Extending a browser C++ simulator

Albert Segarra Roca

Advisor: Jordi Petit

Specialization in Computing

Facultat d’Informàtica de Barcelona • FIB
Universitat Politècnica de Catalunya • UPC BarcelonaTech

This dissertation is submitted for the bachelor degree of

Informatics Engineering

June 2017
Abstract

Getting started in computer programming involves using a number of tools such as editors, compilers, debuggers, and terminals. Learning how to use them can be challenging for programming beginners, and installing them may be demanding for certain organizations such as high schools. C– is a project that aims to bridge this gap by providing an educational C++ development environment that runs in a web browser. The goal of this thesis is to extend and improve the current implementation of C–. This has been done by enlarging the supported C++ language subset, increasing the compiler maintainability and extensibility, providing better feedback and improving the website interface. The performance of C– has been compared with other languages and compilers, showing that it is fast enough for educational use and is faster than previous versions of C–.

Resum

Iniciar-se en la programació implica l’ús d’eines com editors, compiladors, depuradors i terminals. Aprendre a usar-los pot ser desafiànt per programadors principiants, i la seva instal·lació pot ser una tasca exigent per certes organizations com ara instituts o escoles. C– és un projecte que pretén tancar aquesta bretxa tot proporcionant un entorn de programació pel llenguatge C++ que s’executa al navegador. L’objectiu d’aquest projecte de fi de carrera és ampliar i millorar l’implementació actual de C–. Això s’ha aconseguit ampliant el subconjunt del llenguatge C++ suportat, augmentant la mantenibilitat i extensibilitat del compilador, proporcionant millor feedback i millorant la interfície web. El rendiment de C– s’ha comparat amb altres llenguatges i compiladors, el que ha demostrat que és prou ràpid com per ser usat per propòsits educatius i que és més ràpid que versions anteriors del projecte.
## CONTENTS

### 1. Context and Scope

1.1. Introduction and Context .............................................. 6  
1.1.1. Problem .............................................................. 6  
1.1.2. Solution ............................................................. 7  
1.1.3. Stakeholders ....................................................... 8  
1.1.4. State of the art and possible alternatives ................. 9  
1.2. Objectives .............................................................. 10  
1.3. Scope ......................................................................... 11  
1.3.1. Initial scope .......................................................... 11  
1.3.2. Final scope ........................................................... 12  
1.3.3. Requirements ....................................................... 13  
1.3.4. Possible obstacles ................................................. 13  

### 1. Planning

2. Project planning ................................................................. 15  
2.1. Methodology .................................................................. 15  
2.1.1. Development methodology ...................................... 15  
2.1.2. Tracking methods .................................................... 16  
2.1.3. Validation methods .................................................. 16  
2.2. Technology stack ......................................................... 16  
2.3. Time planning ............................................................... 17  
2.3.1. Tasks ..................................................................... 17  
2.3.2. Gantt chart ............................................................. 21  
2.3.3. Potential deviations and alternative plans ............... 22  
2.3.4. Plan modifications ................................................... 22  
2.4. Economic management ................................................... 25  
2.4.1. Cost analysis .......................................................... 25  
2.4.2. Control Management .............................................. 28  
2.4.3. Cost modifications .................................................. 29  
2.5. Sustainability analysis .................................................... 30
CHAPTER

1

CONTEXT AND SCOPE

1.1. Introduction and Context

C++ is a very popular programming language both in industry and academic research, mainly thanks to its robustness, scalability and efficiency, which make it specially suitable for large scale performance-critical applications. For this reason, it is also a typical choice as an introductory or post-introductory programming language in universities and colleges, because professors are familiar with it and the industry demands qualified C++ developers.

Most of its benefits come from the fact that it is a compiled language. The compilation process allows to perform optimizations and detect bugs in the code using static analysis. In this compilation process, the original C++ code gets translated to low level assembly and ultimately machine code to allow its execution by the computer.

1.1.1. Problem

As with almost anything, compilation also comes with some inconvenience. A C++ compiler needs to be installed in the system in order to be able to compile the programs. Compilers in general and C++ compilers in particular are not installed by default in Windows and Macintosh operating systems, also not in some Linux distributions, and even less in mobile environments such as Android and Apple iOS. This adds an extra problem of setting up the environment to allow for the compilation and execution of C++ programs, which is not convenient or even possible in certain situations. Yet another inconvenient of compilation is that of the development process. When developing a program, we usually need to test it very often in order to check for bugs or performance issues. Compilation makes this develop, execute for test, repeat loop slightly more tedious, because it adds a new step in the process, the compilation step, resulting in the develop, compile, execute for test loop. Still, the compile and execute for test pair of tasks can be merged into a single step using shell script automation, but most people are unaware of this, don’t want to bother to, or don’t know how to program shell scripts.

We must also note that coding in any computer language requires the use of software tools such as an editor, preferably one with syntax highlighting, or a debugger, essential to understand
how a program execution behaves when learning programming or tracing errors in programs. Again, those tools usually don’t ship by default with most operating systems so they have to be installed, adding yet more tasks to the environment setup. To illustrate an example where this might be a problem, consider a school or university which is willing to teach C++ programming to its students. It will have to spend money and time in order to set up the environment on its computers. Likewise, the students may want, need, or find convenient to be able to practice programming in situations where they do not have access to a computer with a conventional C++ development environment.

1.1.2. Solution

Browsers, on the other hand, are a common piece of software in almost any operating system. They are broadly used by all kinds of people and most are familiar with them. There is even an operating system which is purely based on web browsing, Chrome OS [19]. Internet access and usage is also nowadays widely spread and growing in most countries [15].

So here comes the question: What if we could embed the C++ development environment in a single website application that runs in the browser?

All modern browsers come with a Javascript interpreter, so they have the capability to run Javascript code. We could write a Javascript program which emulates all the tasks that the C++ compiler, the code editor, the terminal and the debugger do, all in a single application. This application would not need any program installation other than the browser, which already ships with a vast majority of operating systems. Even more, such an application wouldn’t even need a continuous connection to Internet, because once the application (page) is loaded, everything would be executed in the browser, client-side.

C– [14] [13] is the realisation of such an application. It was initially developed by me and two other colleagues from the Compiler Construction subject, Pau Oliver and Ricard Gascons. It was part of the course’s final project, and the idea originally came from the proposal of Jordi Petit, a University professor and also advisor of this thesis. It is built with the objective to improve his co-owned virtual learning environment for computer programming, Jutge.org [6], which is used as a learning and evaluation environment in the University.

Initially, C– was simply an implementation of the core features of the C++ compiler, together with a website interface, equipped with an editor and compilation and execution output displays. Later on, the project was extended with a debugger and a better interface, among other improvements, by Ricard Gascons, as part of his Bachelor’s thesis [12]. This thesis will extend it even further by adding language features from C++ which are missing in the current implementation, improving the compiler’s extensibility and maintainability, improving the compiler error feedback and enhancing the website interface.

![Figure 1.1: Evolution of the C– Project](image-url)
1.1.3. Stakeholders

There are many groups and individuals which may benefit from or produce an effect on the improvements that will be implemented in this thesis. They are mainly divided in five categories or groups: the users, the academia, the programming judge platforms, the project developer and the project advisor.

Users

The users of the application will now have a new tool which will help them in a different way depending on their purposes.

- **C++ developers**: C– is not a tool for developing or running serious real-world applications. Still, this does not mean that experienced C++ developers will not benefit from it. Sometimes when developing applications developers come up with doubts on how a certain piece of code would behave in a given situation, and they may not want to include it in their project’s code until they are sure about its behaviour, because testing it might be more difficult in the context of the whole project, and they might forget to test it in the end. A tool such as C– allows for fast testing of programs because it is very simple, quick to use and accessible from a large number of devices, hence it may help in those situations.

- **C++ students**: Having a C++ environment with a debugger and a proper editor in the browser kills any kind of excuse by the students for not practicing programming outside the university or school. One can practice programming from a PC without the environment, a tablet, or even a big screen smartphone.

- **C++ teachers**: The website also opens the door to new forms of faster and clearer feedback from the teachers, by implementing side functionality on the website which takes advantage of the available data and features.

Academia

- **Schools**: Sometimes small schools lack the budget to prepare the environment on their computers for students to practice programming, or may not be willing to invest money in it because they do not consider it important enough. This project allows them to spend no more budget than that required to buy and setup the computers and the Internet access, which the majority of schools already have in most developed countries [8].

- **Universities**: Most importantly, as stated before its students would benefit from it, and thus so would the university, which wants their students to be taught as efficiently as possible. Universities can also develop new tools on top of the website application or integrate it in their current platforms in order to improve their teaching methodologies used in programming subjects.

Online Programming Judges

Online programming judges (see [3] and [6] as examples) could use the project by integrating it with their website platforms, and provide a convenient interface for their users to program, debug and test their code before submitting it, without them having to ever get out of the browser or attach the code file. In some cases, it might reduce the server traffic and load caused
by users who use the server compiler and automatic tester to see whether their code works, without testing it first on their local machine.

**Project developer**

The project developer will directly affect on it by performing its planning, execution, documentation and validation, following the guidelines from the project advisor. In this case it will also be benefited from it given its condition of student and C++ developer.

**Project advisor**

The project advisor will guide and supervise the project developer during the development process, helping it to achieve the project goals by suggesting improvements and providing with bibliographic resources. He will also benefit from it as a co-owner of an online programming judge and university teacher.

### 1.1.4. State of the art and possible alternatives

Many C++ to Javascript compilers, interpreters, and website development environments have already been built. Yet none of them fully meet the requirements and goals of this project. Follows a description of possible alternative solutions and a justification of their inability to meet the requirements.

**C++ to Javascript compilation**

A possible strategy to implement a C++ development environment in a browser would be to find, implement, or obtain a C++ to JS compiler. With such a compiler, C++ programs developed on the browser could be compiled to JS, which could then be executed in the browser using the Javascript built-in function `eval(code)`. There is a requirement, though. Such a compiler would have to be written in JS in order to be able to be executed in the browser.

All the current implementations of such compilers rely on C++ code. We can find Cheerp [23](written in C++), and Emscripten [5](written in JS). Cheerp is a C++ to JS compiler based on LLVM and Clang, and Emscripten is a LLVM to JS compiler. They both can work (actually that’s how Cheerp works) as a C++ to JS compiler by translating the C++ code to the LLVM intermediate language using Clang and then translating the LLVM language to JS with their own implementation. Clang is written in C++, hence we do not have a full JS implementation for the compiler.

Still, we must notice something: From this setup, we could already possibly obtain a C++ to JS compiler written in JS, if we ignore possible problems that we might come up with, given the limitations of LLVM to JS translations [26]. Given the fact that we have a C++ to JS compiler, we could use it to translate the Clang implementation to JS, outside the browser. Now we would have a full JS implementation of a C++ to JS compiler that we can use in the browser, which works by translating the C++ source code to LLVM using the obtained JS implementation of Clang and then using Emscripten to compile the LLVM code to JS.

We are omitting an important fact though, which is that Clang is a very big project in code size. Its implementation is in the order of several MB of C++ code, a size which may grow...
significantly when compiled to JS, even with code minification. This would make the task of loading the website’s code in a browser effectively impractical.

**Server-side C++ compilation and execution**

The current most common solution is to send the code via an HTTP request to an online server, which has the C++ environment installed and uses a conventional compilation and execution process, in order to respond to the client with the execution output or compilation error of his program.

