An Interactive C Code Execution and Visualization Tool for Online Learning

Veli-Matti Rantanen

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo 7.7.2023

Supervisor

Prof. Riku Jäntti

Advisor

Dr. Yusein Ali

Aalto University
School of Science

Copyright © 2023 Veli-Matti Rantanen

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Veli-Matti Rantanen

Title An Interactive C Code Execution and Visualization Tool for Online Learning

Degree programme Computer, Communication and Information Sciences

Major Security and Cloud Computing **Code of major** SCI3084

Supervisor Prof. Riku Jäntti

Advisor Dr. Yusein Ali

Date 7.7.2023 **Number of pages** 38 **Language** English

Abstract

Introductory programming courses are notorious for their high drop-out rates and the C programming course at Aalto University is no exception. Studies have suggested that this phenomenon is caused by a combination of factors that result in students experiencing significant amounts of stress. Previous works have sought to address the problem by introducing new pedagogic methods and software. In addition, studies have found that intelligent tutoring can be as effective as traditional tutoring methods.

This thesis presents a web-based software that seeks to reduce the stress factors associated with learning the C programming language. The goal of the work is to develop practical methods of generating and visualizing intelligent feedback. The presented tool is capable of analyzing, visualizing and evaluating simple user programs.

Initial impressions suggest that the tool could reduce the stress factors significantly, but the implementation has significant limitations. Based on observations made in the implementation of this work, the development of an open-source framework for implementing similar tools is recommended.

Keywords Intelligent tutoring, visualization, tool, programming, novice programmer, introductory course

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Veli-Matti Rantanen

Työn nimi Vuorovaikutteinen tietokoneohjelma C-koodin tulkitsemiseen ja havainnollistamiseen verkko-oppimisympäristössä

Koulutusohjelma Computer, Communication and Information Sciences

Pääaine Security and Cloud Computing **Pääaineen koodi** SCI3084

Työn valvoja Prof. Riku Jäntti

Työn ohjaaja TkT Yusein Ali

Päivämäärä 7.7.2023 **Sivumäärä** 38 **Kieli** Englanti

Tiivistelmä

Ohjelmoinnin alkeiskurssit ovat tunnettuja tavallista korkeammasta keskeyttämisasteestaan, mikä on havaittu myös Aalto-yliopiston C-ohjelmoinnin peruskurssilla. Tutkimusten mukaan tämä ilmiö syntyy erilaisten kuormittavien tekijöiden yhteisvaikutuksena. Tätä kuormittavuutta on yritetty vähentää erilaisin kasvatustieteellisin ja teknisin keinoin. Tutkimuksissa on myös selvinnyt, että niin kutsuttu älykäs tuutorointi (eng. *intelligent tutoring*) on oppimisen apuvälineenä miltei yhtä tehokas kuin perinteinen tuutorointi.

Tämä diplomityö esittelee verkkoympäristöön toteutetun tietokoneohjelman, joka pyrkii vähentämään kuormittavia tekijöitä C-ohjelmointikielen oppimisessa. Työn tavoitteena on kehittää käytännöllisiä keinoja älykkään palautteen tuottamiseen ja havainnollistamiseen. Toteutettu tietokoneohjelma kykenee analysoimaan, havainnollistamaan ja suorittamaan yksinkertaisia käyttäjän syöttämiä tietokoneohjelmia.

Ensivaikutelmien perusteella ohjelma voisi vähentää kuormitusta merkittävästi, mutta toteutuksessa on käyttöä rajoittavia puutteita. Työssä tehtyjen havaintojen perusteella suositellaan toteutettavaksi avoimen lähdekoodin ohjelmakehystä, joka voisi tulevaisuudessa helpottaa vastaavien työkalujen toteuttamista.

Avainsanat Älykäs palaute, havainnollistaminen, ohjelma, ohjelmointi, aloitteleva ohjelmoija, peruskurssi

Preface

I want to thank my advisor, Dr. Yusein Ali, for the opportunity to work on this topic and their outstanding guidance. I would also like to thank my supervisor, Prof. Riku Jäntti, for their advice and involvement in this work.

I must also extend my thanks to Dr Kalle Ruttik and Dr Jari Lietzén for their feedback on many aspects of this thesis. I am indebted to Roni Fagerholm who was of the greatest assistance in troubleshooting the implementation described herein and who participated in the proofreading of this thesis. I would also like to thank the many other colleagues that I have had the pleasure to work with at the Department of Communications and Networking.

Lastly, I would like to thank my wife and our children for their unwavering support and encouragement, without which the completion of this thesis would have been all but impossible.

Otaniemi, 31.7.2023

Veli-Matti J. Rantanen

Contents

Symbols and abbreviations

Symbols

$\bot$	Nothing, “null”
ε	Empty string
α, β, γ	Strings of any length
A, B	Non-terminals

Operators

$\leftarrow$	Assignment
$\rightarrow$	Implication

Abbreviations

A+	A+ Learning Management System
API	Application Programming Interface
AST	Abstract syntax tree
C	The C programming language, ISO/IEC 9899
C99	The 1999 revision of the above, ISO/IEC 9899:1999
CFG	Context-free grammar
CSG	Context-sensitive grammar
IDE	Integrated development environment
IPT	Intelligent programming tutor
ITS	Intelligent tutoring system
LMS	Learning management system
OOP	Object-oriented programming
PEG	Parsing expression grammar
UI	User interface

1 Introduction

Computers – in the broadest sense – and their programs are a ubiquituous necessity in the 21st century. The average person is likely to own multiple computers, such as smartphones, televisions and traditional computers. Beyond such obvious devices, many appliances and vehicles incorporate microcontrollers, which are (effectively) miniaturized computers. Similarly, most organizations depend on software-based services to perform their daily functions. Every sector from finance to healthcare and manufacturing has become dependent on digital services and devices. Recent trends, such as the cloud and the internet of things, have brought computers into the most mundane of objects and circumstances. The programs used to provide these services and control the devices are as varied as they are numerous; each application presents unique requirements upon the programs used.

As a consequence of the ubiquity of computers, there is an ever-increasing demand for professional programmers, software developers, software architects, scrum masters and many other job titles. As such, computer programming, computational thinking and practical computer skills are taught worldwide at many different levels – from K–12 to university-level [1], [2]. However, programming is not an easy skill to learn; introductory programming courses worldwide report low pass rates and elevated rates of student burnout [2]. The challenges that these novice programmers encounter are as diverse as they are numerous. Learning a new programming language alongside the principles of computing and software engineering accelerates student burnout rates [3]. This effect is often referred to as the "CS1 phenomenon" and numerous studies have tried to address it [4]–[7].

The most challenging topics for novice programmers have been related to the software engineering side of programming, where one must reason about their program in terms of abstractions [3], [4]. Novice programmers themselves identify tasks related to the structure and analysis of a program as the most challenging: designing, subdividing and debugging a program [6]. Almost by definition, the novice programmer exhibits an inability to reason about a program as a whole [4]. This inability has been termed as *fragile knowledge* [8]. Fragile knowledge in the context of programming is defined by Lister et al. as a state of knowledge where an individual is able to accurately recount fundamental concepts and terms or trace small sections of code, but not describe the function of an entire program [9].

Numerous software tools [5], [10] have been developed to aid learners in addressing fragile knowledge and other difficulties. These tools are generally developed for the use of a particular course or some particular school [10]. This thesis reviews a select few of these tools as representative of the state-of-the-art. A handful of related tools are discussed but were not discovered early enough to inform the development of the tool.

This thesis presents a novel tool developed to assist novice programmers in learning the C programming language. The tool – CVisualizer – is designed to assist students in understanding their own programs through lexical, syntactic and semantic analysis, visualizations, descriptions and interactivity. The architectural design and key implementation details of the system are presented. The intent of this work was

to develop practical approaches to generate and visualize intelligent feedback for user programs. The implementation of the tool has significant deficiencies that limit its applicability.

This thesis consists of five chapters, of which this introduction is the first. The next chapter introduces certain background topics relevant to this thesis. The third and fourth present a high-level specification and an implementation of said specification, respectively. Lastly, the fifth chapter contains the concluding remarks on this thesis and the author's thoughts on future possibilities.

2 Background

This chapter is dedicated to discussing the challenges in learning programming, the related concepts and prior work in addressing these challenges. This chapter is divided into three sub-chapters. The first sub-chapter discusses computer programs in general. The second sub-chapter explores the challenges in learning programming and methods of overcoming said challenges. The third and final sub-chapter reviews prior works in the field.

2.1 Computer programs

Computer programs are descriptions of processes that allow a computer to perform the steps of that process. In this sub-chapter, the methods by which a computer program is defined, used and developed are discussed. This sub-chapter consists of three sections. The first section describes the structure and limitations of a programming language. The second section discusses the manner in which a computer program is executed. The last section discusses the tools commonly used for developing computer programs in the real world.

2.1.1 Programming languages

Computer programs are generally defined with a programming language. A programming language is a structured, formal language that accurately specifies the behaviour of a computer program [11]. These languages allow the programmer to prescribe the behaviour of the program as a sequence of steps to take. I.e. they allow a process or protocol to be formally specified.

