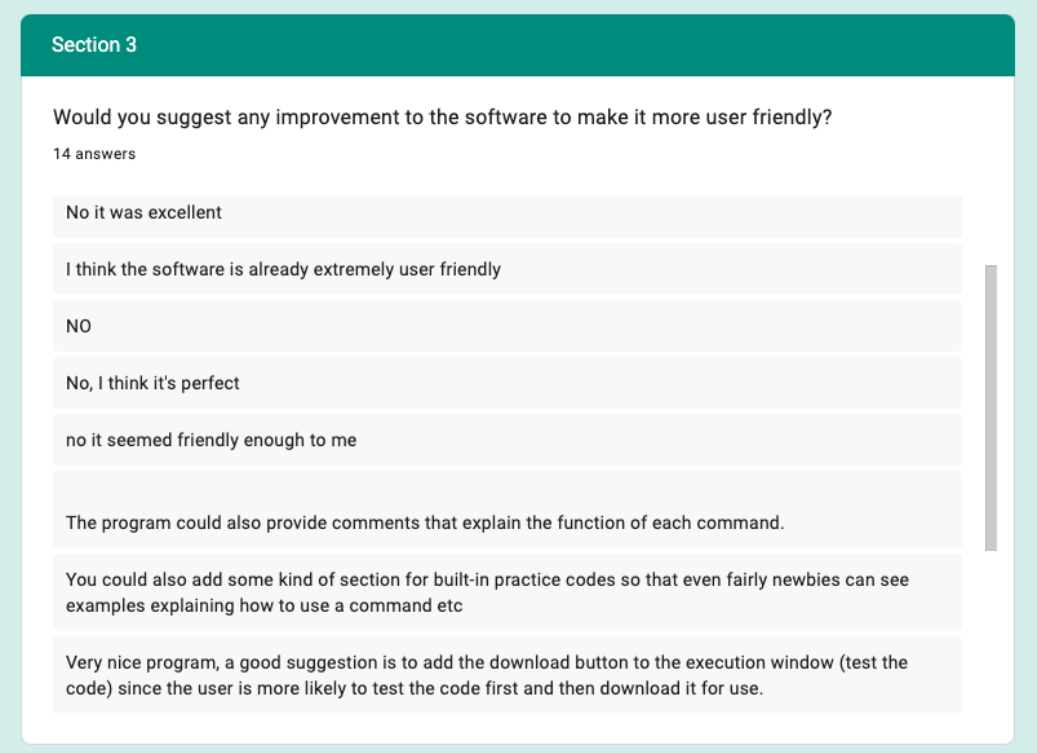


Figure 5.15: Replies for short anwser questions (part 1).