This approach seems very logical and relatively simple to implement but it has some drawbacks. Because the compilation and execution is done in the server, its load may quickly rise if a lot of users are working with the service and compile or run code very often, so it does not scale very well. Also, security measures such as the creation of virtual machine environments for the programs to run on must be taken in order to prevent malicious code to run on the server, which increases the overhead even further. Finally, because the compilation and execution requires communication with the server, this system does need a continuous Internet connection to work.

An example of such a solution is the Jutge.org IDE [20], developed as a bachelor final project by Albert Llop and supervised by Jordi Petit. This solution has already been implemented in the online programming Judge of the university, but has proven to be hard to scale on high user load.

**C++ interpretation in Javascript**

It turns out that a C++ interpreter written in Javascript already exists. It is called JSCPP [30] and was written by Felix Hao. JSCPP which comes with a debugger and accepts a subset of the C++ language. A requirement of the solution implementation, though, is to have an easily scalable, maintainable, and extensible code-base for the application. The JSCPP interpreter is built in a non-standard way, without a proper grammar definition and parser generator, and with a monolithic architecture, which would make further extensions and maintenance increasingly complex as new language features are implemented.

### 1.2. Objectives

The main objective of this project is to improve the current implementation of the C– development environment.

The main aspects which can be improved in the current implementation are:

- **The compiler’s maintainability, extensibility and readability.** This will make it easier for me and for other people that might get involved in the project later on to quickly get familiarized with the codebase and be able to extend it without breaking the current implementation.

- **The number of features supported from the C++ language standard specification.** Currently only basic features are supported, and more features have to be added in order for the environment to be more attractive for students to practice on it.
The compiler error feedback. Currently compiler errors do not report which line of code produced the compilation error, and usually no names of variables or types are reported within the error that would make identifying the issue much easier.

The website interface, which is currently confusing and buggy in some aspects, and doesn’t offer a very good user experience.

The software validation methods, which are currently very scarce. Adding more validation methods is key to ensure the compiler’s and interpreter’s correctness and establish a make-tests-code-test-fix-repeat loop habit that will guarantee a high probability of quickly spotting the breaking of previously working features when implementing new ones.

1.3. Scope

The project underwent major changes in its scope due to circumstances that occurred during its development. They were caused due to three major reasons:

- An overoptimistic prediction of the difficulty to implement new features: The complexity that comes with implementing a new feature is not constant with respect to the size of the compiler. As more features are added into it, adding a new feature becomes more complex because of the interactions that this new feature has with already implemented features. Also, some features have a lot of semantic nuances and special requirements that must be acknowledged and checked, requiring a lot of extra work aside from the actual run-time simulation of the feature. Those aspects were not taken in account enough during the planning of the project, which lead to an overoptimistic prediction of the time needed for the tasks.

- The current implementation of the compiler had several major bugs that were introduced in the previous version and had to be fixed. For some of them there was no obvious trivial fix because the incorrect behaviour was not due to the code itself but to the very nature of the approach of the solution, which was not correct.

- Some small features that were not considered in the initial plan were implemented, some of them were necessary to ease the development process, such as a command line interface, and others turned out to be very easy to implement after the changes that were made in the codebase to enhance its extensibility.

Because of this reason, the changes in the scope specification are detailed in section 1.3.2 and the initial scope is detailed in section 1.3.1.

Note that the initial objectives of the project were not changed at all. All the objectives that are specified in section 1.2 still apply to the project and are reflected in the final scope specification.

1.3.1. Initial scope

- Add new features to the C++ language:
  - Library support, together with the implementation of the essential features from the commonly used libraries: iostream, vector, string and cmath.
  - Structs, including those with recursive definitions.
- C Arrays, including pointers.
- String modification (the current implementation only allows string access).
- References.
- Const attribute for variables.
- Preserve or even enhance the code maintainability, readability and extensibility by modularizing its compiler.
- Improve and enlarge the validation tests to ensure the correctness of the compiler and interpreter with automated testing.
- Improve the compiler feedback on semantic errors. Given that the application is targeted at students, good feedback needs to be provided for them to be able to spot their mistakes more easily. Such feedback should be provided in a similar format as the one used by the commonly used compilers in the industry, for them to be able to easily familiarize with it. Line and column information will be reported with the compiler error message and as much references as possible to variable ids and types will be provided to ease the detection of the source of the problem.
- Improve the website interface. The current interface is confusing in some aspects and has room for improvement.
- Update the debugger so that it works with the new language features.

1.3.2. Final scope

The main changes to the scope regard to the new features that are to be supported. The following features are removed from the scope of the project:

- Library support, mainly due to a lack of time for the reasons mentioned in the beginning of section 1.3.
- Structs. Its implementation was started, but a major difficulty caused by an ambiguity between type and variable identifiers that cannot be resolved during syntactic analysis, together with a lack of time lead to the decision of discarding it.
- References. Its grammar and semantic implementations were completed but the code generation turned out to be more complex than expected.

And the following features are added in turn:

- Global variables: Being able to declare variables in the global scope, also allowing initialization for the declared variables in the form of assignments from expressions.
- Variable shadowing, which allows to declare a variable with a name that already appears in a scope higher in the hierarchy.
- New and delete operators, which allow to allocate and deallocate chunks of memory in the heap. All types allowed by the standard and supported by the current version will be supported as arguments for the new operator, including arrays which have a dynamic size in its first dimension.

Another element that was not included in the first scope specification but is part of the project is fixing any existing bugs in the current implementation.
1.3.3. Requirements

This project emulates a C++ environment, so it must fulfill all the rules defined by the C++ standard, which are specified in it. C++ will as far as possible conform to the latest C++14 standard. Because the final versions of the standard are not available for free, a working draft will be used as a reference [10]. Note that working drafts may be incomplete or incorrect, but they usually closely approximate the published standard.

1.3.4. Possible obstacles

With big and complex programming projects such as C++, obstacles and problems are very prone to occur during the development process. For that reason, they need to be acknowledged in advance and alternatives have to be planned.

- **Breaking of current implemented features** when adding new ones: This is a common problem with any programming project. Adding new features may involve tweaking existing code which may in turn break current functionality. Those type of mistakes will be detected with validation methods, which include several testing strategies, and they will be understood and fixed using debugging tools.

- **Time constraints**: Some feature may turn out to be much more difficult to implement than expected, or bugs may slow down the development process. In such situations, depending on whether the involved feature is important enough to justify the effort or not, other less important features, or the feature itself will be cut out from the project.
Part I

Planning
2.1. Methodology

A good methodology is key for the success of the project. Several tools and existing methodologies which have been extensively used in large scale projects and have proven to be fruitful will be adopted. They are specified in the following sections.

2.1.1. Development methodology

This project will use the Scrum methodology [36], based on the agile framework. As stated in the official Scrum website, *Scrum is a set of principles and practices which helps deliver products in short cycles, enabling fast feedback, continual improvement and rapid adaptation to change*. Its use is appropriate in this project because it allows for flexibility and adaptation to emerging requirements, and it helps dealing with unpredicted challenges that can occur during the development process.

Here is an outline of the scrum cycle, and how it will be applied in this project:

1. **Project backlog**: First of all, a prioritized list of tasks necessary for the completion of the project will be enumerated together with the project director. This is called the project backlog.

2. **Sprints**: At the start of every cycle, a small number of tasks will be pulled from the top of the project backlog. This set of tasks is called the sprint backlog. The expected amount of work required to complete each of the tasks will be estimated and relevant details of the implementation will be discussed with the project director.

3. **Continuous assessment**: Every day, the progress of the tasks development will be assessed via email communication.

4. **Review and retrospective**: Each sprint will have a time span of about 1 to 2 weeks. At the end of the sprint, all the tasks in the sprint backlog should be completed and its results ready to be shown to the stakeholders. The completion and quality of the tasks
will be assessed and possible improvements or new potential tasks arising from the new current situation will be considered.

5. **New cycle**: After the review of the previous sprint, a new cycle will start.

### 2.1.2. Tracking methods

In order to be able to track and review the progress of the project, several tools will be used, which include Version Control Systems (VCS) [38], communication tools, together with the weekly meetings.

- **Git and GitHub**: Git [28] is a Version Control System, which is used to track changes on computer files. In this case, it will be used to track changes in the code and tests, and in order to be able to rollback to previous states of the project if necessary, for instance in case of breaking implementations. C++ is already using Git as a VCS, so no extra work will be required to set it up. GitHub [29] is a website platform that provides an interface to easily visualize the project changes and track the work done by the developers. It does also support issue tracking, which is useful to manage bugs and undesired behaviour of the application which has to be fixed.

- **Meetings**: Every one or two weeks face-to-face meetings will take place with the project director to comfortably perform all the relevant tasks from the scrum cycle explained in the previous section.

### 2.1.3. Validation methods

A strong set of validation methods will be used in order to assess the quality and correctness of the application and documentation. Those include software testing strategies, performance benchmarks and user feedback.

- **Specific hand-crafted C++ program sample tests**: Those will be used to check that each component of the application functions as expected. For each feature of the language, one or more C++ programs will be defined that test the specific feature for correctness, checking that the compiler outputs compilation errors when necessary and that the execution outputs match the expected.

- **Real-world C++ program sample tests**: This sample tests will consist of correct (compiling) C++ programs extracted from several sources such as Jutge.org problem solutions, and their purpose will be to test C++ features in general, complementing the specific tests.

- **Performance benchmarks**: In order to evaluate how efficient in memory and execution time is the C++ implementation after features are implemented, and in order to detect potential bottlenecks in the application, its performance will be compared to previous versions of C++ and other programming languages.

### 2.2. Technology stack

The main technologies that will be used for the development of the project are Javascript for the compiler logic and interface interaction, together with CSS/HTML for the interface layout and design. This is technologies are the standard for developing websites and cannot be changed.
More specifically, the Coffeescript language will be used instead of Javascript. Coffeescript is a language that compiles to Javascript and offers a cleaner, easier to read and more consistent syntax and semantics.

The choice of this language over Javascript is justified for several reasons:

- The project is currently written in Coffeescript, so choosing another compile-to-js language would require rewriting existing parts.
- Coffeescript is easier to write and read than Javascript.
- It offers interesting features such as pattern matching and list comprehensions which Javascript lacks.
- It eliminates some flaws in the Javascript semantics that may lead to bugs, such as different comparison operators, and hard to distinguish null-like values.

2.3. Time planning

This project was started on February 2017 and has a deadline for mid-late June 2017. This means the total available time for its development is of approximately 5 months. Because the amount of tasks that are required to complete the project is not small, a comprehensive time plan has to be made, estimating the amount of time each task needs, and making a schedule that takes in account the dependencies between them. Considering the available time for each day/week of the person in charge of the development, the schedule has to ensure that it will be completed in time.

2.3.1. Tasks

The project tasks can be classified in the following groups: Definition and Planning, Development and Documentation and Presentation.

Definition and Planning

This group consists of tasks which are targeted at defining what the project is about, its current state of the art and its goals, together with the planning that comes after that: the methodologies that will be used, the time plan, the budget plan and the sustainability analysis.

It is part of the Project Management Course and does also include two oral presentations where the definition and planning has to be summarized and presented orally, and a specialization competences review where the adjustment of the project to the Computing degree specialization characteristics has to be justified.

On Table 2.1 you can find a list of each task and the corresponding expected time span.