A programming language is defined with a formal grammar, which describes all syntactically correct programs in that language in terms of rules. It should be noted that formal grammars are not limited to defining programming languages but can be used in the context of any language [12]. One example of a rule is $A \rightarrow \alpha$, which defines that all instances of the token A in the grammar can be replaced with the string α [11]. The grammar G defines the language $L(G)$ to which a sequence of tokens (string) $s = t_1 t_2 t_3 ... t_n$ can belong ($s \in L(G)$). For example, this sentence is a string of tokens that belongs in the English language. In the context of programming, these strings are programs written in a programming language and the tokens are symbols meaningful to the interpretation of the program, such as keywords, identifiers, operators and literals. In the interest of conciseness, we will not formally define these terms.

For a computer to understand a string written in $L(G)$, the string must be transformed to a representation that allows the analysis of the inter-dependencies within that string. For this purpose, a parser is used. A parser is an algorithm that, given a grammar G and an input s, either *accepts* the input if $s \in L(G)$ or *rejects* it otherwise. Any accepted input is *syntactically valid* and any rejected input contains one or more *syntax errors*. In addition to classifying inputs, most parsers generate a data structure, called *abstract syntax tree* (AST), that allows the input to

be analyzed programmatically in the context of the language concepts. This data structure associates the matched rules to the input tokens that constitute them. [11]

The semantics for some languages – including C – require context-sensitive behaviour. In the general case, a parser that permits context-sensitive rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ is drastically less efficient than one that does not [11], [13]. In some cases, this behaviour is implemented as an additional analysis step after parsing while in others a separate parsing mechanism is utilized [11]. One common example of the former is requiring variables to be defined before they are used. An example of the latter, which is also found in C, is that any identifier can be defined as a typename with a language construct. As the parser must know whether a token is an identifier or a typename to correctly parse the input, additional analysis during the parse is required. This particular example is solved by maintaining a so-called *symbol table*, which informs the parser on whether a given identifier is a typename or not [11].

In addition to specifying the steps of a process, most programming languages allow the programmer to apply *abstractions*, such as procedures and data structures. The purpose of abstractions is to ease the definition of high-level concepts (e.g. user interface, messaging) in terms of low-level ones (e.g. addition, memory access)[14]. These mechanisms can take many forms, some of which are discussed in the next sub-chapter.

2.1.2 Structure of a computer program

According to Abelson, programming language provides three mechanisms for describing complex processes: primitive expressions, means of combination and means of abstraction. Primitive expressions are the basic directives of the language, such as arithmetic operations, logical operations and control flow directives. Means of combination refer to methods of combining concepts into new ones, such as sequences or procedures, and means of abstraction refer to methods that allow the programmer to name the concepts that they introduce. [14]

To be useful, a computer program must also be able to interact with its environment; i.e. it must accept input and produce output. This capability is called input/output (I/O) and can take many abstract forms: e.g. file systems, graphical interfaces and mechanical phenomena [14]. These high-level concepts, in turn, are abstractions that are built on top of layers of lower-level concepts. At the very lowest level, the program influences the physical electric circuitry that produces these effects. It is these abstractions that give rise to the notional machine [15].

The *notional machine* is an abstract model of a computer that a programmer uses to reason about the behaviour of computer programs [4]. This reasoning is done in terms of abstract operations that the machine is capable of, rather than the physical capabilities of of the real computer. The abstract operations are generally drawn from the programming language or the problem domain [15]. I.e. a single language may present any number of notional machines of different complexities [15]. An experienced programmer can use more than one notional machine to reason about the behaviour of a program. The ability to construct an accurate and consistent

notional machine is inextricably linked to the programmer's expertise [4].

2.1.3 Industry tools

Many software tools are used by professionals to assist in producing software, including text editors, compilers [11] and debuggers [16]. These tools are also commonly used in introductory programming courses, but present a cause of significant amounts of difficulty for the students [6], [17] which are discussed in 2.2.1.

As discussed above, a computer program is generally expressed by the programmer in a programming language. These languages are generally written into *source files* containing only the textual representation of the program [11], [14]. In the context relevant for this thesis, these files are then processed by a compiler. The compiler is a computer program that produces an executable binary file (executable) from the source files. [11]

Once an executable is produced, a debugger can be used to inspect its operation. A debugger can be attached to the software either before the software is executed or while it is in operation. These tools allow the user to inspect the program state and to navigate the program either step-by-step or by utilizing breakpoints. Beyond viewing the state of the program, a debugger also allows replacing code and modifying values while the program is running. To make the experience of using a debugger more enjoyable, most compilers support the generation of debugging symbols. These allow the debugger to determine where in the source code a given instruction in the program originated from. [16], [18]

2.2 Challenges in learning programming

In this sub-chapter, we introduce the challenges that novice programmers face and discuss some prior approaches to enhance the learning outcomes in introductory programming courses.

2.2.1 General challenges

It is widely understood that programming is difficult; an assumption that is supported by observations of increased student dropout and burnout rates as well as significant amounts of work done to make the skill more approachable. Bower et al. identifies five sources of difficulty that novice programmers struggle with, misattributing their description to du Boulay et al [19], [20].

Bower et al. assert that much of the difficulty for novice programmers lies in attempting to tackle all five sources simultaneously [19]. As the first source, they identify general orientation which they describe as an understanding of what computer programs are useful for. However, as Bower et al. do not elaborate on this further, it is difficult to say what the practical relevance is.

The second and third sources identified are the notional machine and the notation of the language being taught [19]. Although Bower et al. make a distinction between the two, these concepts are inextricable as the semantics of a programming language describe the behaviour of a notional machine [15], [20]. It could be said that novice

programmers lack the ability to independently switch or modify their mental model of a computer to fit the problem at hand [15].

It has also been observed that reading and understanding program code is a skill that is distinct from producing said code [4], [9]. In particular, it has been observed that novice programmers are capable of producing code that they are unable to explain. Winslow et al. assert that this ability to understand program code is perhaps even more important than the ability to write programs [4]. However, they also note that the best way improve this ability is to write programs.

The fourth source of difficulty is identified as structures, referring to structured problem solving. These are more commonly identified as schemas and plans in the literature [4], [6], [19]. Bower et al. have made a distinction between schemas and planning [19], which appears strange as schemas are generally thought to encompass all strategies for problem solving [4].

The final source of difficulty identified are practical skills; e.g. testing, debugging and general computer skills. It has been observed that students are generally not taught effective debugging strategies [4], [16], [18] nor the use of debuggers [18]. It has also been observed that students lack many skills necessary to develop programs, such as an understanding of file systems [21].

2.2.2 C programming

C is a programming language standardized by the International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC)[22]. Before its standardization by ANSI in 1989, C was developed as a successor to B and is intended as a portable replacement for Assembly in low-level systems programming [23].Today, C is commonly taught in electrical engineering and information technology programs.

Most textbooks for teaching the C programming language follow the same general outline shown in table 1 [21], [23]. The most challenging topics are annotated with a star [6]. As we can observe, the most challenging topics appear to be packed at the end. This observation is also made by Budny et al, who assert that most programming courses follow the structure of some particular textbook [21]. Budny et al. reason that when the difficult topics are placed at the very end of a course, students have less time to practice these topics which leads to poor knowledge retention.

	1. Introduction
	2. Variables and expressions
	3. Branching
	4. Looping
⋆	5. Functions
⋆	6. Arrays
⋆	7. Pointers

Table 1: Typical table of contents for a C programming textbook

C has two features that are distinct from many other programming languages,

namely its memory management model and pointers. These two are closely related to one another, as pointers are primitive values that reference an object in memory [22]. Unlike higher-level languages, C does not perform so-called garbage collection – the programmer is responsible for ensuring that all memory that is allocated is also released [23]. It is these low-level concepts that make the language more difficult for novices.

There are three methods in C by which memory can be allocated: static, automatic and dynamic. Statically allocated objects persist through the execution of the program, while automatically allocated objects are deallocated when the function defining them returns [22]. The word "deallocated" is used in contrast with higher-level languages where the object is "destroyed". The latter term would imply garbage collection, which C does not perform – the object is simply left wherever it may be and might be overwritten at a later point in time.

The third method, dynamic allocation, refers to the allocation of an object in the *heap*. The heap is a memory region that allows objects with lifetimes other than those of static or automatic to be allocated [22]. When an object is dynamically allocated, it remains allocated until it is explicitly deallocated [22]. The dynamically allocated object is referred to through a pointer.

2.2.3 Pedagogic approaches

Traditionally, programming courses have employed a lecture-oriented approach where the teacher introduces new concepts and language features, which the students rehearse by solving programming exercises at home [4]. There are also approaches where the course itself is structured differently from traditional methods and other disciplines. In particular, it has been suggested that "reverse classroom" and "project-first" models result in better learning outcomes [1]. In the "reverse classroom" model, the course participants peruse the course material on their own and contact teaching is dedicated to solving the assignments. In the "project-first" model, the participants work on small projects and are introduced to new concepts as they become necessary for the project.

Another notable approach is that proposed by Budny et al. whereby the course participants are taught multiple, otherwise unrelated, languages in a single course [21]. In their approach, Excel, HTML and MATLAB are taught before C. In this approach, students are expected to become familiar with general programming concepts before the syntactic structures of C are introduced. A survey has supported that this kind of poly-glossary approach could be effective [24].

As it is inevitable that novices encounter problems that they cannot resolve by themselves, most courses organize exercise sessions where a more experienced programmer is present to provide advice. One-on-one tutoring is held as the gold standard [4], [7] but peer programming has also been found effective [25]. In peer programming, the learners are assigned to pairs or groups and are expected to resolve the problems amongst themselves.