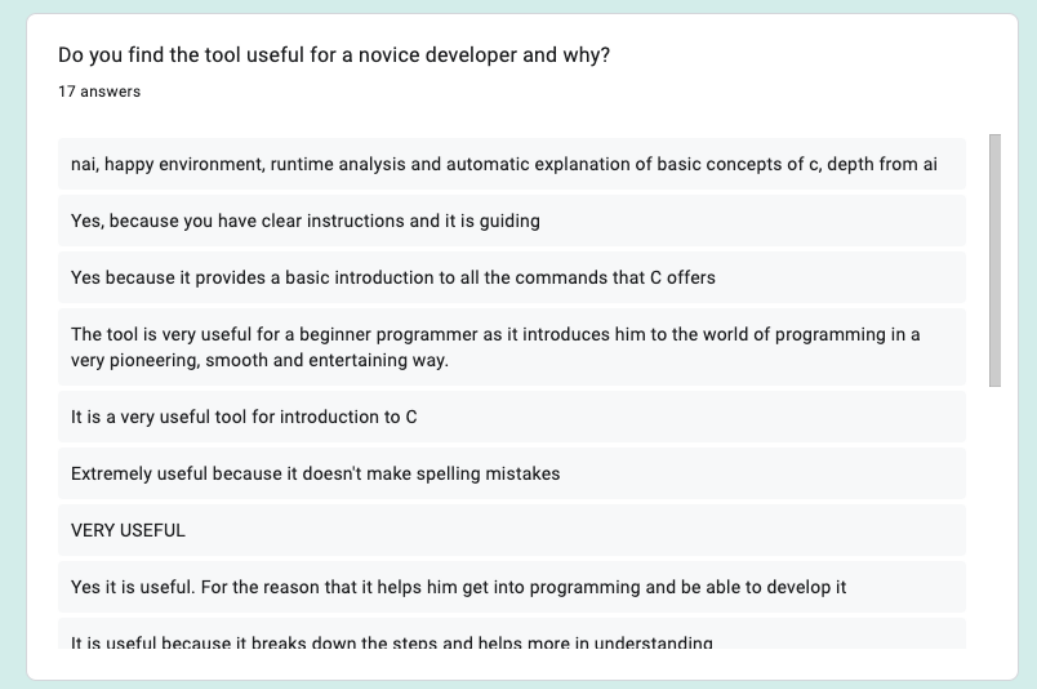


Figure 5.16: Replies for short anwser questions (part 2).

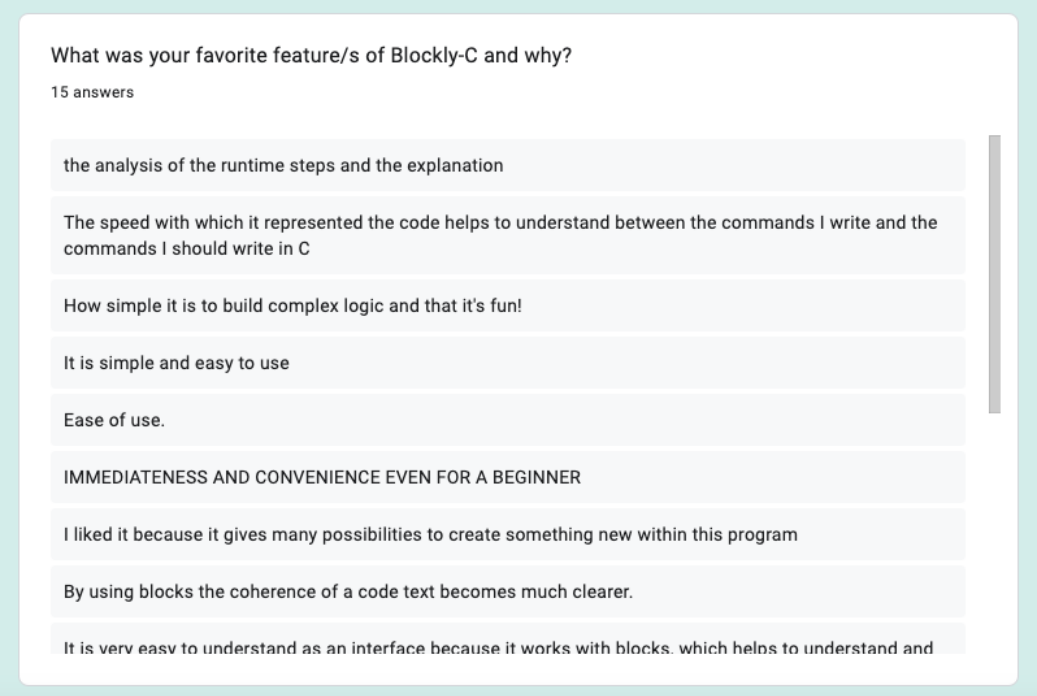


Figure 5.17: Replies for short anwser questions (part 3).

Overall, the questionnaire provided insightful feedback regarding the tool from a user perspective, garnering overwhelmingly positive responses. Participants praised it as being well-organized, engaging, user-friendly, and visually appealing, using descriptors such as 'good,' 'exciting,' 'motivated,' 'interesting,' 'attractive,' 'friendly,' and 'pleasant.' Although negative feedback was minimal, it is crucial to consider these perspectives for future enhancements. Key areas for improvement include enhancing predictability, simplifying usability, and boosting creativity.

Chapter 6

Conclusion and future work

6.1 Introduction

In this thesis, we introduced Blockly-C, a web-based educational platform designed to facilitate learning in C programming through a visual, block-based interface. The thesis comprehensively covered the development process, the underlying technologies, and the key features of Blockly-C, showcasing its potential as an innovative tool for programming education. Experimental evaluations were conducted to validate its effectiveness and usability, providing insights into its practical application in educational settings.