Development

This component consists of the main tasks of the project, aiming at accomplishing its primary goals. It starts by analysing the current implementation, in order to understand how it can be extended and find any possible bugs that may exist in it by using a set of validation tests. Those bugs have to be fixed in order to be able to continue by implementing each of the new planned features.
Table 2.1: Estimation of the time that will be spent on each task related to *Definition and Planning*

<table>
<thead>
<tr>
<th>Task</th>
<th>Expected time span</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context definition</td>
<td>8h</td>
</tr>
<tr>
<td>State of the art analysis</td>
<td>6h</td>
</tr>
<tr>
<td>Scope definition</td>
<td>2h</td>
</tr>
<tr>
<td>Methodology specification</td>
<td>2h</td>
</tr>
<tr>
<td>Time plan</td>
<td>8h</td>
</tr>
<tr>
<td>Budget plan</td>
<td>6h</td>
</tr>
<tr>
<td>Sustainability analysis</td>
<td>3h</td>
</tr>
<tr>
<td>Preliminary presentation</td>
<td>4h</td>
</tr>
<tr>
<td>Specialization competences review</td>
<td>8h</td>
</tr>
<tr>
<td>Final planning document and oral presentation</td>
<td>12h</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>59h</strong></td>
</tr>
</tbody>
</table>

Note that objectives such as preserving the maintainability of the code or updating the debugger to work with the new features are considered as part of each feature implementation task.

On Table 2.2, you can find a list of each task and the corresponding expected duration.

<table>
<thead>
<tr>
<th>Task</th>
<th>Expected time span</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current implementation analysis</td>
<td>10h</td>
</tr>
<tr>
<td>Generation of validation tests</td>
<td>8h</td>
</tr>
<tr>
<td>Fixing of bugs in the current implementation</td>
<td>30h</td>
</tr>
<tr>
<td>Implementation of const attribute for variable declaration</td>
<td>12h</td>
</tr>
<tr>
<td>Implementation of C arrays and pointers</td>
<td>45h</td>
</tr>
<tr>
<td>Implementation of String modification</td>
<td>4h</td>
</tr>
<tr>
<td>Implementation of Structs</td>
<td>50h</td>
</tr>
<tr>
<td>Implementation of References</td>
<td>15h</td>
</tr>
<tr>
<td>Implementation of library support</td>
<td>60h</td>
</tr>
<tr>
<td>Improvement of the compiler feedback</td>
<td>30h</td>
</tr>
<tr>
<td>Improvement of the website interface</td>
<td>35h</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>299h</strong></td>
</tr>
</tbody>
</table>

Table 2.2: Estimation of the time that will be spent on each task related to the *Project Development*

**Documentation and Presentation**

This set of tasks aims at documenting all the development process in the written project memory, and preparing an oral presentation for an audience. The corresponding expected time spent on each task can be seen in Table 2.3.

Note that the project memory composition task does not include the *Definition and Planning* part explained in section 2.3.1.
<table>
<thead>
<tr>
<th>Task</th>
<th>Expected time span</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project memory composition</td>
<td>55h</td>
</tr>
<tr>
<td>Oral presentation preparation</td>
<td>25h</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>80h</strong></td>
</tr>
</tbody>
</table>

Table 2.3: Estimation of the time that will be spent on each task related to the *Documentation and Presentation*

**Total time**

The previous analysis leads to a total project development time of **438 hours**, as seen in Table 2.4.

The project has a duration of approximately 120 days, considering it starts from February 1st and has to be finished by June 1st, leaving some extra time for unexpected problems and delays. This means that the developer should spend approximately an average of **3.65 hours a day** to complete the project in time.

<table>
<thead>
<tr>
<th>Task</th>
<th>Expected time span</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition and Planning</td>
<td>59h</td>
</tr>
<tr>
<td>Development</td>
<td>299h</td>
</tr>
<tr>
<td>Documentation and Presentation</td>
<td>80h</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>438h</strong></td>
</tr>
</tbody>
</table>

Table 2.4: Estimation of the time that will be spent on each task group and total estimated development time

**Resources**

In order to complete the tasks, human resources, hardware resources and software resources are required. Some groups of tasks need different resources than others, so for each resource the group of tasks that make use of it is detailed.

**Human resources**

- **Project manager**: Needed for the *Definition and Planning* tasks and the *Documentation and Presentation* tasks.
- **Software engineer with knowledge in compiler construction**: Needed for the *Project Development* tasks and the *Documentation* task.
- **Software tester**: Needed for the *Project Development* tasks, specifically related to the generation of validation tests and the testing of each feature implementation.
- **Designer**: Needed for the *Project Development* tasks, in order to design the improvements to the website interface.

**Hardware resources** For every task of the project a computer with a **screen**, **keyboard** and **mouse** is required, together with **Internet connection** and **electricity** to power everything.
Software resources  Follows a list of the required software resources:

- **Text processor**: Used to write the documentation.
- **Presentation designer**: Used to prepare the presentation material.
- **Programming IDE**: Necessary to facilitate the software development tasks (a text processor could also be used, but is much more tedious and time-consuming).
- **Grammar parser**: Because writing a grammar parser generator to parse the C++ code from scratch is not feasible nor reasonable, an already written one will be used.
- **Version Control System**: In order to track and manage changes in the software code.
- **Internet browser**: In order to search for online resources and communicate with stakeholders via email.
- **Operating System**: Serves as a base to install and use the rest of the software.
2.3.2. Gantt chart

<table>
<thead>
<tr>
<th>February</th>
<th>March</th>
<th>April</th>
<th>May</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>11-20</td>
<td>21-28</td>
<td>1-10</td>
</tr>
</tbody>
</table>

- Context definition
- State of the art analysis
- Scope definition
- Methodology specification
- Time plan
- Budget plan
- Sustainability analysis
- Preliminary presentation
- Specialization competences review
- Planning document and presentation
- Current implementation analysis
- Validation tests
- Fixing current bugs
- Const attribute
- C Arrays and pointers
- String modification
- Structs
- References
- Library support
- Compiler feedback
- Website interface
- Project memory
- Oral presentation

Figure 2.1: Gantt chart showing the task dependencies and the approximate timeline
In Figure 2.3.2, we can see a Gantt chart with all the tasks, its dependencies and its expected approximate temporal planning. Because some tasks have no dependencies between each other, they can be developed in parallel. This is true for the Definition and Planning tasks and the first Development tasks, which can be started even if the planning is still not complete.

In the case of the Documentation and Presentation tasks, given its transverse nature they have to be developed in parallel with all the other tasks, even if there are some obvious dependencies from the development tasks to the documentation, which have not been made explicit to avoid a too complex diagram. The same can be said for the interface improvement, which includes adding some controls and display new information related to the new implemented features.

### 2.3.3. Potential deviations and alternative plans

There is a chance that deviations to the presented schedule occur during the development process. They could be caused by one or more of the following reasons:

- Changes in the project scope related to the adaptive nature of the agile methodology.
- Breaking of current implemented features when adding new ones.
- Finding and having trouble to fix bugs which were present in the project before starting its extension.
- Having difficulties to implement some feature because its complexity is higher than was expected.

These deviations may lead to longer development time. For this reason, to ensure that there will be no delays, a margin time-frame of 15 days has been set from 1st June to 15th June to ensure that those deviations can be addressed during this period. In case of being used, it allows for an increase of 54.75h to the available time (considering 3.65h/day), which will add to a total of development time of 492.75h.

Because this may not be enough if the delay is severe, and given the large amount of goals that has been set out for this project, one or two non-essential tasks such as the implementation of library support or the compiler feedback improvement will be cut out from the project, effectively reducing the expected development time by at least 90h to a total of 348h, considering that there is also time associated with the project memory and presentation that is related to both tasks. These two tasks don’t have any dependency relation with any other task, and they don’t need any special resources, so no other modifications to the schedule will have to be performed.

Both measures ensure that the project will be completed well in time for the deadline, giving a total margin of 54.75 + 90 = 144.75h.

### 2.3.4. Plan modifications

As stated in the beggining of section 1.3, major changes were made to the scope of the project during its development, which in turn affected the initial planification. The justification of these changes is specified in that section. Other elements that were specified both in the initial and final scope also underwent time duration changes, either because they were found to be harder or easier to implement than expected, which is common in software development projects.
The new planification is specified in this section, with the actual time spent in each task (Tables 2.5 and 2.6) and the new Gantt chart resulting from the changes (Figure 2.3.4).

New tasks are marked in green color, removed tasks are marked in red color and tasks for which the time changed are marked in orange color. Note that some tasks are marked in red (they were removed) but their time is still accounted for. This is because they were started but not completed.

Also note that the task Reorganization of the code to improve maintainability was in the initial scope but didn’t figure in the first planification because it was included inside the Fixing bugs in the current implementation task. It is included separately in the final planification for more clarity and specificity.

**Development**

<table>
<thead>
<tr>
<th>Task</th>
<th>Time span</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current implementation analysis</td>
<td>10h</td>
</tr>
<tr>
<td>Generation of validation tests</td>
<td>8h 35h</td>
</tr>
<tr>
<td>Fixing of bugs in the current implementation</td>
<td>30h 75h</td>
</tr>
<tr>
<td>Reorganization of the code to improve maintainability</td>
<td>15h</td>
</tr>
<tr>
<td>Implementation of const attribute for variable declaration</td>
<td>12h 6h</td>
</tr>
<tr>
<td>Implementation of global variables</td>
<td>5h</td>
</tr>
<tr>
<td>Implementation of variable shadowing support</td>
<td>6h</td>
</tr>
<tr>
<td>Implementation of C arrays and pointers</td>
<td>45h 55h</td>
</tr>
<tr>
<td>Implementation of String modification</td>
<td>4h 0h</td>
</tr>
<tr>
<td>Implementation of new and delete operators</td>
<td>13h</td>
</tr>
<tr>
<td>Implementation of Structs</td>
<td>50h 15h</td>
</tr>
<tr>
<td>Implementation of References</td>
<td>15h</td>
</tr>
<tr>
<td>Implementation of library support</td>
<td>60h 0h</td>
</tr>
<tr>
<td>Improvement of the compiler feedback</td>
<td>30h</td>
</tr>
<tr>
<td>Improvement of the website interface</td>
<td>35h 48h</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>299h 328h</strong></td>
</tr>
</tbody>
</table>

Table 2.5: Actual time spent on each task related to the Project Development

**Total time**

<table>
<thead>
<tr>
<th>Task</th>
<th>Expected time span</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition and Planning</td>
<td>59h</td>
</tr>
<tr>
<td>Development</td>
<td><strong>299h 328h</strong></td>
</tr>
<tr>
<td>Documentation and Presentation</td>
<td>80h</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>438h 467h</strong></td>
</tr>
</tbody>
</table>

Table 2.6: Actual time spent on each task group and total development time
Figure 2.2: Gantt chart showing the task dependencies and timeline for the new plan
As you can tell from the Gantt chart in Figure 2.3.4, the timeframe that was set for June in case of delays was finally used to account for the extra development time due to the changes in the scope.

2.4. Economic management

2.4.1. Cost analysis

This project will make uses of several resources, which have an associated cost. The following sections present a quantitative analysis of the different types of costs involved.

**Human resources**

Before estimating the cost associated to each task, we need to define the cost per hour of the different human resource roles that will perform them. On Table 2.7 we can find the cost per hour of each role, based on [35]’s data for Spain, where the project will be developed, and assuming 170 hours of work per month.

<table>
<thead>
<tr>
<th>Role</th>
<th>Wage per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT Project manager (PM)</td>
<td>30€</td>
</tr>
<tr>
<td>Software engineer (SE)</td>
<td>25€</td>
</tr>
<tr>
<td>Web designer (WD)</td>
<td>19€</td>
</tr>
</tbody>
</table>

Table 2.7: Estimated hourly wage for each Human Resources role

With these hourly wages in mind, we can estimate the human resources cost of the project by referring to the expected time duration of each task and the roles associated with it. The resulting estimation is specified on Table 2.8.