The techniques discussed above tend to be limited in the amount of tutoring available to an individual student due to the limited number of course staff. As

such, most programming courses require significant amounts of independent learning. Contrary to the widely-held belief of tutoring being the most effective teaching method, Lahtinen et al. found in a survey that students feel that they learn the most when working alone on programming coursework [6]. They also found that students felt example programs to be the most helpful resource in learning programming, while visualizations of programs were the least helpful.

2.2.4 Automated approaches

To enhance these independent learning outcomes beyond what is achievable through the methods discussed above in 2.2.3, a wide array of software tools has been developed in the last four decades [5], [7]. Kelleher et al. identify more than 60 tools that have been developed to either teach programming or to empower the user by reducing the entry barriers to programming, classifying the tools into 20 categories divided between these two approaches [5].

The first approach – teaching programming – focuses on making the learning process of a general-purpose language, such as C, C++ or Python, more approachable to a novice programmer. Tools in this category approach the goal in different ways, such as by making the language easier to write or by aiding in the discovery of errors [5]. Intelligent tutoring systems and the related works discussed below, as well as subject of this thesis, fall into this category.

Kelleher et al. continue to describe that in the second approach, the developer seeks to empower the users of the tool by reducing the difficulty of programming [5]. These tools let the user describe their program without a programming language or in a significantly simplified one, such as by drawing a block diagram or by describing the program in a natural language. For many of these tools, the aim is to teach procedural thinking or to foster an interest in programming in general rather than to directly teach programming.

Intelligent tutoring systems (ITS) are software that aid the user in solving course exercises by providing advice in a natural language, emulating the traditional method of a human tutor providing feedback, and have been effectively used in domains including, but not limited to, computer science and programming. Studies have found ITS to be nearly as effective as one-on-one human tutoring [25]. In the context of teaching programming, ITS are sometimes called intelligent programming tutors (IPT). Beyond the feedback provided by ITS, IPT often provide additional functionality. Features found in IPT include questions on the entered program; plan verification and presentation; linkage to lesson material and references; and visualizations. [10]

2.3 Related works

In this sub-chapter, we review prior works. In particular, we focus on software developed to assist novice programmers in learning a general purpose language. This sub-chapter consists of five sections. The first section describes the selection criteria. The three subsequent sections describe the selected tools. The last section discusses

three additional tools that were discovered during the final stages of editing this thesis.

2.3.1 Selection criteria

As discussed previously, there are numerous tools developed to assist novice programmers. As only a limited number of tools could be reviewed for this thesis, the following criteria was used to limit the number of results. Three tools were selected such that each of them represented a distinct approach to the problem on the basis of the abstract.

1. the publication is at most 10 years old
2. the tool is a software tool
3. the tool has been implemented
4. the tool focuses on a general-purpose language

2.3.2 Eclipse DAPS

Eclipse Debugging Assistant for Programming Students (Eclipse DAPS) is an IPT. The author presents an extensive analysis of the rationale behind the tool, its requirements and implementation in [26]. The tool is realized as a plugin for the Eclipse IDE. That is, the student must install the IDE and download the plugin to be able to use the tool. The tool extends the IDE with a handful of interface elements that allow the student to interact with the tool.

One explicit choice made was to only present feedback when the user specifically requests it by pressing a button added to the interface by the plugin. When activated, the plugin identifies problems in the programmer's Python code. When a problem is located, a diagnostic message is shown to the user alongside a list of possible causes. Links to course material are embedded in the message to aid the user in locating relevant material.

Eclipse DAPS guides the user towards a solution with questions about the cause of the problem. These questions present a list of answers – typically related to the possible causes displayed alongside the diagnostic message. The user can interact with the guidance by selecting one of these answers by clicking on it, mimicing Socratic dialogue.

2.3.3 Block–C

While neither an ITS nor an IPT, Block–C [27] is a visual programming aid that is notable for the fact that it allows programs to be defined using visually distinct blocks. The tool is a standalone application, which the user installs and runs on their computer.

The blocks in the tool have distinct shapes, with their left edge indicating the type of the syntactic element (e.g. expression, statement, declaration) and the right edge

indicating the appropriate nested elements. A comprehensive catalogue simplifies the process of identifying the appropriate elements. The tool prevents the user from constructing a syntactically invalid program by only permitting blocks with matching edge shape.

The tool is reported to reduce time to complete tasks in a test setting and has received very positive user feedback. However, the author also reports that there was no difference in learning outcomes as measured by the course examination, where the students were not permitted to use the tool.

2.3.4 ANGELA

ANGELA claims to be an IPT that aims to aid learners by converting their programs to a three-dimensional representation utilizing augmented reality [28]. The tool converts the learner's Java program to a dynamic graph, and presents the graph to the user as a road network. The elements of the program are presented as features of that network, such as traffic signs and tunnels, with the program evaluation represented by a lorry driving through the network.

This approach allows the tool to be used in program complexity analysis – a complex program becomes large network – and concurrency visualization. The latter is achieved by adding additional networks and lorries, which allows the visualization of multithreading and synchronization methods, such as semaphores and fences.

The tool is developed as an augmented reality application. While this can be extremely beneficial to individuals that benefit from tangible models, it drastically limits the applicability of the tool for self-learning: the author observes that the institution has a limited number of augmented reality headsets and that almost no students have their own. Therefore, the tool is limited in applicability to small in-person teaching events.

2.3.5 Other tools

While editing the final revisions of this thesis, three tools that had eluded the initial survey were discovered: PVC.js [29], Python Tutor [30] and SeeC [31]. All of these tools are capable of visualizing the execution of C program code, with a focus on visualizing pointers. The first two tools can be integrated into a website – though Python Tutor also requires backend software – and all three tools are published with open source licenses. None of the tools accept programs with syntax errors and cannot therefore guide the user in resolving these.

Rather than interpreting the program step-by-step, these tools run the program and generate a trace of the program that is shown step-by-step to the user and which the user navigates. This places an upper bound on their capabilities as they are severely limited in terms of memory. In comparison with the three tools discussed above, these tools have limited feedback generation capabilities; for example, Python Tutor is only able to reproduce the diagnostics of `gcc`.

3 System architecture

In this chapter, we present the motivation behind this thesis and the high-level specification for a software suite that could improve learning outcomes for introductory-level programming students.

3.1 Motivation

In this sub-chapter, the motivation behind the development of the subject of this thesis are explored.

At the time of writing, Aalto University uses A+ Learning Management System (A+ LMS, A+) for distributing digital content over the web for a limited but growing number of courses in electrical engineering and computer sciences. The students access the platform with their browser. A+ allows the users to browse course materials at their leisure and to submit their exercise solutions, which are graded automatically. The automation is implemented with an elaborate system of containers, where a new container is provisioned for every submission. These submission containers (graders) are excellent from a security and system reliability standpoint, but have been observed to be fairly slow in terms of responsiveness. It can take several minutes for the grader to notify the submitter that their solution failed after a syntax error, which can lead to mundane troubleshooting taking a significant amount of time.

At Aalto, the introductory C programming course is mandatory for first-year electrical engineering students. There are hundreds of participants in each iteration of the course. The course is hosted on the A+ platform. As discussed in 2.2, the individual guidance from course staff that participants can receive is limited. To allow students to run simple programs without making a submission or installing a toolchain, a Jupyter-based approach for compiling and running C programs has been tested. However, this approach has three apparent limitations: the notebook takes quite long to start, there is no interaction with the user and it is not possible to inspect the state of the program during execution.

It has been observed that many students struggle with practicalities such as file systems and compiler toolchains well into the semester [17], [21]. This phenomenon appears to be independent of whether the student in question is a first-year student and their prior exposure to computers, with a minor exception for prior programming courses. It is possible to reduce the difficulty of installing toolchains by organizing guided exercise sessions. However, the variety in systems and devices make comprehensive installation strategies difficult to formulate.

There are particular problems that are difficult for students to resolve on their own. For example, students seem to be unable of resolving segmentation faults independently and declaring nontrivial pointers, arrays and functions appears to cause difficulties. Another point of confusion can be found in the implicit type conversions of C, i.e. pointer decay and promotions. The behaviour of the `sizeof` operator – which determines the size of an object – has also been observed to be challenging for students.

Many of these errors are something that could be resolved by applying static or

dynamic analysis. However, most if not all existing tools that are used professionally – such as Valgrind and `gdb` – do not generate feedback that is approachable to the students. On the other hand, a variety of tools has been developed specifically to assist novice programmers as discussed in 2.2.4. However, many of these tools expect the programmer to produce syntactically-correct programs and to understand said programs. Neither assumption holds with regards to novice programmers, as discussed in 2.2.1.

Although a handful of tools are reviewed in 2.3, the ones discovered during the initial survey are not applicable to the circumstances of an introductory course in C programming for three reasons. Firstly, none of the tools are in distribution. Second, the tools do not support the C programming language. The final reason is that none of the tools, even if available and modified to accommodate C, could be integrated into the online portal of the course. The other three tools, discussed in 2.3.5, were discovered only after the tool in this thesis was implemented. As such they could not be assessed comprehensively. However, all three tools are significantly limited in their capability to trace more complex programs.

3.2 Usage scenarios

In this sub-chapter, two usage scenarios are presented to exemplify behaviour expected from the software. Although fictional, both scenarios exemplify common errors in C programming. These scenarios were used to inform the architectural design of the system in the next sub-chapter.