However, the journey of Blockly-C is not complete, and there are opportunities for further research and development. This concluding chapter summarizes the main contributions of this work and outlines potential future directions for Blockly-C. It also addresses the limitations and challenges encountered during the development process, emphasizing the need for ongoing refinement to fully harness Blockly-C's capabilities in real-world educational environments. By identifying these future research areas, this thesis aims to encourage continued innovation in programming education tools, striving to make learning programming more accessible and engaging for students worldwide.

6.2 Conclusions

The completion of this thesis marks a significant milestone in the development of Blockly-C, a web-based visual programming environment designed

for novice C programmers. The feedback from the evaluation phase, conducted with students from the University of Chania, has been overwhelmingly positive, highlighting the tool's user-friendliness and its efficacy as an educational resource.

The evaluation process, encompassing a comprehensive demonstration and a structured survey, provided invaluable insights into the user experience and effectiveness of Blockly-C. Participants, primarily students engaged with the tool, constructing and manipulating C programs using the block-based interface. The exercise demonstrated Blockly-C's intuitive design, allowing users to seamlessly translate logical constructs into executable C code. This practical approach to learning programming concepts, particularly those unique to C, like pointers and memory management, was well-received.

A notable aspect of the feedback was the emphasis on Blockly-C's role in reducing the intimidation and complexity often associated with learning a new programming language. Students appreciated the immediate visual feedback and the error prevention mechanisms inherent in the block-based design. This feature was particularly beneficial in helping them understand the syntax and structure of C programming without getting bogged down by common syntactical errors.

The integration of Python Tutor's visualization tool for step-by-step code execution and memory visualization was another highlight. This feature allowed students to observe the dynamic changes in program state, offering a deeper understanding of how their code manipulates data and memory, a crucial aspect of C programming.

While the feedback was predominantly positive, it also opened avenues for future enhancements. Suggestions for improvements included the addition of more advanced programming constructs and features to facilitate a smoother transition from block-based to text-based coding. This feedback will be invaluable in guiding future development phases of Blockly-C, ensuring it continues to evolve as a comprehensive educational tool that meets the needs of its users.

In conclusion, Blockly-C has proven to be a significant step forward in programming education tools. Its ability to simplify complex programming concepts, combined with an intuitive and engaging user interface, makes it an ideal platform for novices beginning their journey into the world of C programming. The success of this project lays a strong foundation for future research and development in the field of educational programming tools, with the potential to expand its scope to accommodate more languages and

advanced programming topics. This thesis not only contributes to the field of computer science education but also sets the stage for further innovations that can make learning to program more accessible and enjoyable for everyone.

6.3 Future Work

The future development of Blockly-C presents a horizon rich with opportunities for enhancement and further research. Building on the positive reception and constructive feedback received from students, the next steps aim to refine and expand the tool's scope and utility in various educational contexts.

A pivotal aspect of future work involves a more detailed and comparative evaluation of Blockly-C. Envisioned is a comprehensive study within real-life educational settings, such as middle and high schools, where the effectiveness of Blockly-C can be contrasted with traditional C programming editors. This study would ideally involve student groups using either a standard C editor or Blockly-C to complete identical programming exercises. By recording the time taken and the number of successful completions, a clear picture of Blockly-C's educational impact compared to conventional methods can be established.

The expansion of Blockly-C to mobile platforms stands as another significant area for development. This enhancement would not only increase the accessibility of Blockly-C but also allow users the flexibility to learn and practice programming across various devices. Such a move would position Blockly-C as a more versatile and user-friendly educational tool, appealing to a broader audience.

Another valuable addition to Blockly-C would be the integration of precreated C code examples, complete with detailed explanations and case studies. These embedded examples would serve as learning aids, helping users grasp how to apply Blockly-C in diverse programming scenarios and deepening their understanding of C programming.

Enhancing each Blockly-C block with explanatory descriptions could revolutionize the learning experience for users. By providing informative details on each block's purpose and application, users, especially those new to programming, could independently understand and utilize the tool more effectively. This feature is particularly crucial for self-learners who rely on integrated guidance within the tool.

Customizable learning paths, tailored to individual user proficiency, represent a step towards a more personalized and adaptive learning tool. Such customization ensures that users engage with content that is commensurate with their skill level, thus optimizing the learning experience.

Incorporating gamification elements into Blockly-C could significantly boost user engagement and motivation. Elements like achievements, leaderboards, and interactive challenges would make the learning process more dynamic and enjoyable, particularly for younger learners who respond well to game-like learning environments.