Note that from the website interface improvement task 15h out of the total 35h have been considered as part of the design of the interface, whereas the rest is considered part of the project development, which consists in implementing the design. This was necessary because the role implicated in the development is different for each sub-task.

**Hardware resources**

Because this is mainly a software development project, computer hardware is essential for its development. Follows Table 2.9 with all the hardware that will be used, its associated purchase price, useful lifespan, expected usage time and amortization cost.

The amortization cost is computed following the Formula:

\[
\text{Amortization cost} = \frac{\text{Purchase price}}{\text{Useful life span}} \times \text{Usage time} \tag{2.1}
\]

Where the useful life span is computed using the Formula 2.2. Here, the working days in the life span of the hardware resource are obtained using data from [25] and \textbf{Working hours per working day} = 8 is assumed. All hardware resources are assumed to have a life span of 4 years since the purchase.
<table>
<thead>
<tr>
<th>Task</th>
<th>Role</th>
<th>Hours</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context definition</td>
<td>PM</td>
<td>8h</td>
<td>240€</td>
</tr>
<tr>
<td>State of the art analysis</td>
<td>PM</td>
<td>6h</td>
<td>180€</td>
</tr>
<tr>
<td>Scope definition</td>
<td>PM</td>
<td>2h</td>
<td>60€</td>
</tr>
<tr>
<td>Methodology specification</td>
<td>PM</td>
<td>2h</td>
<td>60€</td>
</tr>
<tr>
<td>Time plan</td>
<td>PM</td>
<td>8h</td>
<td>240€</td>
</tr>
<tr>
<td>Budget plan</td>
<td>PM</td>
<td>6h</td>
<td>180€</td>
</tr>
<tr>
<td>Sustainability analysis</td>
<td>PM</td>
<td>3h</td>
<td>90€</td>
</tr>
<tr>
<td>Preliminary presentation</td>
<td>PM</td>
<td>4h</td>
<td>120€</td>
</tr>
<tr>
<td>Specialization competences review</td>
<td>PM</td>
<td>8h</td>
<td>240€</td>
</tr>
<tr>
<td>Planning document and presentation</td>
<td>PM</td>
<td>12h</td>
<td>360€</td>
</tr>
<tr>
<td>Current implementation analysis</td>
<td>SE</td>
<td>10h</td>
<td>250€</td>
</tr>
<tr>
<td>Validation tests</td>
<td>SE</td>
<td>8h</td>
<td>200€</td>
</tr>
<tr>
<td>Fixing current bugs</td>
<td>SE</td>
<td>30h</td>
<td>750€</td>
</tr>
<tr>
<td>Const attribute</td>
<td>SE</td>
<td>12h</td>
<td>300€</td>
</tr>
<tr>
<td>C Arrays and pointers</td>
<td>SE</td>
<td>45h</td>
<td>1125€</td>
</tr>
<tr>
<td>String modification</td>
<td>SE</td>
<td>4h</td>
<td>100€</td>
</tr>
<tr>
<td>Structs</td>
<td>SE</td>
<td>50h</td>
<td>1250€</td>
</tr>
<tr>
<td>References</td>
<td>SE</td>
<td>15h</td>
<td>375€</td>
</tr>
<tr>
<td>Library support</td>
<td>SE</td>
<td>60h</td>
<td>1500€</td>
</tr>
<tr>
<td>Compiler feedback</td>
<td>SE</td>
<td>30h</td>
<td>750€</td>
</tr>
<tr>
<td>Website interface</td>
<td>WD, SE</td>
<td>15h, 20h</td>
<td>785€</td>
</tr>
<tr>
<td>Project memory</td>
<td>PM</td>
<td>55h</td>
<td>1650€</td>
</tr>
<tr>
<td>Oral presentation</td>
<td>PM</td>
<td>25h</td>
<td>750€</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>438h</td>
<td><strong>11555€</strong></td>
</tr>
</tbody>
</table>

Table 2.8: Estimated cost for human resources tasks

Useful life span = Working days in life span * Working hours per day \hspace{1cm} (2.2)

<table>
<thead>
<tr>
<th>Resource</th>
<th>Purchase price</th>
<th>Useful lifespan</th>
<th>Expected usage time</th>
<th>Amortization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop</td>
<td>2,483€</td>
<td>8032h</td>
<td>438h</td>
<td>135.4€</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2,483€</strong></td>
<td><strong>8032h</strong></td>
<td><strong>438h</strong></td>
<td><strong>135.4€</strong></td>
</tr>
</tbody>
</table>

Table 2.9: Estimated amortized cost for the hardware resources

Note that the only hardware necessary for the project is the laptop. There are no other peripherals needed for the project, nor other types of hardware resources.

Regarding the cost of hosting the website server and domain, no extra costs will be incurred given that the domain and servers are and will continue to be hosted in the UPC’s Jutge.org servers.
Software resources

Because all software resources that are used in this project are free, they are included in Table 2.10 together with their licenses, but their cost analysis is not performed.

<table>
<thead>
<tr>
<th>Software resource</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebStorm</td>
<td>Licensed to Albert Segarra Roca (<a href="mailto:albert.segarra.roca@est.fib.upc.edu">albert.segarra.roca@est.fib.upc.edu</a>)</td>
</tr>
<tr>
<td>Git</td>
<td><a href="https://github.com/git/git/blob/master/LGPL-2.1">https://github.com/git/git/blob/master/LGPL-2.1</a></td>
</tr>
<tr>
<td>GitHub</td>
<td><a href="https://desktop.github.com/eula/">https://desktop.github.com/eula/</a></td>
</tr>
<tr>
<td>Trello Free</td>
<td><a href="https://trello.com/pricing">https://trello.com/pricing</a></td>
</tr>
<tr>
<td>Google Chrome</td>
<td><a href="https://www.google.com/chrome/browser/privacy/eula_text.html">https://www.google.com/chrome/browser/privacy/eula_text.html</a></td>
</tr>
<tr>
<td>Sharelatex</td>
<td><a href="https://github.com/sharelatex/sharelatex/blob/master/LICENSE">https://github.com/sharelatex/sharelatex/blob/master/LICENSE</a></td>
</tr>
<tr>
<td>Jison</td>
<td><a href="https://zaa.ch/jison/docs/#license">https://zaa.ch/jison/docs/#license</a></td>
</tr>
<tr>
<td>NodeJS</td>
<td><a href="https://github.com/nodejs/node/blob/master/LICENSE">https://github.com/nodejs/node/blob/master/LICENSE</a></td>
</tr>
<tr>
<td>Travis CI</td>
<td><a href="https://travis-ci.com/plans">https://travis-ci.com/plans</a></td>
</tr>
<tr>
<td>Mendeley</td>
<td><a href="https://www.mendeley.com/terms">https://www.mendeley.com/terms</a></td>
</tr>
<tr>
<td>AngularJS</td>
<td><a href="https://github.com/angular/angular.js/blob/master/LICENSE">https://github.com/angular/angular.js/blob/master/LICENSE</a></td>
</tr>
<tr>
<td>ACE Editor</td>
<td><a href="https://github.com/ajaxorg/ace/blob/master/LICENSE">https://github.com/ajaxorg/ace/blob/master/LICENSE</a></td>
</tr>
<tr>
<td>jQuery Terminal</td>
<td><a href="https://github.com/jcubic/jquery.terminal/blob/master/LICENSE">https://github.com/jcubic/jquery.terminal/blob/master/LICENSE</a></td>
</tr>
<tr>
<td>Mocha</td>
<td><a href="https://github.com/mochajs/mocha/blob/master/LICENSE">https://github.com/mochajs/mocha/blob/master/LICENSE</a></td>
</tr>
</tbody>
</table>

Table 2.10: List of Software resources used in the project together with their licenses

Other resources

Consumption resources such as Internet and Electricity will be used in order to power the laptop. Because the project developer will spend a portion of the time developing the project in the university, which provides free Internet and electricity, the cost will be computed accordingly.

It is expected that the development will take place 30% of the time at the University, which leaves a 70% of the total project development time to be accounted for this costs, which is a total of 306.6h.

- **Internet**: The Internet service where the project will be developed about 70% of the time has a cost of 30€ per month. Because a month has an average of 720h, a rough estimate of the cost per hour is of 30/720 = 0.04167€/h. Because the development time to be accounted for the use of this paid Internet resource is of 306.6h, the total expected cost associated with the project development is of 306.6 * 0.04167 = 12.77€

- **Electricity**: The average power consumption of a laptop is of 60W. The electricity costs approximately 0.13€/kWh where the project will be developed about 70% of the time.

\[1\] This software is free on the purchase of the laptop specified in 2.4.1
This means the expected amount of priced energy that will be consumed is $306.6 \times 60 = 18396\text{Wh} = 18.396\text{kWh}$. Taking in account the electricity cost per kWh, this means that the total electricity cost is $18.396 \times 0.13 = 2.39\text{€}$.

Total cost

The total cost of the project, including a 10% contingency, is specified on Table 2.11.

<table>
<thead>
<tr>
<th>Type of resource</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human resources</td>
<td>11,555.00\text{€}</td>
</tr>
<tr>
<td>Hardware resources</td>
<td>135.40\text{€}</td>
</tr>
<tr>
<td>Software resources</td>
<td>0.00\text{€}</td>
</tr>
<tr>
<td>Internet</td>
<td>12.77\text{€}</td>
</tr>
<tr>
<td>Electricity</td>
<td>2.39\text{€}</td>
</tr>
<tr>
<td>Subtotal</td>
<td>11,705.56\text{€}</td>
</tr>
<tr>
<td>Contingency (10%)</td>
<td>1,170.56\text{€}</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>12,876.12\text{€}</strong></td>
</tr>
</tbody>
</table>

Table 2.11: Estimation of the project’s cost

2.4.2. Control Management

There are some risks to the project that may result in an increase of its cost, the main ones being a possible increase in the development time or a failure or loss of the hardware. Those risks have to be detected, and the possible deviations have to be quantified with control indicators.

Indicators

In order to assess deviations in the cost related to changes in the development time, the real amount of hours spent in the project by each of the human resources roles will be measured and annotated in a table every day. At the end of each week, the total amount of hours spent by each of the roles during the week will be computed and compared with the expected time, as specified in the Gantt chart. The sum of the difference between the expected and real times for each role, times its corresponding wage per hour (See Table 2.7) will give the total deviation in the human resources budget for that week, which can be used as a control indicator.

In the case of deviations due to hardware loss or failure, the deviation can be computed as the cost of the repairs or the new hardware that has to be acquired.

Possible solutions

A possible solution to an increase in cost due to a longer development time would be cutting one or two non-essential features from the project as stated in the time plan, which would reduce the final increase in total human resources cost.

In the event of a hardware loss or failure, given that the only hardware needed for the project is a computer, the University computers, which are free to access, could be used to develop the project instead.
In the case of other minor increases on the cost, the large 10% contingency ensures the cost won’t be exceeded.

2.4.3. Cost modifications

As specified in section 2.3.4, some modifications were made to the initial project planning. This caused an increase in the total development time, which also meant an increase in the total project cost due to the extra hours of human resources involved.

In table 2.12, you can find a list of the new human resources costs for each task and for all the tasks altogether.