3.2.1 Overflow

Another user is struggling with a programming exercise. Every time they submit the program for automatic evaluation, one of the tests fails with a wildly incorrect value. The user navigates to the course page to make use of the sandbox that the teacher mentioned in the first lecture. They remember the teacher mentioning that the software can help in locating errors that the compiler does not find. They copy their program into the editor and click the button labeled “run”.

The software indicates that it is busy for a second or two, before an orange underline appears to indicate a warning. Hovering over the underlined code reveals a tooltip message: the operation resulted in an overflow. The message also states that an overflow occurs if either the storage type or the operand types are not large enough to store the result. Links to glossary definitions are presented for the terms “overflow” and “type promotion”. In addition, the tooltip describes how many bytes or bits the result would have occupied.

3.2.2 Array boundary violation

The user is struggling again. Their program is crashing, seemingly at random. The only error presented is a cryptic `Segmentation fault` message. After changing everything that they can think of in their program code, the user searches for answers online. It does not take long before they discover a Q&A site for software developers.

There they learn that this message appears if they dereference an invalid pointer. However, they are unsure which of their pointers could be “invalid”.

Some of the answers mention something called “Valgrind”, which appears to be some kind of software that can pinpoint memory access errors and has to be installed on one’s computer. Another option mentioned is a “debugger”, but that’s apparently something that needs to be installed as well. The user tries to figure out how these are installed, but is mostly just confused. They decide to try that helpful software on the course website instead. It helped them with that tricky overflow earlier, so maybe it can help with this segmentation fault, too.

The user copies their program into the editor and clicks on “run”. The software does its thing for a short while, but eventually marks an expression with a red underline indicative of an error. The user moves the cursor over the marked expression and a message is displayed. The message states that the expression accesses the array past its limits with index 138, which is one step past the end of the array.

The message goes on to describe that this most commonly happens when a loop iterating over an array uses the less-or-equal-to operator `<=` instead of the less-than operator `<` or when a memory block is allocated for a string with no space for the null terminator. The message also prompts whether the user wants to see where the allocation was made.

3.3 Architecture

In this sub-chapter, the system architecture illustrated in figure 1 is discussed. The system consists of three primary components and a two types of modules. The primary components are the editor, the parser and the evaluator, each of which manages data. The two types of modules are responsible for analyzing and visualizing the data managed by the primary components.

As shown by the diagram, the editor accepts the user input. It may also provide a number of other useful features to enable the user to become more familiar with certain conventions, such as syntax highlighting, code folding and line numbering. Although not shown in the diagram, the editor is responsible for managing the textual representation of the program. The editor is also tasked with passing the input program to the parser for lexical, syntactic and semantic analysis. The reason for the editor to trigger this behaviour is because the editor is able to determine whether the program should be parsed again – i.e. when the program has changed.

The parser then generates a semantic representation of the program that can be used to interpret the program. The representation should also allow evaluation states to be associated with the corresponding locations in the program code. This representation is interpreted by the evaluator and analyzed by analyzers. The former is responsible for managing a program state according to directions from the user. To this end, it should be able to traverse the semantic representation generated by the parser.

The analyzers are used to extract meaningful information from the semantic representation and the program state. These can be variable types or interrelations between different segments of program code. They should transform the information

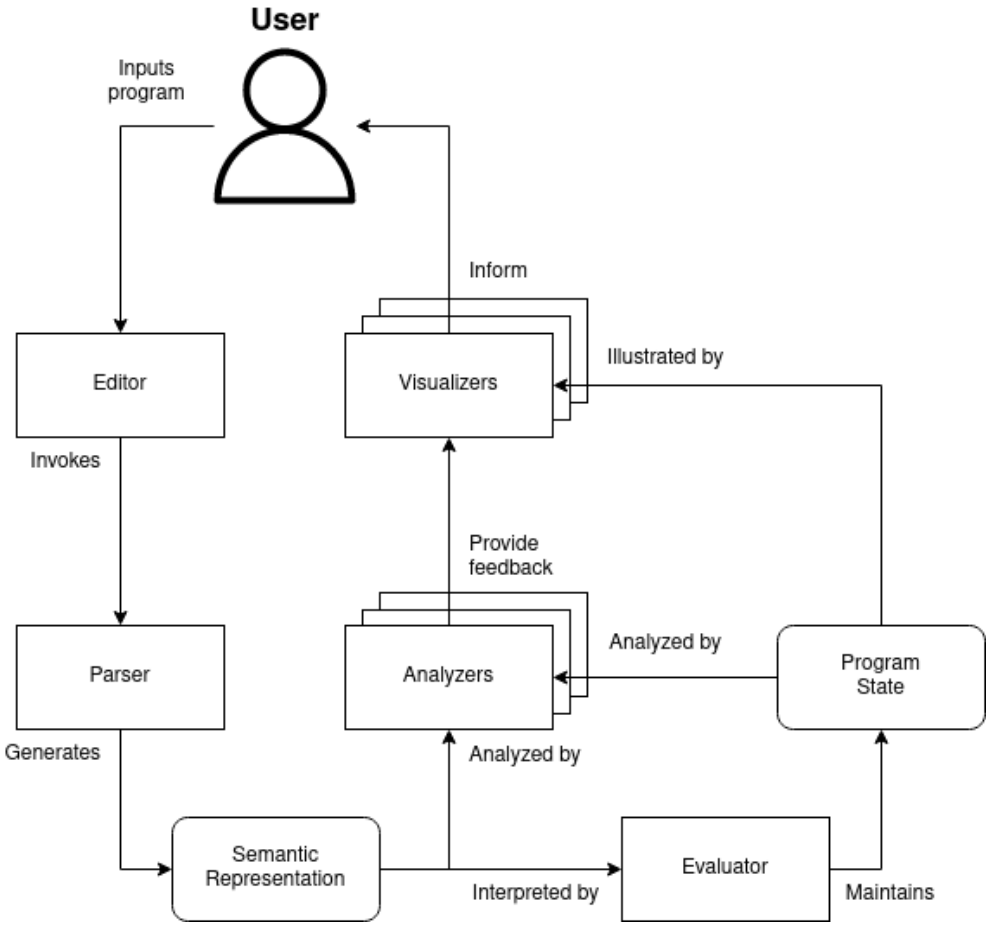

Figure 1: Conceptual system diagram

to produce feedback, which is digested by the visualizers that present the information to the user. It should be possible to disable particular types of visualizations and analysis independent of one another, which will allow the environment to be tailored to the needs of the user.

As implied above, the user must be able to interact with the evaluator. This is not indicated in figure 1 for the sake of clarity. For the user to be able to interact with the program state, the program state is also illustrated by the visualizers. This allows the user to see what actions their program is taking.

Most of the (visible) behaviour described in the scenarios in 3.2 falls into the domain of the analyzers and visualizers. The analyzers are responsible for locating and indicating the issues, while the visualizers are responsible for generating the user interface components for these indications. However, the parser and the evaluator are also needed to generate and manage the necessary state.

4 Implementation

This chapter describes CVisualizer, an implementation of the software specified in the previous chapter. A source code distribution of the tool is available under the MIT License [32]. This chapter consists of four subchapters, the first detailing the architectural core of CVisualizer, the second the syntax analyzer, the third the evaluation scheme and the fourth the interactive capabilities of the system.

4.1 System architecture

To analyze the student code and generate meaningful feedback, a parser was needed. The function of this parser is to translate C programs to an AST for analysis as described in 2.1.1. Although there exist parsers for the C programming languages written in JavaScript, it is not apparent to what extent they support the partial parsing CVisualizer would require. It therefore appeared necesssary to implement a parser that could be tailored to the needs of the system. The details of this parser are discussed in 4.2.

4.1.1 CodeMirror

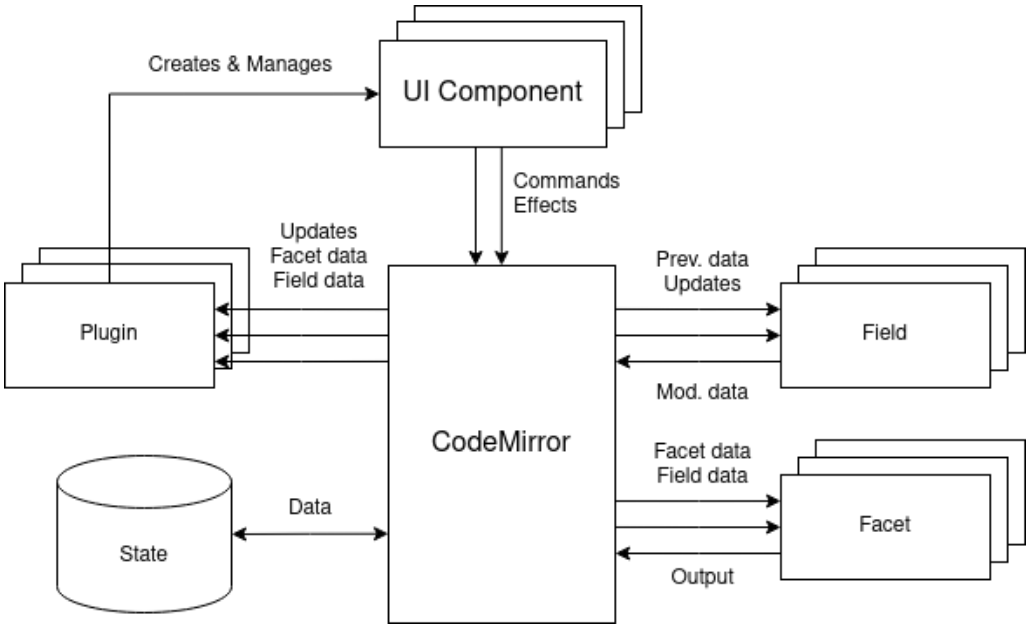

Figure 2: The CodeMirror data model

To deploy on the web, we chose CodeMirror 6 [33] as the text editor. This choice was made as CodeMirror is a browser-based text editor that supports the myriad of features that IDE text editors have, such as syntax highlighting and line numbering. Other browser-based editors also exist – such as browserpad [34] – but CodeMirror was the only candidate that had most of the common text editing functions implemented and the required support for extendability. It is this extendability that makes

CodeMirror well-suited to be the core of CVisualizer. This also allows features of CVisualizer to be toggled to reduce the complexity of the software depending on the surrounding context.