The establishment of an online community for Blockly-C users can foster collaboration, knowledge sharing, and peer learning. Besides being a hub for collective growth and support, such a community would provide invaluable feedback for continual improvement of Blockly-C.

Finally, conducting longitudinal research to assess the long-term impact of Blockly-C on learning outcomes and retention of programming concepts is essential. This research would offer deeper insights into how effective Blockly-C and similar tools are in the realm of programming education, thus guiding future educational tools development.

In summary, the roadmap for Blockly-C's future involves not only technical enhancements but also an expansion in its educational approach and research into its long-term efficacy. Each of these proposed directions converges towards making Blockly-C a more comprehensive, engaging, and accessible tool, contributing significantly to the landscape of programming education.

References

[1] K Abe, Y Fukawa, and T Tanaka. "Prototype of visual programming environment for C language novice programmer". In: *Proc. 8th Int. Congr. Adv. Appl. Informat. (IIAI-AAI)*. 2019, pp. 140–145.

[2] *Blockly: Official website*. `https://developers.google.com/blockly`. Accessed: 2023-12-12.

[3] M Craig and A Petersen. "Student difficulties with pointer concepts in c". In: *Proceedings of the Australasian Computer Science Week Multiconference*. ACM, 2016.

[4] PJ Guo. "Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education". In: *Proc. of the 44th ACM Technical Symposium on Computer Science Education*. ACM, 2013, pp. 579–584.

[5] M Heinsen Egan and C McDonald. "Reducing Novice C Programmers' Frustration through Improved Runtime Error Checking". In: *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 2013, pp. 322–322.

[6] M Heinsen Egan and C McDonald. "Runtime error checking for novice C programmers". In: *4th Annual International Conference on Computer Science Education: Innovation and Technology (CSEIT 2013)*. 2013, pp. 1–9.

[7] R Ishizue et al. "PVC.js: Visualizing C programs on Web browsers for novices". In: *Proc. 49th ACM Tech. Symp. Comput. Sci. Educ.* 2018, pp. 245–250.

[8] *Jison: Official website*. `https://gerhobbelt.github.io/jison/docs/`. Accessed: 2023-12-12.

[9] C Kyfonidis, N Moumoutzis, and S Christodoulakis. "Block-C: A block-based programming teaching tool to facilitate introductory C programming courses". In: *IEEE Global Engineering Education Conference, EDUCON*. 2017, pp. 570–579.

[10] E Lahtinen, K Ala-Mutka, and H-M Järvinen. "A study of the difficulties of novice programmers". In: *ACM SIGCSE Bulletin*. ACM, 2005.

[11] W Lidwell, K Holden, and J Butler. *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design*. Rockport Pub, 2010.

[12] I Milne and G Rowe. "Difficulties in learning and teaching programming-views of students and tutors". In: *Educ. Inf. Technol.* (2002).

[13] *User Experience Questionnaire (UEQ: Official website*. `https://www.ueq-online.org/`. Accessed: 2023-12-12.

[14] D Weintrop and U Wilensky. "Comparing blocks-based and text-based programming in high school computer science classrooms". In: *ACM Trans. Comput. Educ. TOCE* 18.1 (2018), p. 3.

.1 Appendices

Demonstration Exercise for Blockly-C Web Application

General

- In this demonstration, it will be presented an online platform designed as visual-based educational tool for learning C language.
- Access the web application through the following url: https://blockly.gdsuites.gr/thesis/core/
- The application comprises two primary sections: the code creation/editing page and the code execution/memory visualization page.
- Upon entering, you'll be directed to the editing page. This area consists of two main components: the "Workspace," which occupies the majority of the screen where users arrange the blocks to formulate a program, and the adjacent "Code" field, which displays the generated C code.
- The code can be edited only with the use of the Workspace.
- On the left side of the Workspace, there are placed all the blocks that can be used grouped into categories. Each block corresponds to a specific C language command.
- The application is designed to only allow syntactically and grammatically correct connections between blocks, adhering to the rules of C programming language.