<table>
<thead>
<tr>
<th>Task</th>
<th>Role</th>
<th>Hours</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context definition</td>
<td>PM</td>
<td>8h</td>
<td>240€</td>
</tr>
<tr>
<td>State of the art analysis</td>
<td>PM</td>
<td>6h</td>
<td>180€</td>
</tr>
<tr>
<td>Scope definition</td>
<td>PM</td>
<td>2h</td>
<td>60€</td>
</tr>
<tr>
<td>Methodology specification</td>
<td>PM</td>
<td>2h</td>
<td>60€</td>
</tr>
<tr>
<td>Time plan</td>
<td>PM</td>
<td>8h</td>
<td>240€</td>
</tr>
<tr>
<td>Budget plan</td>
<td>PM</td>
<td>6h</td>
<td>180€</td>
</tr>
<tr>
<td>Sustainability analysis</td>
<td>PM</td>
<td>3h</td>
<td>90€</td>
</tr>
<tr>
<td>Preliminary presentation</td>
<td>PM</td>
<td>4h</td>
<td>120€</td>
</tr>
<tr>
<td>Specialization competences review</td>
<td>PM</td>
<td>8h</td>
<td>240€</td>
</tr>
<tr>
<td>Planning document and presentation</td>
<td>PM</td>
<td>12h</td>
<td>360€</td>
</tr>
<tr>
<td>Current implementation analysis</td>
<td>SE</td>
<td>10h</td>
<td>250€</td>
</tr>
<tr>
<td>Validation tests</td>
<td>SE</td>
<td>35h</td>
<td>875€</td>
</tr>
<tr>
<td>Fixing current bugs</td>
<td>SE</td>
<td>75h</td>
<td>1875€</td>
</tr>
<tr>
<td>Reorganization of the code</td>
<td>SE</td>
<td>15h</td>
<td>375€</td>
</tr>
<tr>
<td>Const attribute</td>
<td>SE</td>
<td>6h</td>
<td>150€</td>
</tr>
<tr>
<td>Global variables</td>
<td>SE</td>
<td>5h</td>
<td>125€</td>
</tr>
<tr>
<td>Variable shadowing</td>
<td>SE</td>
<td>6h</td>
<td>150€</td>
</tr>
<tr>
<td>C Arrays and pointers</td>
<td>SE</td>
<td>55h</td>
<td>1375€</td>
</tr>
<tr>
<td>New and delete operators</td>
<td>SE</td>
<td>13h</td>
<td>325€</td>
</tr>
<tr>
<td>Structs</td>
<td>SE</td>
<td>15h</td>
<td>375€</td>
</tr>
<tr>
<td>References</td>
<td>SE</td>
<td>15h</td>
<td>375€</td>
</tr>
<tr>
<td>Compiler feedback</td>
<td>SE</td>
<td>30h</td>
<td>750€</td>
</tr>
<tr>
<td>Website interface</td>
<td>WD, SE</td>
<td>15h, 33h</td>
<td>1110€</td>
</tr>
<tr>
<td>Project memory</td>
<td>PM</td>
<td>55h</td>
<td>1650€</td>
</tr>
<tr>
<td>Oral presentation</td>
<td>PM</td>
<td>25h</td>
<td>750€</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>467h</td>
<td><strong>11555€ 12280€</strong></td>
</tr>
</tbody>
</table>

Table 2.12: Cost for human resources tasks after the original planning change

This means an increase of 725€ for the total cost of the project, which can be deducted from the 10% contigency of 1170.56€ that was set in the initial cost plan. In table 2.13, you can find the total final cost of the project after the changes.
<table>
<thead>
<tr>
<th>Type of resource</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human resources</td>
<td>12,280.00€</td>
</tr>
<tr>
<td>Hardware resources</td>
<td>135.40€</td>
</tr>
<tr>
<td>Software resources</td>
<td>0.00€</td>
</tr>
<tr>
<td>Internet</td>
<td>12.77€</td>
</tr>
<tr>
<td>Electricity</td>
<td>2.39€</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>12,430.56€</strong></td>
</tr>
</tbody>
</table>

Table 2.13: Total project’s cost

2.5. **Sustainability analysis**

A comprehensive analysis of the sustainability of the project is done in this section. It is divided into three categories, economic, social and environmental.

2.5.1. **Economic**

The project is economically sustainable in the sense that the costs have been clearly detailed and are reasonable and tight enough, mostly coming from human resources. The time spent on each task is proportional to its relevance, which makes it economically efficient on the human resources side.

Because there are a lot of stakeholders than can benefit from the project, a monetization could be feasible by offering the software for instance to universities or programming platforms and charging for its use, but due to the open source nature of the project and the fact that it is not aimed to be profitable, this will not be considered.

2.5.2. **Social**

The project will help programmers, students and universities by facilitating the access to a C++ development environment, reducing the initial effort required for beginners to get started in learning programming.

It can also be used to promote programming in schools, where it is currently not taught sometimes because of the lack of resources.

2.5.3. **Environmental**

As computed in section 2.4.1, the total amount of electricity consumed by the project is expected to be approximately of $60W \times 438h = 26.28kWh$. This amounts to 18.5kg of $CO_2$, which is roughly the same as the $CO_2$ expelled in an average 71km car drive [37].

On the other hand, the laptop that will be used for the development is designed with features aimed at reducing its environmental impact [33].

Also, almost no paper will be used for its development, given the fact that most notes will be taken in the computer, and all deliverables are handed in electronically.
Regarding the environmental impact caused by the website’s hosting, the Jutge.org servers running the website comply with European sustainability standards regarding power consumption and equipment disposal, which ensures that they have a low environmental impact.

2.5.4. Sustainability matrix

On table 2.14 you can find the sustainability matrix of the project, which summarizes the explanation of the previous sections. See [16], provided as part of the GEP Course material, for the meaning of each cell.

<table>
<thead>
<tr>
<th></th>
<th>Project development (0:10)</th>
<th>Exploitation (0:20)</th>
<th>Risks (-20:0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environmental</td>
<td>8</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>Economic</td>
<td>7</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>Social</td>
<td>9</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Sustainability (-60:90)</td>
<td></td>
<td></td>
<td>75</td>
</tr>
</tbody>
</table>

Table 2.14: Sustainability matrix
The C– project began a year ago in the form of a final project for the Compiler Construction subject, developed by me and two other colleagues, Ricard Gascons and Pau Oliver. The initial implementation supported basic language features but had a major flaw: execution could not be paused, and thus it was not possible to debug the program and the I/O was not interactive, all the input had to be provided before the execution started and the output was only received when the execution finished.

This problem was solved by Ricard Gascons in his final thesis, which implemented support for step-by-step execution, which means being able to pause the program, provide input and receive output interactively. He also implemented a debugger on top of it and an improved website interface that included debugger controls and a terminal. No extra features were added to the language.

This chapter will explain the current state of the project right after Ricard’s final thesis.

3.1. Language features

Currently, the following C++ language features are supported:

- **Primitive types**: int, double, char and bool.
- **Strings**: Access or modification of individual characters is not supported.
- **Variable declaration and assignment**: Global variable declarations, variable shadowing and the const attribute are not supported.
- **Arithmetic operators**: binary +, −, *, /, % and unary + and −.
- **Logic operators**: binary &&, ||, unary ! together with their aliases and, or, not.
- **Comparison operators**: =, !=, >, >=, < and <=.
- **while and for loops**.
- **if and if-else statements**.
Function definitions and function calling.

- `cin`, `cout` and `endl`. They are partially supported, as they are simulated as language constructs instead of using stream objects and operator overloading as in C++.

### 3.2. Architecture

The general architecture of the C– development environment is depicted in Figure 3.1.

![C– Development Environment Architecture](image)

Figure 3.1: C– Development Environment Architecture

The backend module contains all the logic of the environment related to the compilation and interpretation of programs, whereas the frontend module contains the implementation of the website, which serves as an interface to the environment.

#### 3.2.1. Backend architecture

The backend is composed by 4 modules, which are represented in Figure 3.2.

![Backend Architecture](image)

Figure 3.2: Backend architecture

**Parser**

The parser module contains the language grammar, which serves as an input to Jison, a Bison based bottom up parser generator for Javascript. Jison uses the grammar definition to generate a scanner and parser for the language.

The scanner performs the lexical analysis, which consists in translating the stream of characters representation of the program to a stream of tokens representation. Each token represents the different kinds of words that can appear within the program, such as type names, identifiers, literals, etc.

The list of tokens is then used by the parser to generate an Abstract Syntax Tree (AST) which represents the parsed program. Because Jison is a bottom-up parser, the tree is built starting from the leafs and going up to the root.

Figure 3.3 shows a simple example of the process.
Extending a browser C++ simulator

Albert Segarra Roca

Figure 3.3: Scanning and parsing example
Semantics

The semantics module is in charge of walking through the AST to find semantic errors. Examples of semantic errors are invalid type uses, use of undeclared variables, redeclarations, incorrect function calls, etc.

Another task of the semantics module is to generate type casting instructions when there is a type mismatch and it can be solved by implicit casting. It does so by inserting new nodes in the AST structure.

Interpreter

The interpreter module performs the task of simulating the execution of a C− program. Its input is the processed AST resulting from the semantic analysis.

The basic components of the interpreter module are the following:

- **Runner**: The runner is in charge of executing the program instructions, which includes for statements, if statements, while statements, declarations, etc.
- **Stack**: The stack is used as a memory, and contains activation records, which are mappings from variable ids to variable values.
- **Function**: The function module maps function IDs to their corresponding AST, and modifies the Stack by pushing a new activation record on a function call or popping it when the function returns.
- **Expression**: The expression module is in charge of evaluating expressions.
- **IO**: This module handles the input/output operations that occur during execution.
- **vm-state**: This is a storage module that handles internal data of the virtual machine that is used to make the step-by-step execution possible.

Currently, the interpreter uses a Stack to store all the declared variables an its values. The Stack contains activation records, which are essentially mappings from variable ids to variable values. Each function call pushes a new activation record to the stack, which is popped when the function returns.

Debugger

The debugger module contains the implementation of the Debugger API, which includes different methods that can be used to indicate when to resume and pause execution. Currently, the following methods are supported:

- **Step out**: Keeps executing instructions until the current function returns.
- **Step into**: Executes a single instruction and pauses execution. If the current instruction is a function call, pauses at the first instruction of the function being called.
- **Step over**: Keeps executing instructions until the next instruction in the current function call stack level is reached.
3.2.2. Compilation and execution process

Figure 3.4 depicts the process that an input program goes through from the beginning of its compilation until its execution is finished.

First, the C– program is scanned and translated into a list of tokens. The list of tokens is then received as input by the parser, which uses a bottom-up parsing algorithm to produce an Abstract Syntax Tree (AST) that describes the program.

The AST is then semantically analysed to check for semantic errors and generate type casts, resulting in a new preprocessed AST, which is the equivalent of the executable of the program, ready to be interpreted.

Before the execution start, the interpreter receives the input of the program and the preprocessed AST, and starts running the program by recursively walking through the AST, evaluating expressions and executing instructions.

When the execution finishes, the exit status of the program and the three output buffers are returned. The output buffers consist of the standard error buffer, which is non-empty if some execution error occurred, the standard output buffer, which contains the output of the program generated by cout statements, and the interleaved buffer, which is the combination of the two previous buffers.