CodeMirror has been designed with a data-oriented paradigm. It defines atomic, abstract constructs that interface with the editor and its internal data in different ways. Namely, it completely separates the notions of inputs (transactions), text content (document), application data (fields and facets), storage of data (state) and the presentation of the state (plugins), depicted in figure 2. It should be noted that the components do not store their own data under this model, but rather leave the management of it to the CodeMirror instance, which passes the "current" state to the components upon invocation. The key benefits of this arrangement is that extensions can be shared between multiple editors, can be disabled without recreating the editor and operations such as "undo" and "redo" can be performed on arbitrary data from the extensions.

Under CodeMirror, input to the editor is dispatched as transactions, which contain the changes and effects to apply to the document and its presentation. Transactions are generated as a result of user input (e.g. inserting or removing text) but they can also be dispatched by program code. The latter kind of transaction will often include one or more *effects* that are used to pass data – for example, information on user events – to the editor components.

As the components themselves do not store data, a *field* is used to describe the lifetime events of some value in the editor. Fields are updated periodically after transactions occur and recompute a new value from their previous value and the changes described by the transaction. This new value is then stored by the editor. *Plugins* resemble fields, but are additionally notified of user interaction events (e.g. hovering and clicks) directly and can modify the presentation of the document. Lastly, a *facet* is a component that describes how a set of values can be mapped to a new value. The inputs of a facet can be values of other components or the contents of the document. A facet differs from a field in that a facet only recomputes its value if its inputs change. Vice versa, the facet does nothing if the inputs do not change.

4.1.2 Other considerations

Given the requirement of a web-based solution and time constraints, there was realistically only two choices for languages when developing the application: TypeScript or JavaScript. As the former is transpiled to the latter and because the chosen parser generator does not generate TypeScript compatible parsers, JavaScript was chosen as the programming language. Other language options would also have been available through the use of WebAssembly, to which C/C++ and numerous other languages can be compiled to and which can be distributed as binaries for a virtual machine that can be found embedded in most modern browsers. However, the interface between JavaScript and WebAssembly would have introduced additional challenges in the implementation.

4.2 Parser

Given that CVisualizer is required to analyze the user's program, it is necessary to utilize a parser to extract the semantic structure of the program from the source code. In this section, the factors considered and design behind this component are discussed.

4.2.1 Framework

As hand-written parsers are difficult to maintain, the decision was made to leverage an existing parsing framework. The apparent choice would have been to use GNU `bison` [35] parser generator, which is widely used for the purpose. However, there is no JavaScript target for `bison`, so a choice had to be made between implementing said target – which is a substantial task in and of itself – and choosing a different generator. Limiting the search to parser generators for ones capable of producing a parser suitable for browsers yielded PEG.js[36] and nearley.js[37]. Chevrotain[38] – an embedded DSL used to define $LL(k)$ parsers – and ANTLR.js [39] were also looked into. However, ANTLR.js was disregarded as the documentation for the JavaScript bindings was not apparent and other options existed.

In addition to the above, CodeMirror has integrated support for parsers. However, said parsers are not suitable for processing source files for analysis. Lezer, the parser generator used to generate parsers compatible with CodeMirror, is optimized for parsing in the context of text editing, where the primary requirement is that the parse tree can be rapidly updated after the input changes and is therefore optimized for subsequent updates at the expense of not being able to support parsing state. For example, it is impossible to make use of a symbol table, which is crucial to parsing the C programming language. Additionally, C is not one of the built-in languages, so the built-in C++ syntax highlighting is used instead.

Of the initial candidates, Chevrotain would have been the preferred choice with better measured performance than the other two options and a claim of good error recovery capabilities. However, it was uncertain whether the resulting $LL(k)$ parser would be sufficiently powerful to parse C as this class of parsers do not support left recursion [11]. I.e. rules of the form $A \rightarrow A\beta$ would have needed to be rewritten in a less idiomatic manner. Additionally, the syntax by which the grammars are defined would have made it difficult to change frameworks later. Both of the generators – PEG.js and nearley.js – produce parsers that are more powerful than the $LALR(1)$ traditionally used to parse C.

Nearley.js parsers are defined in a dialect of BNF that is commonly called extended Backus-Naur form (EBNF) and generates Earley parsers, which are capable of parsing all context-free grammars, while PEG.js consumes parsing expression grammars (PEG) that are distinct but resemblant of EBNF and CFG. PEG would have introduced additional complexity in describing the grammar while being less powerful and efficient, so nearley.js was chosen as the parser generator. The Earley algorithm exhaustively explores all possible interpretations of the input program – when an input token is consumed, it can cause states to be added, removed or advanced in the current parse state. Similar to most LR parsers, nearley.js supports reduction

actions that are executed whenever a rule is completed. These actions are defined as functions.

4.2.2 Implementation capabilities

The implemented parser accepts programs written in a subset of the 1999 revision of the C programming language (C99) [22]. The most notable omitted features are the preprocessor and bitfields. `#include` directives are processed separately with slightly different semantics that allow JavaScript-defined libraries. Many of the less-common features of the language were also omitted, such as the so-called "Kernighan & Ritchie" -style function declarations and digraphs. Most of these latter features have caused modern compilers to emit warnings for several years and their removal from the standard has either been proposed or confirmed. Some notable inclusions from C99 are variable-length arrays, initializers and compound literals. It was decided that focus should be on a core set of broadly-applicable language features and approachable diagnostics rather than a standard-compliant implementation.

At a late stage in the implementation of CVisualizer, it was discovered that nearley.js is not reentrant. In particular, if one defines a symbol table within the parser, that symbol table would be shared with subsequent or parallel invocations of parsers with the same grammar. Neither the parser nor the grammar present an external interface that could be used to reset the symbol table. There is also no interface by which a symbol table could be passed into the parser. As the grammar itself is stored as a JavaScript module, it is subject to caching. Caching, in combination with the closures of the reduction actions, makes it practically impossible to duplicate grammars.

To circumvent this, the lexer was modified to store the symbol table as the lexer state could be accessed from outside the parser. However, different instances of the parser would also share a reference to the same lexer, as the lexer is instantiated from within the grammar. Ultimately, the possibility of multiple instances of CVisualizer on one page was discarded.

Another discovery was that nearley.js does not support graceful error recovery, unlike many other parser generators. In particular, the programmer must feed the input line-by-line to the parser in order to be able to locate more than one syntax error. Nearley.js lacks the capability of constructing an AST where the erroneous subtrees are annotated and ignored during syntactic analysis. This makes it difficult to generate meaningful diagnostics to the user for syntactically invalid programs.

The parser constructs an AST where different language constructs are represented by instances of a class. Each class also defines the semantic meaning of that language construct. The evaluation process of these nodes is described at length in the next subchapter.

4.3 Evaluation engine

This subchapter describes the data structures and algorithms used to implement the evaluation engine, which interprets the semantic representation generated by

the parser. The subchapter consists of two sections. The first section describes the interactions between the semantic representation and the evaluation engine. The second section illustrates the data structures used to model the values and variables used in evaluation, as well as the manner in which these are stored and accessed.

4.3.1 Evaluation process

As JavaScript is single-threaded, it was necessary to use APIs that allows the evaluation of the user's program to be interleaved with the execution of other code on the page. Simultaneously, it was necessary to allow the program flow to be controlled externally. There are two different approaches to achieve non-blocking behaviour in JavaScript: callbacks and promises. Both methods utilize the event queue to achieve asynchronous behaviour. We will discuss these next.

Callbacks are, effectively, plain functions. However, in JavaScript, functions can have "closures" that allow them to access values and variables from the context that they were defined in. This allows these functions to have external state that they can modify. In the callback pattern, a call to one such function is added to the event queue when the previous function completes. By subdividing complex operations, this pattern allows the time-consuming operations to be performed such that they do not block the event queue. The main drawback with callbacks is that there is no convenient way to pass errors "backwards" to the previous function.

The second approach is to use promises, which are a built-in primitive type. Promises extend the callback pattern with the object-oriented paradigm – instead of the caller passing a callback to the callee, the callee returns a promise object. A promise is *resolved* when the callee completes its operation and *rejected* if an error occurs – in either case, the promise is considered *settled*. When a promise settles, there are two ways to execute further code – the first is to invoke a method of the promise with a callback and the second involves the use of the `await` keyword. In the latter case, the execution of the caller function is suspended until the promise settles. If the promise was resolved, `await` expression extracts the value it resolved to. If it was rejected, then an exception is raised instead.