Exercise

As a demonstration exercise, we will build a simple program, which will print in the console a right triangle composed with stars “*”. The desired outcome of the console is this one:

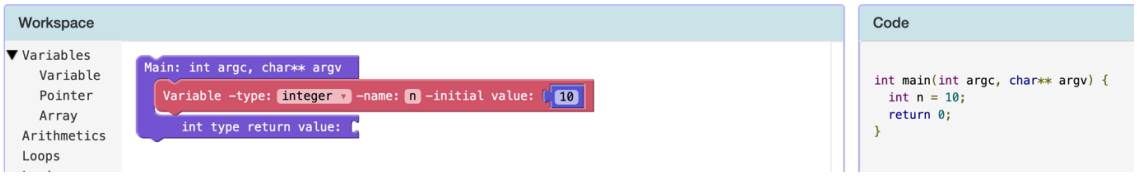
```
*
**
***
****
*****
******
*******
********
*********
```

Below are detailed instructions to guide you through the presentation. For additional assistance and to verify your progress, the final page includes images showcasing the expected appearance of your workspace upon completing the program. This section also presents the corresponding generated C code for reference.

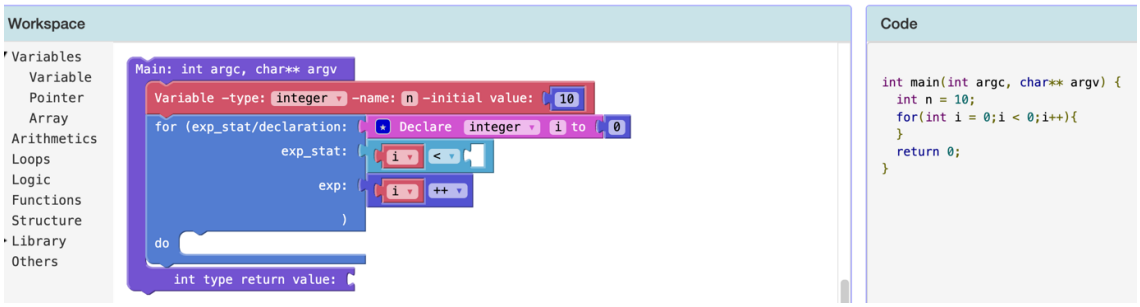
Instructions

1. Upon launching the web application, the workspace automatically is populated with a purple block that represents the main function of C program. This block can't be edited or deleted. All the commands must be placed within the main function.

2. To begin, expand the "Variable" category by clicking on it, then select the sub-category "Variable." This action will display a list of available blocks within this sub-category. Locate the second red block, which it comes with an initial placeholder value. Click, drag, and drop this block into the main function block located in the workspace. This block serves as a single variable declaration in C. As you add new blocks to the workspace, the code section on the right automatically updates, translating each block into its equivalent C code. Modify the default variable name from "i" to "n" and adjust its initial value from "0" to "10". By now your workspace should look like this:



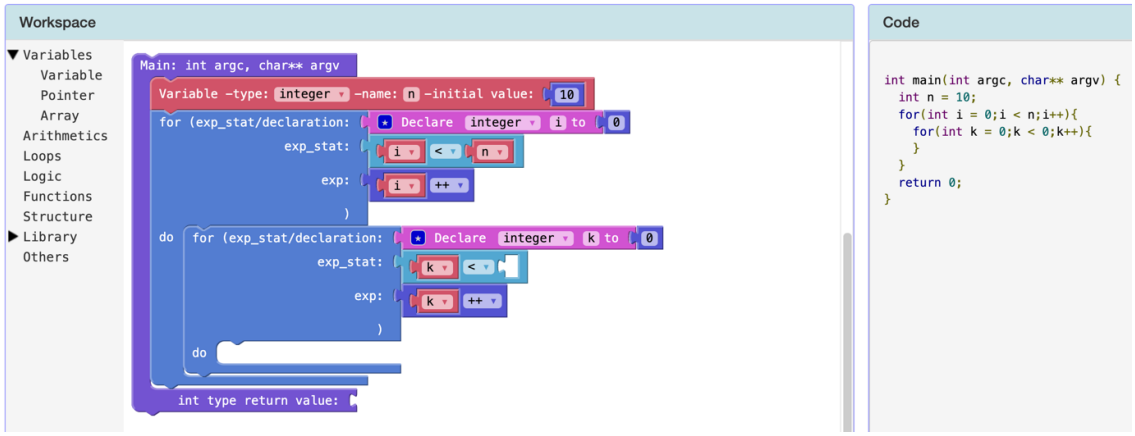
3. Next, navigate to the "Loops" category and select the 7th block in the list, which represents the "for" loop command in C. This block is pre-configured with certain condition values. By default, it declares an integer variable named "i" with an initial value of "0". The loop's execution condition is set to "i < 0", and at the end of each iteration, the value of "i" is incremented by 1 ("i++" is equivalent to "i = i + 1"). Place this block within the main function, positioning it below the previously added Variable Declaration block. To modify the loop's condition, right-click on the blue block within the loop condition that signifies the value "0". This action will highlight the block and open a menu with various options. From this menu, select "Delete Block" to remove it. Alternatively, you can delete a block by left-clicking and dragging it into the trash bin located at the bottom right corner of the workspace. Your workspace should now resemble the following layout:



4. Press again the sub-category “Variable” and select the last block from the list. Position this block where you previously deleted the block in step 3. Click on the “--Select--” field in the block and choose the variable “n” from the dropdown menu options. This block provides a dropdown list featuring all the declared variables accessible at the current code position. The selection of "n" will adjust the loop's condition to utilize the variable "n" that you declared earlier. The condition should be “i < n”.

5. Follow the instructions from step 3 to add another 'for' loop block, but this time, place it inside the 'for' loop you created in step 3. This action creates two nested 'for' loops. As in the previous step, remove the zero value from the condition of the newly added inner 'for' loop.

6. In the inner 'for' loop, change the name of the declared variable from “i2” to “k”. Then, modify the variables in both the loop condition and the increment statement, replacing “i” with “k” to ensure the inner loop operates independently with its own variable. The outcome should be like this:



7. Repeat the step 4 and place the get variable block inside the condition of the **inner** for loop and choose as a value the variable “i”. Its condition should be “k < i”.

8. Navigate to Library-> Stdio and drag the 2nd block of the list into the inner 'for' loop. This block represents the “printf” function of C which prints output to the console. Inside the quotation marks of this block, insert a star character “*”.

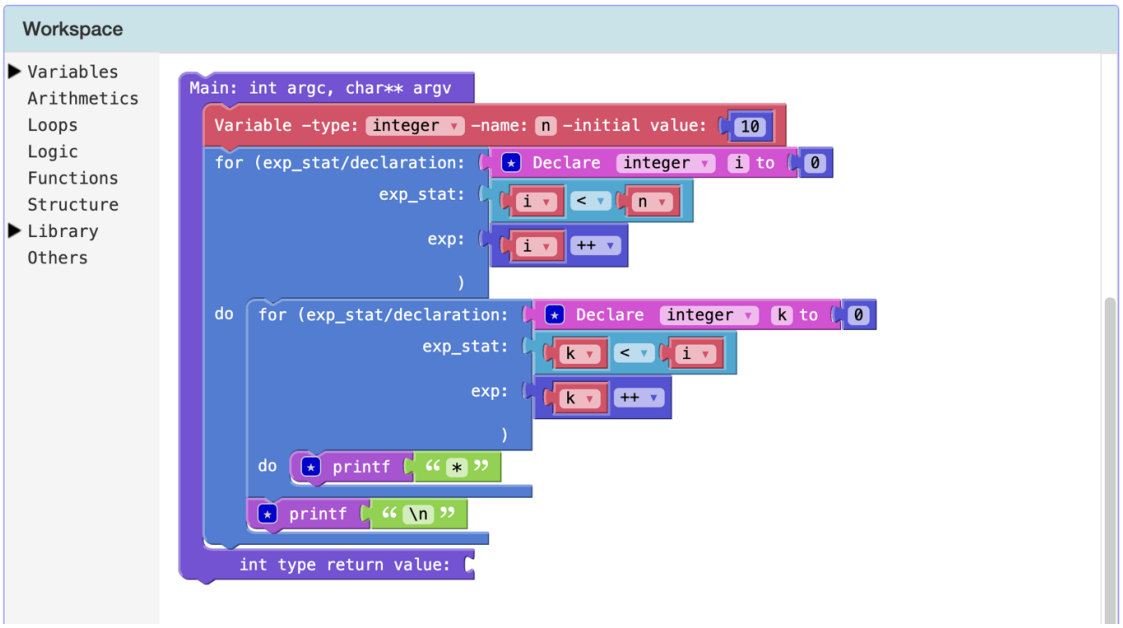
9. Right-click on the "printf" block and select "Duplicate" to create a copy. Place the newly duplicated "printf" block within the outer 'for' loop, positioning it just below the inner 'for' loop.