If any errors occur during one of the stages, the process stops and the error is returned. Errors can occur in any of the mentioned stages.
3.3. Website client

The website client is currently equipped with the following features:

- An **editor** with syntax highlighting and breakpoint setting support.
- A **terminal**, which serves as an interface to send input and receive output from the program. Compilation errors are also displayed in the terminal, and they are marked in bold to be able to differentiate them from execution output.
- A **variable grid**. It can be used to see the variables that are currently declared and their corresponding values. It can also be used to edit their value, although no validation is performed to ensure that the new values match the type of the variable.
- Execution and debug controls. They are used to start running the program, stop it, and execute debugging operations such as stepping in and out of functions.
- A status bar, which indicates the current status of the execution (running, paused or waiting for input).

3.4. Bugs and flaws

While analyzing the current implementation, it was found that there were almost no validation methods that ensured the correctness of the compiler, interpreter and debugger. There were only three unit tests, and they were not thorough enough.

For this reason, a test suite was created consisting of 988 solutions to Jutge.org problems, each with their input and expected output samples.

The test suite spotted some bugs that will be explained in this section and that lead to the decision of reimplementing all the changes that were made during Ricard’s thesis related to the implementation of step-by-step execution.

3.4.1. Main problem

The source of the bugs was related to a change that was made to enable step-by-step execution, which consisted in moving function calls and `cin` instructions out of expressions.

This meant that an expression such as:

```cpp
foo = 2 + f(bar);
```

was translated into:

```cpp
1 tmp = f(bar);
2 foo = 2 + tmp;
```

The same was done with `cin` statements, which can also be part of expressions.

Performing this kind of changes lead to different sorts of problems, for instance, the expression:
would cause the function call to be evaluated, even if it shouldn’t because of the lazy evaluation
behaviour of the **and** operator. Although this flaw was pointed out by Ricard in his thesis as
something that had to be fixed, it was only a simple example of the kind of problems that arise
from this instruction movement.

Another problem occurred with the following:

```c
while (f(bar)) {
  
}
```

This program would be translated to:

```c
tmp = f(bar);
while (tmp) {
  
}
```

This meant that the function call was only evaluated once right before the start of the loop.
The condition check at the first iteration would succeed, but further condition evaluations would
not call the function anymore and would rely on a temporary value which was not available
anymore, resulting in incorrect behaviour.

Because this instruction movement was key in the approach that was taken to implement step-by-step execution, a decision had to be made on whether to try to fix the issue or change the
approach. After some analysis, a decision was made to change the approach. This was backed
by several reasons:

- The solution to the problem seemed non-trivial.
- There could potentially be other problems arising from this instruction movement other
  than the ones exposed in this section.
- Given that almost no testing was done with the current version of the compiler during
  its development, there was no guarantee to the correctness of the software, and it was
  possible to find even more bugs that would delay the project severely.
- The version of the compiler that was left after the Compiler Construction Project success-
  fully executed all the programs in the test suite after a minor bugfix related to negative
  integer division.

The new approach to implement step-by-step execution and its actual implementation will be
detailed in Chapter 4.

### 3.4.2. Website

The current interface supposed an improvement to the previous version, but had some confusing
aspects and UI design problems:
The terminal did not write any separator character between the output of two program executions. This meant that there was no easy way to tell outputs apart from different executions.

The program’s output was only written to the terminal when it contained an endline character. This meant that a program which only wrote text without any endline would produce no output in the terminal.

The control buttons used to run, stop and debug the program were very small, making it difficult to press them.

The color used to mark the current line when debugging the program was a dark blue that had almost no contrast with the black editor background, which made it very hard to distinguish which was the current line at which the execution was paused.

For those reasons, a major interface redesign is performed in this project. It is detailed in chapter 13.
Part II

Design and Implementation
This chapter will explain all the changes that were made to the project architecture to enhance its maintainability and extensibility, while at the same time solving the current bugs originated from the previous approach to solving step by step execution.

4.1. New approach to step-by-step execution

The main problem with the implementation of the interpretation that was designed in the first version of the compiler and that continued in its second version is its recursive nature. The interpretation is done on the AST, which is a recursive data structure, and this enforces recursivity as the easiest solution to interpret it. Recursivity has the inconvenient that is very hard to pause, because some part of its state is scattered across the computer Stack and is lost when the function calls return.

This vision of the problem leads to a new approach to solving it. What if instead of storing the program in a recursive data structure, we stored it in a list instead? In the end, the AST can always be represented by a sequence of elements if we perform a tree traversal and enumerate its elements.

And if we think about it, that’s in fact how conventional C++ compilers work. They perform the recursive traversal during compilation, and that recursive traversal generates a list of instructions for each function of the program.

Now that we have represented the program with an iterative data structure instead, we can run the program iteratively by executing each instruction of the list one by one.

In order to be able to restart execution later we must store the state of the execution in a data structure. This data structure will be used as the means to stop and restart the execution without losing any information.
4.1.1. Flow control

Because the program must be translated into a list of instructions, and an instruction must be atomic, i.e., it must not have any other instruction within it, all flow control statements such as loops and if-then-else statements will have to be translated into branch instructions.

The reason why instructions must be atomic is that we should only have to store the minimal information to know what is the current execution state. If we need to pause in the middle of a while loop, ideally we should not need to remember that we were on a loop, because that complicates the storage of the execution state. Instead, we should only have to remember which instruction we were executing, which the branching scheme allows to do.

4.2. AST nodes as classes

The main bottleneck to the extensibility of the compiler at its current state is the semantics module. Currently, all the semantic analysis is kept in a single file, with a big switch statement inside a recursive function that performs an AST traversal.

This problem would worsen even more if we have to extend the semantics module to generate instructions, so we must find a solution to break the semantics module into components.

When it comes to modularize code, the first and most obvious solution that comes to a developer’s mind is the creation of classes. In this case a class can be used to represent each type of AST node, in such a way that the implementation of the semantic analysis and code generation of that type of AST node is handled inside the specific AST node class instead of having a big switch statement that decides which code to run depending on the type of node.

This is implemented in the C– compiler by creating a hierarchical structure of AST node classes, where the Ast class is the parent class for all AST nodes, and each type of node defines a subclass which implements the specific code for the semantic analysis for that node. This schema can also be used to create generic subclasses for very similar nodes, such as, for instance, the logic comparison nodes, where pretty much the only thing that changes from the implementation of one to the other is the operation that has to be performed.

C– even extends this solution to the interpretation stage. The same node class that implements the semantic analysis and code generation also implements the interpretation of the code that it generates, which will likely often be simply a single instruction which is an instance of the same node class with the necessary data for its interpretation.

4.3. Semantic analysis and instruction generation

Listing 1 shows the general structure of the Ast subclasses that implement the compilation and execution of the Ast node/instruction.

Every node is in charge of compiling its children nodes by calling their `compile` method, which every node defines, and which receives the compilation state as a first parameter. The compilation state stores information about the declared variables, the current size of the stack, the scopes, etc.

After compiling its children nodes, the node performs its semantic checks, computes its type and reserves a temporary variable for its result. Note that this step only applies for expressions,
class AstSubclass extends Ast
    compile: (compilation_state) ->
        [ children1, children2, .... ] = @children

        { type: typeLeft, instructions: insLeft, result: resultLeft } =
            children1.compile(compilation_state)
        { type: typeRight, instructions: insRight, result: resultRight } =
            children2.compile(compilation_state)

        check_for_semantic_errors()

        type = compute_type(typeLeft, typeRight) # type returned by the node.
            # In case of expressions returns
            # the type of the expression,
            # otherwise a VOID type is returned

        result = compilation_state.reserve_temporary_reference(type)

        instructions = produce_instructions(typeLeft, typeRight, insLeft, insRight)

        # Return the results of the compilation of this node
        # to the parent node
        return { type: type, instructions: instructions, result: result }

execute: (vm_state) ->
    dataForInterpretation = @children
    interpret(vm_state, dataForInterpretation)

Listing 1: General structure of an AST node subclass

other types of nodes such as loops simply generate branch instructions without returning any
result.

After reserving its result, the node generates its instructions by concatenating the children
instructions as needed and adding its instructions to the end.

Finally, the type, the instructions and the result of the node are returned as the result of the
compilation.

At the global scope, the function nodes will receive the instructions from the compilation of all
its children nodes, and will concatenate them to form the list of instructions of the function,
which is ready for an iterative interpretation.

4.3.1. Compilation state

The compilation state is in charge of storing the symbol table list, which is used to store information about the declared variables such as their memory address, and their type and specifiers.

The scoping of variables is implemented with a list of symbol tables, with the first element of the list being the symbol table of the global scope, and the next elements being the children scopes from function definitions and if-then-else statements or loops.

This allows to easily implement variable shadowing simply by adding a new symbol table every time a new scope is created and removing it from the list when exiting from the scope. Whenever a variable is used, it is searched in the symbol table list, starting from the last symbol table (deepest scope) all the way until the global scope (first element of the list). When the variable is found, the search is stopped and the information about the variable is returned. This allows for variable shadowing because the same variable name can be in two different symbol tables without creating any conflict within them.

The compilation state is also in charge of the generation and release of temporary variables for expression results. Because the compiler generates a sequence of instructions for an expression, the intermediate results of the expression must be stored somewhere. There is a memory space reserved for temporary variables, and the compilation state is the one in charge of issuing new temporaries and getting their space back, effectively keeping track of which addresses are available in the temporary memory space.

4.4. Runtime interpretation

As stated in section 4.1, an execution state data structure is necessary for storing the state of the execution. The following section explains the data that is stored by the execution state, also called virtual machine.

4.4.1. Execution state

The execution state holds the following structures and information:

- The **memory structure**. It holds all the information that the program can have access to, and it is explained thoroughly in Chapter 5.
- The **current function being executed**.
- Whether the execution has **finished**.
- The **instruction pointer**, which indicates the offset in the current function’s instruction list for the current instruction being executed.
- The **stack pointer**, which indicates the offset in the stack memory. It is increased on every function call by the total size of the local variables and parameters in the function, and it is decreased by the same amount when the function returns.
- The **function call stack**. This stack grows by one element on every function call, and gets reduced by one element every time a function returns. Every element contains the
instruction pointer, the stack pointer and the current function from the execution state before the function call, so that when it returns the pointers and the current function can be restored as the execution state.
In order to be able to properly implement arrays and pointers, a simulation of a memory was necessary to allow a more general and accurate simulation of the execution environment.

This change would also solve a problem present in the current implementation, which was the inaccurate representation of types. Currently the character type could hold values greater than its theoretical maximum for its expected size, which is 1 byte. This is because JavaScript is a dynamically typed language which does not allow typed declarations, so the character values are stored in a general number type which uses a different storage depending on which value is assigned.

The implementation of the memory will solve this problem thanks to a feature offered by the Javascript language, the Typed Arrays.

### 5.1. Javascript Typed Arrays

The Javascript Typed arrays are array-like objects which can be declared to be of a specific type. The following types are supported:

- Uint8Array (unsigned char in C++)
- Int8Array (signed char in C++)
- Uint16Array (unsigned short in C++)
- Int16Array (signed short in C++)
- Uint32Array (unsigned int in C++)
- Int32Array (signed int in C++)
- Float32Array (float type in C++)
- Float64Array (double type in C++)

The way these work, to allow for maximum flexibility and efficiency is by splitting its implementation into buffers and views.
A buffer is an object which represents a chunk of data, with no specific type. In order to access the data of the buffer, a view must be used. The view is instantiated on a specific buffer, and can be used to access the data as if it had a specific type. There is a view class for every type mentioned above.

This allows to define multiple views over the same underlying buffer, as depicted in Figure 5.1.

In this way, a memory can be implemented by Buffers and Views. Whenever we need to read or write into a given location in the memory which has a specific type, we can simply use a View with that type over the general memory Buffer to read or write the value.