The `await` keyword can – generally speaking – only be used within a function defined with the `async` keyword. When a function is defined using the `async` keyword, the function implicitly returns a promise that is resolved when the function definition "returns" and rejected if an exception is raised. The combination of these two keywords allows non-blocking procedures to be defined in a manner that closely resembles normal sequential code.

Figure 3 illustrates the behaviour of promises as they are used in CVisualizer. In the figure, the node being evaluated requests the environment to evaluate a sub-node. The environment returns a promise, which evaluates the sub-node or some part of it when the user issues a step effect by clicking on a control flow button. The exact portion evaluated depends on the button clicked.

In the figure, the sub-node is first stepped into (e.g. to the first expression in an expression list) and then the nested sub-node (`subNode2`) is completely evaluated with "step over". The evaluation of the nested sub-node is included in the evaluation

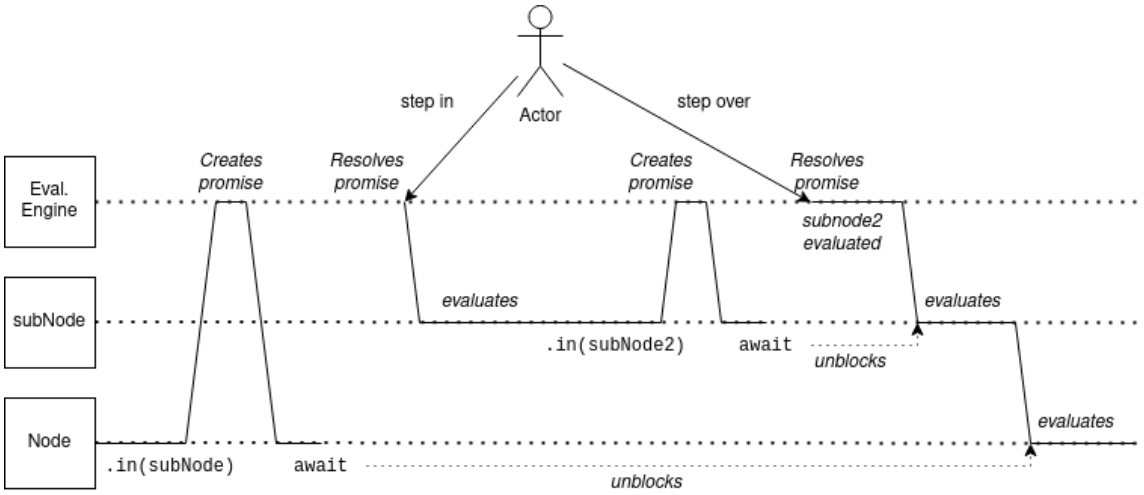

Figure 3: Promises in the evaluation process

engine thread to simplify the diagram. Once the nested sub-node is completely evaluated, the corresponding promise resolves and the execution resumes at the location of the `await` expression in the first sub-node. Once this sub-node returns, the first promise is resolved and the first node can continue execution.

4.3.2 Memory model

The evaluation engine also stores the variables of the program as tuples of name, address, type and value. A value, on the other hand, is described as a tuple of type and value; variables are a subclass of the class of value. In this model, the type is an instance of a class that is responsible for defining the conversions from other value types to that corresponding to the type — for example, an integer type converts numeric type values to integer types. Variables are stored in two tables; one which allows access to them by address and the other by name.

An instance of both tables exists for every frame at any given state; when the execution leaves the frame, the corresponding tables are destroyed. This allows a very convenient model for separating scopes and visualizing the list of variables at any state. The frames have additional metadata attached to them, such as the location where they were created from as well as the name of the function that they correspond to.

However, this approach has an apparent limitation; it is not amicable to implementation of arrays or structs. As each name and address correspond to an instance of a variable, there is no convenient method to indicate a reference to a member of array. Furthermore, it is likewise difficult to identify whether a given access through a pointer is within the boundaries of the array. Unfortunately, the time that was reserved for resolving these issues was spent on correcting the lack of a symbol table instead.

4.4 User interface

The user interface consists of the visual components presented to the user, which both inform the user and allow them to interact with the other components of the system. This subchapter consists of four sections, each of which details one of the interface components: editor, syntax tree visualizers, evaluation controls and memory visualizer.

4.4.1 Editor

As discussed above, CodeMirror forms the core of the implementation. CodeMirror provides the editor and the capability for extensions. For CVisualizer, the editor was designed to follow industry-standard tools in terms of look-and-feel. The intent behind this design choice is to allow the users to familiarize themselves with the tools necessary for professional programming. At the same time, the scope of functionality was minimized to avoid overwhelming the user.

In figure 4, the configured editor without any CVisualizer extensions is depicted. From the figure, it is possible to observe syntax highlighting, line numbering and active line highlighting. In addition to these features the editor supports bracket matching and undo/redo, as well as many useful shortcuts for selecting and navigating the program. The shortcuts were enabled to limit the frustration potentially caused to more experienced students.

```
1  #include <stdbool.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int main() {
7      int a = 12;
8      // struct cc ff = {.aa = 1};
9      //printf("Hi there! %d - %s\r\n", pp[12], (char*)pp);
10     // printf("Hi there! %d \r\n", *pp2);
11
12     //print_something();
13
14     char* line = NULL;
15
16     printf("\n");
17     while (true) {
18         line = NULL;
19         lineSize = 0;
20         // printf("Read %d chars from stdin %d \n", readSize);
21
```

Figure 4: The editor view

4.4.2 Syntax tree visualizer

The first visualization component implemented was a simple visualizer that displays a *leafish* – a term that is defined later in this subchapter – subtree of the AST for a given position. In figure 5, one configuration of the visualizer is shown. In the figure, the leafish subtree for a function call is displayed as the user hovers over the relevant part of code. As can be observed from the figure, some leaf nodes – e.g. operators and punctuation – are used to label subtrees. This behaviour is expected to make it easier for a novice to understand the tree structure.

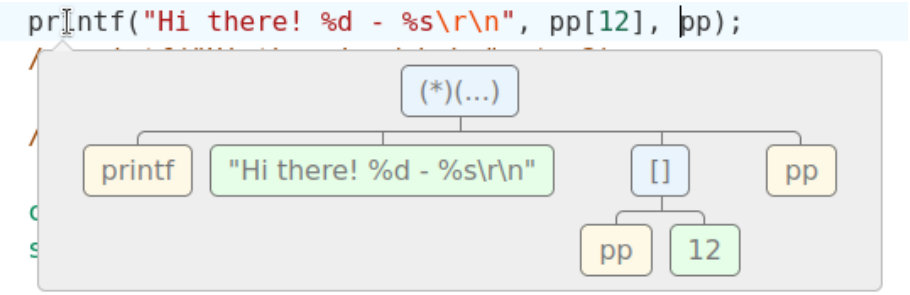

Figure 5: Visualization of a function call

In figure 6, the AST for an assignment is displayed using the same configuration as before. As one of the most common points of confusion for novices appears to be operator precedence, it was hoped that these visualizations would allow them to explore their programs from a different point of view. As in this implementation type analysis is only performed when an expression is executed, it is not possible to show e.g. type promotions or result types in these visualizations.

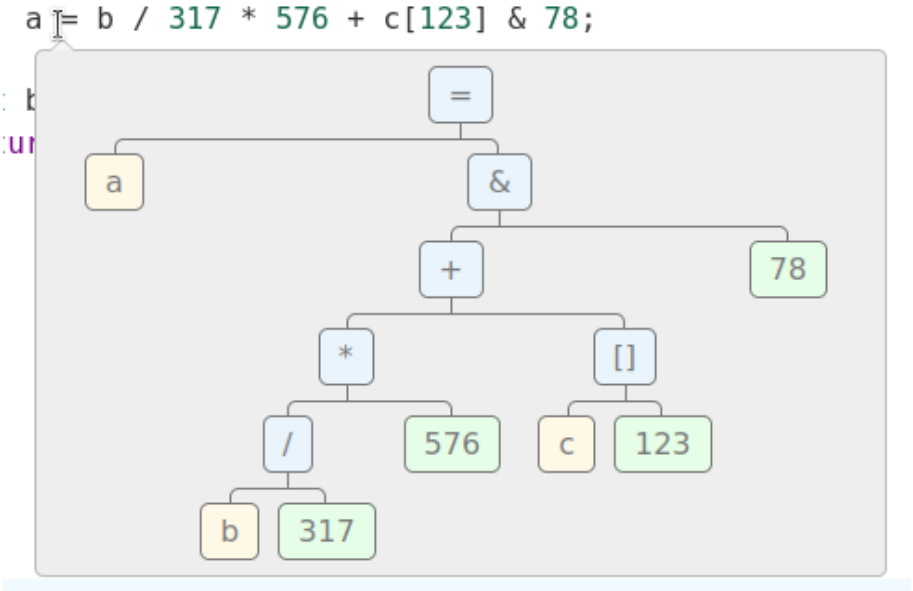

Figure 6: Visualization of a complex assignment expression

The behaviour of the visualizer is configurable in several ways; for example, it is possible to render the AST in an external context and to configure the depth of the visualization. In the shown configuration, different types of nodes are shown in different colors: identifiers in one, literals in another and operators in a third. This particular configuration only renders expressions. Other nodes were excluded as rendering the tree may cause the user to make incorrect assumptions about the language construct.