10. At the newly created “printf” block replace the star (“*”) within the quotation with “\n”

Your final program should like this:

Code

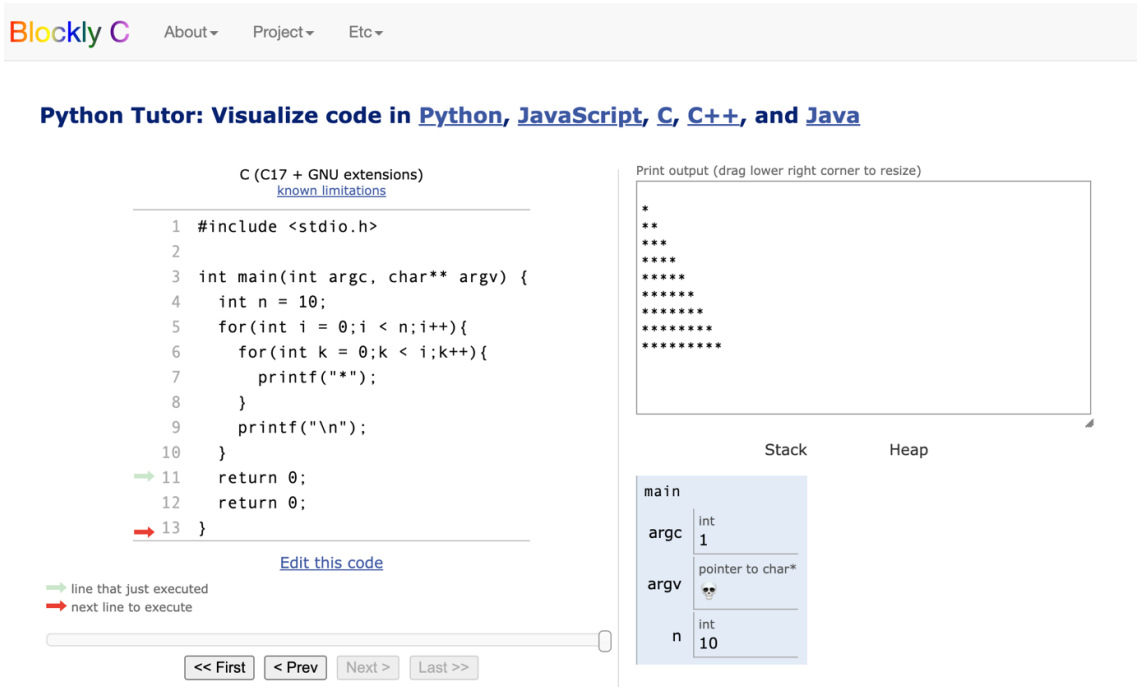
```
#include <stdio.h>

int main(int argc, char** argv) {
  int n = 10;
  for(int i = 0;i < n;i++){
    for(int k = 0;k < i;k++){
      printf("*");
    }
    printf("\n");
  }
  return 0;
}
```



Other Functionalities of Blockly-C

Next will be demonstrated the other functionalities that the web application offers. By clicking the “Download” button at the top bar you can download your code in a .c file on your local computer in order to save your work. The “trash” button offers a reset function, allowing users to declutter their workspace by removing all added blocks. In addition, the “Import” button allows you to import any existing .c file from your local directory. It compiles the uploaded code and converts it into visual blocks, allowing you edit it and read it more easily. The “run” button re-directs you to the visualization page which you can actually run your code and visualize the stack and heap memory of your program live during execution.

1. Press the “Download” button and store your code into your computer.
2. Press the “Trash” button and delete all your workspace.
3. Press the “Import” button and choose the file that you saved in step 1.
4. Press the “Run” button and wait a few seconds to compile it.
5. After compilation press sequentially the “Next >” button and observe the live visualization of memory until you reach at the end of the program.
6. When the program is executed, you will be able to see right rectangle of stars printed in the console:



As a final step of the presentation please fill out the evaluation form for the web application at:
https://docs.google.com/forms/d/e/1FAIpQLSfi5lsvRNsirVzar2no3e5jEtkUo7vMKi8k9lA0BSOcei0CCA/viewform?usp=sf_link