5.2. Memory organization

The C– memory architecture was implemented by creating four different buffers, each representing a different memory location.

Because pointers are not dependent on whether the value is stored in the stack or the heap, a unified addressing scheme must be created to allow for an easy implementation of pointers.

The addresses are represented by a 32 byte value. The first $2^{31} - 1$ values are assigned for Stack addresses, whereas the rest of the addresses are assigned for Heap addresses. The highest weight bit indicates which memory the value is stored into. If it is 1, then the value is stored in the Heap, whereas if it is 0 the value is stored in the Stack.

5.2.1. Stack buffer

This buffer represents the Stack memory, where the local variables and the parameters from functions are stored. It has a total maximum size of 128MB.
5.2.2. Heap buffer

This buffer represents the Heap memory, where global variables and dynamic memory are stored. It has a total maximum size of 256MB.

5.2.3. Temporaries buffer

This buffer is used to store temporary variables created by the compiler. It has a total size of 64MB.

5.2.4. Return buffer

This buffer is used to store the last return value. It works as the %eax register in x86 assembly. It has a size of 8 bytes, the maximum size of any primitive type.

5.3. Memory location representation

The representation of memory locations is implemented by the Memory Reference class. It stores the following information:

- The address of the memory location
- The type of the value stored in the memory location

There is a subclass of the memory reference class for every single one of the four buffers explained in the previous section. Each of them is in charge of accessing the appropriate buffer when reading or writing into memory.

In the case of the stack memory references, the stack pointer is added to the value of the address before accessing the memory.

In all cases, the memory is accessed using the appropriate view for the type of the memory reference.

5.4. Reusing the memory

Because the allocation of the buffers by the Javascript runtime has an overhead, the memory has been developed in such a way that it can be reused from one execution to the next.

In this way, it only has to be allocated once, eliminating the allocation overhead in further executions.
The `const` specifier is used in variable and argument declarations to specify that the variable being declared should never be modified, and should remain with its initial value.

Because this feature has no effect during the execution of the program and is purely semantical, only semantic and grammar changes had to be implemented. This chapter explains those changes in detail.

### 6.1. Grammar changes

The only productions that need to be modified in order to implement the `const` specifier are those of variable declarations and function argument declarations. Listing 2 displays the current implementation of those rules.

```
1 declaration: [ 
2   o 'type declaration_body', -> new Declaration $1, $2 
3 ] 
4 
5 arg: [ 
6   o 'type id', -> new FuncArg $1, $2 
7 ]
```

Listing 2: Declaration and argument rules before the `const` implementation

By looking at the C++ standard at section 7.1 [dcl.spec], we can find that a simple declaration is mainly composed from a list of specifiers and a declarator (the `declaration_body/id` in the `declaration/arg` rules).

The `const` attribute is considered a declaration specifier, as well as the variable type. Specifiers can be written in any order, so both `const int x;` and `int const x;` are valid declarations.
This leads to the implementation of the const specifier by generalizing the current type variable. This can be done by creating a more general specifier-seq production, which contains a sequence of specifiers, each of them being either a type or a const specifier. Listing 3 shows the new implementation of the rules.

```
description: [
  o 'declaration_specifier_seq declaration_body', -> new Declaration $1, $2
]

arg: [
  o 'declaration_specifier_seq id', -> new FuncArg $1, $2
]

declaration_specifier_seq: [
  o 'declaration_specifier_seq declaration_specifier', -> $$$.push $2
  o 'declaration_specifier', -> [$1]
]

description_specifier: [
  o 'CONST'
  o 'type'
]
```

Listing 3: Declaration and argument rules after the const implementation

The type variable has been replaced with the declaration_specifier_seq variable, which is declared as a production that consists of a non-empty sequence of declaration specifiers, which can be either a type token or a CONST keyword.

The CONST keyword should also be added to the list of token rules. It is important to note that the rule should be positioned before the id token rule to ensure that it always gets tokenized as a const token, and not an id token.

```
$ /const\b/, 'CONST'
```

This concludes the grammar changes for the implementation of the const declaration attribute. Note that with this grammar implementation the following malformed declaration would be allowed, because there is no restriction on the number of specifiers that a declaration can have:

```
const int const int double x;
```

This kind of problems, however, should not be handled by the grammar and should be detected and reported by the semantic analysis instead. Section 6.2 will explain how this can be implemented.

### 6.2. Semantic changes

The required semantic changes are the following:
1. **Variable class**: This class holds information about a declared variable. In order to be able to detect and mark whether a variable has been declared with the const specifier, this information is stored in the class by adding a new property to it.

2. **Declaration AST Node**: We need to modify the compile method to account for the ast structure change. This includes looping through the specifier sequence list, checking that the following conditions are met:
   - There is exactly one type specifier.
   - There is at most one const specifier.

   If one of these conditions are not met, a compiler error is thrown indicating the situation. The variable is declared with the type specifier that was found and the boolean const property is set to true if the const specifier is present.

3. **Assignment AST Node**: If the assignment does not belong to a declaration (initialization of a variable), the compiler function must check that the variable being assigned does not have the const boolean property set to true. In case this condition is not met, a compiler error is thrown. Note that operations such as `a += b` or `++a` already produce an assignment node, so there is no need to re-implement this check in its corresponding class.
The current version of the compiler does not support global variables, but minor changes are required to support it after the changes that were explained in Chapters 4 and 5.

7.1. Grammar changes

In order to allow declarations in the global scope, we must change the prog production rule. Listing 4 shows the current implementation of the prog rule.

```
1 prog: [
2   o 'block_includes block_functions EOF', -> new ProgramAst $2
3 ]
```

Listing 4: Program rule before global variable implementation

As you can see from the rule, a program is considered to be a block of include preprocessor definitions followed by a block of function definitions.

Even if the block_includes variable is currently only specified so that programs with include and using definitions can compile successfully, the current implementation does not match the specifications of the C++ standard, given that an include declaration can be anywhere in the global scope, and not necessarily before all function declarations.

We can solve this problem and implement the global variable declaration at the same time by creating a general production top_level_decl_seq which contains a sequence of top level declarations. Each top level declaration can be either a conventional declaration, a function declaration or an include definition. Listing 5 shows the new grammar after the mentioned changes.
### 7.2. Semantic changes

The only semantic changes that are necessary to support global variables are modifying the `ProgramAst` AST node to account for the grammar changes.

Because the `top_level_decl_seq` produces a `List` node, which already handles the compilation of each of its elements, the only thing we need to do is compile the first children by calling its `compile` method.

This compilation will give as a result a list of instructions. This is because global variables can be initialized with expressions, which could possibly even be function calls that will have to be evaluated during the execution of the program.

The resulting instructions are added as part of the instructions of the **entry function**, a compiler-generated function from which the program execution starts, and the last instruction of which is a call to the **main** function.

No other changes are necessary, given that the `compilation-state` module does already assign heap or stack memory references to a variable declaration depending on whether it is declared in the global scope level or inside a function.
This is one of the main new language features that was implemented in this thesis, which allows to declare arrays and pointers and use the subscripting, dereference, and addressing operators.

8.1. Grammar changes

The grammar changes are mainly related to variable declaration, which now includes the ability to declare pointer and array types.

The operators related to pointer and array operations are also implemented, consisting of the subscripting, dereference and addressing operators.

Listing 6 shows the grammar changes for the pointer and array implementations.

The decl_var_reference production replaces the previous id production that was used for variable declarations, and implements what the standard refers to as a declarator.

A declarator must always have at least an id, which can be surrounded by array and pointer declarations, which can itself be embedded one inside the other.

We must also note that when declaring a pointer we must be able to make the distinction about whether the const specifier belongs to the pointer or the value being pointed. This is what is solved in the 4th rule of the decl_var_reference production, which allows to specify exactly one const specifier after each pointer declarator. This specifier is used to indicate that the pointer is const, rather than the value being pointed.

Apart from the declaration, we must also implement the three mentioned operators, which are added as rules to the expression production. Note that we define special precedence rules for the dereference and addressing operators to be able to distinguish their precedence from that of the multiplication and arithmetic & operations.

Finally, a change which is not included in Listing 6 is the adaptation that was made to assign-like expressions such as a = b, a += b, ++a, etc. Previously, the value that was modified in the assignment operation could only be an id. Because we must also be able to assign
values to arrays and pointers, the variable operand has been generalized to accept any kind of expression.

The check on whether the expression is valid to be assigned to or not is done in the semantic analysis using LVALUE and RVALUE expression classification, which will be explained in the next section.

8.2. Semantic changes

The semantic changes involve adding the new pointer and array types, changing the declaration AST node for the new possible declarations, implement the AST nodes for the new operators, classify expressions by LVALUE and RVALUE classification and implement pointer arithmetic.

8.2.1. New types

The implementation of the Array and Pointer types consisted in extending the base Type class that is used to represent types. This new classes allow to store new information specific to arrays and pointers.

Arrays

In the case of arrays, the dimension of the array and the type of the elements is stored as a property of the type.

Multidimensional arrays are supported by the recursive nature of the type definition, which allows for an element to also be an array.

Pointers

In the case of pointers, the type of the element being pointed is stored as a property of the class, together with the information about whether the value being pointed is const or not.

8.2.2. Declaration

In the case of the declaration, the second children of the AST is now a declarator. Each type of declarator (id, array, pointer) has their own AST node which is in charge of performing the necessary checks and passing its currently built type into the inner declaration.

The innermost declaration is the id declaration, which receives the final declaration type as a parameter and declares the variable with the received type.

Follows a list of the semantic checks that have to be performed by the declarators:

- Both pointers to strings and arrays of strings are not allowed, given that in the current implementation strings are not stored in memory.
- Declaring an array of void type is not allowed.
The dimension specified in the array declaration must be a positive literal of integral type. The restriction to literals is made because currently C++ lacks a constant expression evaluator, so even if the expression only consists of literal operands its value is not computed at compile-time. On the other hand, the declaration of arrays with dynamic size is forbidden by the standard.

Declaring an array of functions is not allowed.

### 8.2.3. Expression classification

An expression classification must be performed so that expressions such as

\[
2 = 5;
\]

are not allowed.

The way this is implemented is by classifying the results from expressions depending on whether they are RVALUES or LVALUES.

#### RVALUES

RVALUES are expression results which do not represent a memory location, and thus cannot be used as the left operand of an assignment. Such expressions include arithmetic expressions, value literals, etc.

#### LVALUES

LVALUES are expression results which do represent a memory location, so they can be used as the left operand of an assignment. These include ids, array accesses, dereferences, etc.

### 8.2.4. Operator implementation

Both the dereference and the subscripting operator have very similar implementations. Both have to check that the operand value is either of array of pointer type.

If this condition is met, the result of the expression is an LVALUE of the same type as the array/pointer element type.

In addition, the subscripting operator must check that the subscript value is of integral type.

In both cases, the result of the expression is a new type of memory reference, which instead of reading and writing to the address that it contains what it does is first read the value contained in the address and then performs the read/write operation on the new address, which is the value returned by the first read.

### 8.2.5. Pointer arithmetic

The + and − operators must be modified in order to correctly perform pointer arithmetic when one of the operands is a pointer.