Above, the term *leafish* was used. This term is used here to refer to a node that is a "close" ancestor of a leaf node in the AST. As a complete AST for even the simplest of programs can consist of thousands of nodes, it is important to select a relevant subtree around a given location. This subtree should provide a meaningful insight into the way the computer understands the program at that location. Hence, algorithm 1 was designed for this purpose. In CVisualizer, the set of *leafish* nodes is defined to be the set of expression nodes for the reasons discussed above.

Algorithm 1 *Leafish* tree selection algorithm

L : the set of *leafish* nodes
R : the AST root node
P : the position of interest

$N \leftarrow R$
while $N \neq \bot \wedge N \notin L$ **do** ▷ *Iterating over L is infeasible*
    $N' \leftarrow N$
    $N \leftarrow \bot$
    for all $C : N'$ **do**
        if $\|P \cap C\| \neq 0$ **then** $N \leftarrow C$

N is either $\bot$ or a *leafish* subtree.

In algorithm 1, it is remarked that iterating over the set of leafish nodes L is infeasible. It is not viable to keep a record of the nonterminals of interest in the AST as the parser is not capable of such stateful behaviour as described in 4.2.2. However, as long as it is possible to inspect a node N and determine whether $N \in L$, it is also possible to traverse the AST to locate all members of L.

Given that only the *leafish* subtree containing position P is of interest, it is possible to search only subtrees containing P. As the children of each node in the AST are ordered by their positions and no siblings can overlap, it is trivial to perform a search in $O(\log n)$ with respect to the input size. Ergo, the algorithm uses a breadth-first search of the AST for nodes containing P and descends until a subtree $N \in L$ is found. When compared with recording the set L, this algorithm also has the benefit that the set L can be chosen by passing $N \mapsto N \in L$ as an argument.

4.4.3 Evaluation controls

In figure 7, the user's view of the evaluation environment is presented. I.e. the figure shows the editor with the evaluation components enabled. To name the additional

components: at the top is a toolbar with controls, on the right there is the memory visualizer and in the program code there is an evaluation indicator. Additionally, there is an indicator alongside the line numbers on the left to assist with locating the evaluation indicator. The indicator and the controls are discussed here, while the memory visualizer is discussed in 4.4.4.

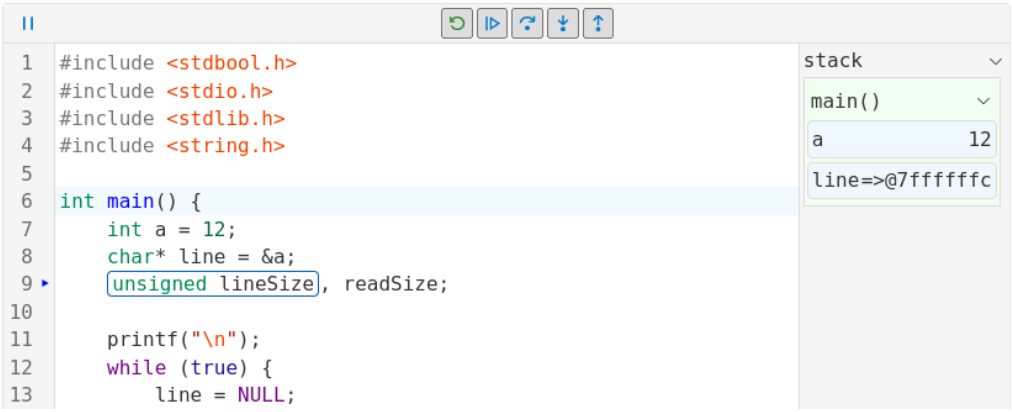

Figure 7: Overview of the evaluation environment

The toolbar has five buttons and one indicator; the indicator is located in the top-left corner of the editor and displays the current status of the program. Four states can be indicated: waiting, paused, running and error. The distinction between the "waiting" and "paused" states is that the "waiting" state indicates that the program has not yet been started and therefore stepping is not permitted. The icons were selected from the Visual Studio Code icon database [40] to match the look-and-feel of the development environment used for the course.

In addition to the status indicator, the five buttons used to control program execution were briefly mentioned above. The buttons are shown in more detail in figure 8, alongside a tooltip. From left-to-right, the five buttons are: restart, run, step over, step in and step out. These behave in the manner that can be expected from a debugger. When the program is in the "waiting" state, the restart button is replaced with a "start" button that is functionally identical but is clearer to the user. Each of the buttons displays a descriptive tooltip when the user hovers over it, as shown in the figure. The descriptions allow the user to explore the interface without having to reference a manual.

4.4.4 Memory Visualizer

The memory visualizer is a component of the evaluation environment and allows the user to browse the program state. The visualizer shows two different memory segments: the *stack* and the `.data` segment. It is possible to define an arbitrary number of segments, however. All values declared within functions – including parameters – are displayed in the stack, while global variables are shown in the `.data` segment.

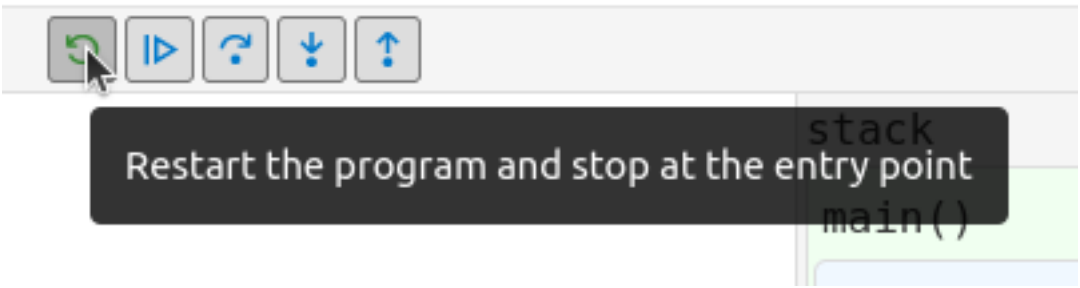

Figure 8: An example hover tooltip for the evaluation controls

Figure 9 illustrates the memory visualizer. In the visualization, every scope – e.g. compound statement and function call – is represented with a separate visualization frame. Where the namespace of prior frames is hidden, i.e. between call frames, the frames are visually separated as well. Although shown "open" here, the prior frames are folded when a function is invoked to signal to the user that the variables have become hidden. The user can click on the frames to toggle the visibility of their contents at any time.

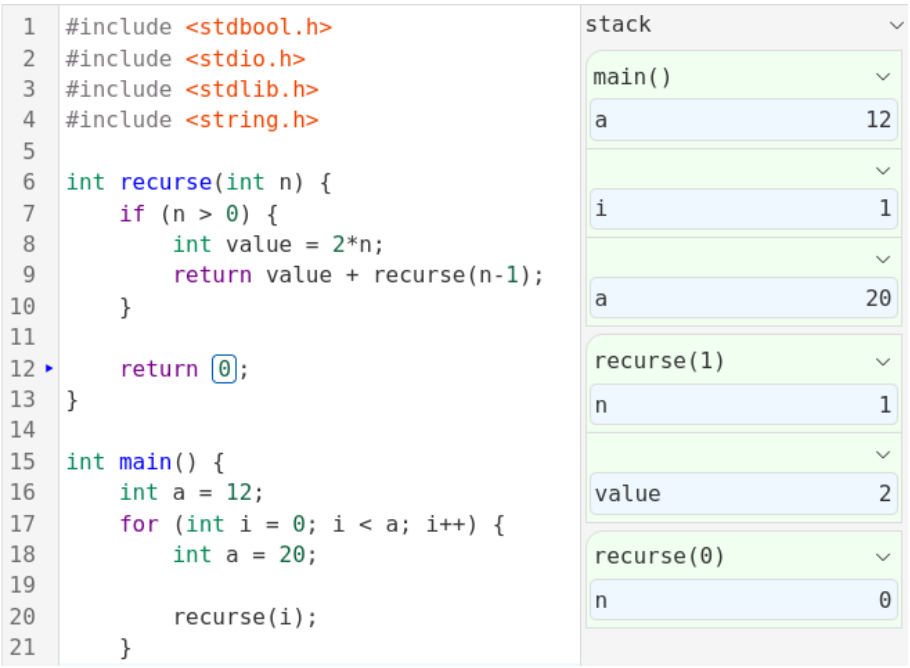

Figure 9: Memory visualization for multiple frames

It was chosen to omit type information and addresses from the default view to minimize visual clutter; these are less important than the values stored. Instead, the address and the type of the object are shown as a tooltip when the element is hovered over with the cursor. In the current version, the type is a simple reiteration.

5 Conclusions

Although there are promising components to it, CVisualizer is not suitable for deployment at this moment. Towards that goal, there are a number of improvements to be made upon the tool presented in this thesis. First, the lexical analyzer should be separated from the syntax analyzer. Second, the parser generator must be replaced. Third, the evaluation engine must be reworked to permit definition of arrays and structs.

The purpose of separating the lexical analyzer is to reduce the number of redundant parses and to allow token-level analysis even in the case where the parser fails completely. In particular, if the sequence of tokens produced by the lexical analyzer does not change, then the semantic representation also does not need to be updated. This separation would also allow an optimization by the way of only re-analyzing the token stream where the user makes changes. The change would also allow certain desirable editing features, such as symbol renaming, to be implemented with less effort.

In 2.3.5, three tools – are discussed. At least two of these – PVC.js and Python Tutor – appear to fulfill the criteria for the needs of the Aalto introductory C programming course discussed in 3.1. However, as discussed in 2.3.5, the existence of these tools was discovered only after the subject of this thesis had already been implemented. It can also be said that while these tools appear superficially similar to CVisualizer, there are also fundamental differences in interactivity and feedback generation. Unlike CVisualizer, these tools are also not extendable.

Of the above, PVC.js utilizes a parser generated with ANTLR4. For this thesis, ANTLR4 was disregarded for the lack of documentation for its JavaScript bindings as discussed in 4.2.1. However, as it has been used in a similar application, its applicability should be reconsidered for CVisualizer or other future works. At the same time, the implementation of a $LALR(1)$ parser generator should also be considered. Regardless of which implementation method is chosen, the developer should ensure that the generated parser is capable of error recovery, maintaining symbol tables and re-entrant behaviour.

Lastly, the evaluation engine should be reworked to utilize a different method for storing values and the associated types; as it is, the engine is not suitable for analyzing array boundaries and values more complex than a single-dimensional array or a singular struct. At the time of writing, the apparent way to achieve this would be to store the primitive values (e.g. numeric or pointer values) separately from the composite types and then inspect these upon access.

References

[1] N. O. Ezeamuzie, "Project-first approach to programming in K–12: Tracking the development of novice programmers in technology-deprived environments," *Education and Information Technologies*, vol. 28, no. 1, pp. 407–437, 2022, ISSN: 1360-2357. DOI: 10.1007/s10639-022-11180-8.

[2] C. Watson and F. W. Li, "Failure rates in introductory programming reurldate," in *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ser. ITiCSE '14, Uppsala, Sweden: Association for Computing Machinery, 2014, pp. 39–44, ISBN: 9781450328333. DOI: 10.1145/2591708.2591749.

[3] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003. DOI: 10.1076/csed.13.2.137.14200.

[4] L. E. Winslow, "Programming pedagogy–a psychological overview," *SIGCSE Bull.*, vol. 28, no. 3, pp. 17–22, 1996, ISSN: 0097-8418. DOI: 10.1145/234867.234872.

[5] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 83–137, 2005, ISSN: 0360-0300. DOI: 10.1145/1089733.1089734.

[6] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers," *SIGCSE Bull.*, vol. 37, no. 3, pp. 14–18, 2005, ISSN: 0097-8418. DOI: 10.1145/1151954.1067453.

[7] A. Pears, S. Seidman, L. Malmi, *et al.*, "A survey of literature on the teaching of introductory programming," *SIGCSE Bull.*, vol. 39, no. 4, pp. 204–223, 2007, ISSN: 0097-8418. DOI: 10.1145/1345375.1345441.

[8] D. N. Perkins and F. Martin, "Fragile knowledge and neglected strategies in novice programmers," in *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, Washington, D.C., USA: Ablex Publishing Corp., 1986, pp. 213–229, ISBN: 089391388X.

[9] R. Lister, E. S. Adams, S. Fitzgerald, *et al.*, "A multi-national study of reading and tracing skills in novice programmers," *SIGCSE Bull.*, vol. 36, no. 4, pp. 119–150, 2004, ISSN: 0097-8418. DOI: 10.1145/1041624.1041673.

[10] T. Crow, A. Luxton-Reilly, and B. Wuensche, "Intelligent tutoring systems for programming education: A systematic review," in *Proceedings of the 20th Australasian Computing Education Conference*, ser. ACE '18, Brisbane, Queensland, Australia: Association for Computing Machinery, 2018, pp. 53–62, ISBN: 9781450363402. DOI: 10.1145/3160489.3160492.

[11] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques, and Tools* (Addison-Wesley series in computer science and information processing). Addison-Wesley Publishing Company, 1986, ISBN: 9780201100884.

[12] N. Chomsky, "Three models for the description of language," *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 113–124, 1956. DOI: 10.1109/TIT.1956.1056813.

[13] N. Chomsky, "On certain formal properties of grammars," *Information and Control*, vol. 2, no. 2, pp. 137–167, 1959, ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(59)90362-6.

[14] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs* (MIT Electrical Engineering and Computer Science). The MIT Press, 1996, 688 pp., ISBN: 9780262510875.

[15] P. E. Dickson, N. C. C. Brown, and B. A. Becker, "Engage against the machine: Rise of the notional machines as effective pedagogical devices," in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '20, Trondheim, Norway: Association for Computing Machinery, 2020, pp. 159–165, ISBN: 9781450368742. DOI: 10.1145/3341525.3387404.

[16] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 572–583, ISBN: 9781450356381. DOI: 10.1145/3180155.3180175.

[17] S. N. Freund and E. S. Roberts, "Thetis: An ANSI C programming environment designed for introductory use," in *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '96, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 300–304, ISBN: 089791757X. DOI: 10.1145/236452.236560.

[18] F. T. Willmore, "Debugging with gdb," *Introduction to Scientific and Technical Computing*, pp. 85–96, 2016.

[19] M. Bower and K. Falkner, "Computational thinking, the notional machine, pre-service teachers, and research opportunities," in *Proceedings of the 17th Australasian Computing Education Conference (ACE 2015)*, Australian Computer Society, vol. 27, 2015, p. 30.

[20] B. Du Boulay, T. O'Shea, and J. Monk, "The black box inside the glass box: Presenting computing concepts to novices," *International Journal of man-machine studies*, vol. 14, no. 3, pp. 237–249, 1981.

[21] D. Budny, L. Lund, J. Vipperman, and J. Patzer, "Four steps to teaching C programming," in *32nd Annual Frontiers in Education*, vol. 2, 2002, F1G–F1G. DOI: 10.1109/FIE.2002.1158140.

[22] International Organization for Standards, "Programming languages — C," ISO/IEC, ISO/IEC 9899:1999, 1999.

[23] D. M. Ritchie, S. C. Johnson, M. Lesk, B. Kernighan, *et al.*, "The C programming language," *Bell Sys. Tech. J*, vol. 57, no. 6, pp. 1991–2019, 1978.

[24] L. Mannila, M. Peltomäki, and T. Salakoski, "What about a simple language? Analyzing the difficulties in learning to program," *Computer Science Education*, vol. 16, no. 3, pp. 211–227, 2006. DOI: 10.1080/08993400600912384.

[25] K. VanLehn, "The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems," *Educational psychologist*, vol. 46, no. 4, pp. 197–221, 2011.

[26] S. Kivirinta, *Reducing persisting cognitive dissonance and Computer Science 1 drop-out rates using visual debugger aid*, 2015.

[27] C. Kyfonidis, N. Moumoutzis, and S. Christodoulakis, "Block-C: A block-based programming teaching tool to facilitate introductory C programming courses," in *2017 IEEE Global Engineering Education Conference (EDUCON)*, 2017, pp. 570–579. DOI: 10.1109/EDUCON.2017.7942903.

[28] S. Schez-Sobrino, C. Gmez-Portes, D. Vallejo, C. Glez-Morcillo, and M. Á. Redondo, "An intelligent tutoring system to facilitate the learning of programming through the usage of dynamic graphic visualizations," *Applied Sciences*, vol. 10, no. 4, 2020, ISSN: 2076-3417. DOI: 10.3390/app10041518.

[29] R. Ishizue, K. Sakamoto, H. Washizaki, and Y. Fukazawa, "Pvc.js: Visualizing C programs on web browsers for novices," *Heliyon*, vol. 6, no. 4, 2020.

[30] P. J. Guo, "Online Python tutor: Embeddable web-based program visualization for CS education," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13, Denver, Colorado, USA: Association for Computing Machinery, 2013, pp. 579–584, ISBN: 9781450318686. DOI: 10.1145/2445196.2445368.

[31] M. Heinsen Egan and C. McDonald, "An evaluation of SeeC: A tool designed to assist novice C programmers with program understanding and debugging," *Computer Science Education*, vol. 31, no. 3, pp. 340–373, 2021.

[32] V.-M. Rantanen. "CVisualizer Repository." (2022–2023), [Online]. Available: https://version.aalto.fi/gitlab/rantanv2/cvisualizer/ (visited on 2023-07-31).

[33] M. Haverbeke. "CodeMirror." (2007–2023), [Online]. Available: https://codemirror.net/ (visited on 2023-07-28).

[34] W. Pimenta. "Browserpad." (2015–2023), [Online]. Available: https://github.com/browserpad/browserpad (visited on 2023-07-28).

[35] Free Software Foundation, Inc. "GNU Bison." (1988–2021), [Online]. Available: https://www.gnu.org/software/bison/manual/bison.html (visited on 2023-07-28).

[36] D. Majda and F.-z. Ryuu. "PEG.js." (2010), [Online]. Available: https://github.com/pegjs/pegjs (visited on 2023-07-28).

[37] K. Chandra and T. Radvan. "nearley: A parsing toolkit for JavaScript." (2014), [Online]. Available: https://github.com/kach/nearley (visited on 2023-07-28).

[38] S. Soel. “Chevrotain.” (2018–2023), [Online]. Available: https://chevrotain.io (visited on 2023-07-28).

[39] T. Parr. “ANTLR.” (1989–2023), [Online]. Available: https://www.antlr.org/ (visited on 2023-07-28).

[40] Microsoft. “Visual Studio Code - Icons.” (2019), [Online]. Available: https://github.com/microsoft/vscode-icons (visited on 2023-07-28).