They must also check that both operands are not pointers (it is not allowed to add up two pointers) and that the other operand is an integral value.
If those conditions are met, the result is a new pointer that consists in the sum of the pointer operand and the integral operand multiplied by the size in bytes of the type being pointed.
accessor: [
  o ' [ expr ]', -> $2
]

dimension: [
  o ' [ expr ]', -> $2
  o ' [ ]', -> new EmptyDimension
]

declaration_body: [
  o 'declaration_body , decl_assign', -> $$.push $3
  o 'declaration_body , decl_var_reference', -> $$.push $3
  o 'decl_assign', -> [ $1 ]
  o 'decl_var_reference', -> [ $1 ]
]

arg: [
  o 'type_specifier_seq decl_var_reference', -> new FuncArg $1, $2
]

decl_var_reference: [
  o 'id', -> new IdDeclaration $1
  o 'decl_var_reference dimension', -> new ArrayDeclaration $1, $2
  o '* decl_var_reference', (-> new PointerDeclaration $2), prec: "deref"
  o '* CONST decl_var_reference',
     (-> new PointerDeclaration(new ConstDeclaration($3))), prec: "deref"
  o '( decl_var_reference )', -> $2
]

expr = [
  .
  .
  .
  o '& expr', (-> new AddressOf $2), prec: "ref"
  o '* expr', (-> new Dereference $2), prec: "deref"
  o 'expr accessor', -> new ArraySubscript($1, $2)
]

Listing 6: Pointer and array implementation grammar changes
CHAPTER

9

NEW AND DELETE

The new and delete operators can be used to dynamically allocate and deallocate memory in the heap.

Implementing this feature was essential in order for C- to be able to create dynamically sized memory structures, because the current implementation follows the C++ standard in not allowing to declare dynamically sized arrays.

This chapter will explain the grammar, semantic and runtime changes that were made to the compiler and interpreter in order to support the new and delete operators.

9.1. Grammar changes

9.1.1. New operator

Listing 7 shows the changes that were made to the grammar in order to support the new operator. They are based on the specification of the new operator explained in section 5.3.4 [expr.new] of the standard [10].

It must be noted that the new-placement and new-initializer sections that are specified in the standard for the new declaration have not been implemented, restricting it to receive a single argument, a type identifier.

Because the new operator is an expression, its main production rule new_expr is added as an option to the expression production.

A new operator expression consists of the new keyword, followed by a type specifier sequence which may contain const and type specifiers, and a type declarator, which contains possible array and pointer type declarations. This type declaration may contain optional surrounding parenthesis.

Array of pointer type declarations are not allowed as arguments to the new operator, which is why the type_decl_imm rule is split into a nonpointer_type_decl, which only allows inner declarations to be array declarations.
The prec:"type decl" specifies the preference of the rule being defined. In this case, it is used to avoid a shift-reduce conflict that is thrown due to the following ambiguity:

```c
new int * i;  // syntax error: parsed as (new int*) i, not as (new int)*i
```

In this example we can see an ambiguity between the multiplication operator and the pointer declaration specifier. The ambiguity should always be resolved by shifting the '*' token, i.e parsing it as a pointer declaration specifier (new int*) i instead of a multiplication operation (new int)*i. This can be achieved by setting the priority of the type declaration rule to a priority lower than the multiplication operator, which is exactly what the prec:"type decl" does (see the full list of operators and its precedence in the Appendix [A]).

9.1.2. Delete operator

In the case of the delete operator, we perform similar changes, which are shown in Listing 8.

In this case we also need to add the delete_expr variable to the expressions production.

The delete_expr production is defined to be either an array deletion or a simple deletion. The operand is simply an expression which is expected to result in the pointer that has to be free’d.

9.2. Semantic changes

9.2.1. New operator

The following checks are performed during the semantic analysis of a new operator:

- An array declarator may only have the first dimension as a dynamic dimension. This is specified by the standard. This means that all dimensions starting from the second dimension must be constant values of integral type with a positive value. Because C- still doesn’t have a constant expression evaluator, the constant values must be literals.
- A string type specifier is not allowed for new expressions, given that in the current implementation strings are not stored in memory.
- A void type is not allowed as the type specifier for a new expression. This is specified by the standard.

If all the conditions are met, the new declaration results in a value with the following type:

- If the specified type is an array type with a given element type T, then the type of the result of expression is an RVALUE of pointer to T.
- Otherwise the type of the result of the expression is an RVALUE of pointer to the specified type.

The expression is an RVALUE.
9.2.2. Delete operator

For the delete operator, the only thing we need to check is that the operand type is either of array type or pointer type. The result of the expression is an RVALUE with void type.

9.3. Runtime changes

9.3.1. New operator

For non-array types the implementation of the new instruction execution simply allocates as much memory as needed for the type that was specified, and writes the initial address of the allocated memory to its result reference.

For array types, the new instruction must evaluate the first dimension value expression, which is stored as an operand of the instruction.

It must check that the resulting value is positive, throwing an execution error if this condition is not fulfilled. If the value is positive, the total allocation size is multiplied by the first dimension value, and the rest of the instruction execution is proceeded as with non-array types.

9.3.2. Delete operator

Currently, the implementation makes no distinction between the delete [] operator and the simple delete operator. The standard says that the first should only be used with pointers that were previously allocated with new [] (array types). The latter should only be used with pointers allocated with new, otherwise the use results in undefined behaviour.

Because the allocation and deallocation process does not know about the type of the value being allocated or deallocated, the current implementation simply will always deallocate all the memory that had been previously allocated with new. This conforms with the standard as the two correct uses (new[] with delete[] and simple new with simple delete) work as expected with this implementation.

The interpretation of the instruction simply deallocates the pointer received as operand, throwing an execution error if it is not a pointer that had been previously allocated with new or if it has already been deallocated.

9.3.3. Allocation and deallocation

The allocation and deallocation is performed using the malloc npm package, which implements a skip-list based memory allocator.

Because the allocator writes meta-information about the allocated blocks in the memory itself (in this case the heap) a reserved space must be left at the first addresses of the heap consisting of 272 bytes.

Also, because there is some space in the heap occupied by the global variables, this space must be allocated before the start of the execution of the program to let the library know that it shouldn’t use it for dynamic allocation. This is handled by the runtime module.
type_specifier_seq: [  # Note that this production was already implemented
  o 'type_specifier_seq type_specifier', -> $$ .push $2
  o 'type_specifier', -> [$1]
]

type_specifier: [  # Note that this production was already implemented
  o 'CONST'
  o 'type'
]

nonnull_dimension: [
  o '[' expr ']', -> $2
]

nonnuller_type_decl: [
  o 'nonnull_dimension', -> new NewArrayDeclaration(new NewDeclaration, $1)
  o 'nonnuller_type_decl dimension', -> new NewArrayDeclaration $1, $2
]

type_decl_imm: [
  o ' ', (-> new NewDeclaration), prec: "type_decl"
  o 'nonnuller_type_decl', (-> $1), prec: "type_decl"
  o '* type_decl_imm', -> new NewPointerDeclaration $2
  o '* CONST type_decl_imm', -> new NewPointerDeclaration(new ConstDeclaration($3))
]

type_decl: [
  o 'type_decl_imm', -> $1
  o '( type_decl_imm )', -> $2
]

new_expr: [
  o 'NEW type_specifier_seq type_decl', -> new New $2, $3
  o 'NEW ( type_specifier_seq type_decl )', -> new New $3, $4
]

expr: [
  .
  .
  .
  o 'new_expr'
]

Listing 7: Grammar changes for the new operator
delete_expr: [ 
  o 'DELETE [ ] expr',  -> new Delete $2, $4 
  o 'DELETE expr',  -> new Delete $2 
]

expr: [ 
  .
  .
  .
  o 'delete_expr'
]

Listing 8: Grammar changes for the delete operator
Compiler feedback is very important in the C– development environment, because its main target is educational, and programming beginners might find it more difficult to understand compilation errors.

For this reason, several improvements were made to the compilation error reporting by the compiler.

### 10.1. Specifying IDs and Types

Some of the compilation errors involve a specific variable or function, and refer to specific types.

In order for the programmer to quickly see which are the variables and types involved in the error, variable IDs and type names are provided whenever possible. Listing 9 shows an example of a compilation error that specifies both an ID and a type.

```
10:7: semantic error: Size of array 'arr' has non-integral type 'string'
    int arr["2"];
```

Listing 9: Compilation error with a reference to a variable ID and type

### 10.2. Line and column reporting

Line and column information is reported in every compilation error, at the beginning of the error message and with the format `line:column:`, so that the programmer can quickly locate the line and column of code that is causing the compilation error. When a compilation error spans more than one line, the first line is the one that appears in the error message. The same is true for the column.
This is achieved thanks to the Jison parser generator, which allows to add location information to each AST Node when it is constructed. In this way, when a compilation error occurs the line and column of the involved AST Node are added to the compilation error.

For further clarity and to help in certain cases where all the information about the compilation error can be found in the affected lines, the contents of the involved lines are also included in the compilation error, with the specific affected columns underlined with the ‘‘ character. This can be seen in listing 9, where the dimension of the array declaration is underlined to indicate the specific columns inside the line that caused the compilation error.

### 10.3. Error help descriptions

Because the short compilation error messages sometimes are not enough for beginners to understand what is wrong with their code, a longer description is included with every compilation error that explains the most common situations where the error could occur, and exposes a minimal code example with the compilation error. Listing 10 shows an example description that is included for the compilation error that fires when an undeclared variable is used.

This error occurs when trying to read or write on a variable which has not been declared in the same scope of use or any parent scope.

Example:

```cpp
int main() {
    if (true) {
        int x;

        ... // do something with x
    }

    x = 3; // Error! x has not been declared in this scope
    // or the parent scope, its declaration is
    // within a children scope created by the if statement,
    // and thus it is not visible here
}
```

Listing 10: Compilation error description for the ‘undeclared variable use’ compilation error
Because of the changes that were made during the architectural restructuring of the compiler and interpreter, the debugger implementation had to be completely changed. This chapter will explain the implementation details of the debugger and its API.

11.1. Breakpoints

In order for the debugger to be able to know when to pause execution, it must keep track of the current breakpoints that are set in the program.

Breakpoints can be added and removed even when debugging has started and the debugger is paused. For this purpose, two methods are exposed that allow to add and remove breakpoints, the `addBreakpoints` method and the `removeBreakpoints` method, which both receive a list of the breakpoints that have to be added or removed respectively.

In order to be able to associate the breakpoint with an instruction in the program, the compilation process adds line information to every instruction that is generated. With this information, the debugger finds the first instruction in the program that has the same line as the breakpoint being added, and marks it with a boolean property.

11.2. Continuing execution

The execution continuation is the main method in the debugger. It accepts different parameters that make it very general and allow every other debugging action to be defined in terms of this method.

The method keeps executing instructions until one of four conditions is met:

- The program finishes.
- The current instruction contains a breakpoint.
- The program is waiting for input.
A boolean condition function \( f(program\text{-}state) \rightarrow \text{bool} \) that is received as a parameter of the continue function returns true.

In any case, the current state of the program is returned, which allows to access its memory, see the current variables that are alive at the given execution point and be able to modify their values.

The boolean condition function defaults to a function that always returns false. This function can be used for the implementation of other debugging actions, as will be explained in the following sections.

### 11.3. Step over

As explained in the previous section, the continue method receives a parameter which consists of a boolean function that itself receives a single parameter, the program state, and returns a boolean value indicating whether the program should stop.

This can be used to easily implement the step over instruction only by defining the mentioned boolean function and calling the continue function.

In this case, when the step over function is called it stores the line of the current instruction in the execution, let’s call it \textit{initial-line}, and the current depth of the function call stack, let’s call it \textit{initial-depth}. The condition function is then defined to stop whenever the current instruction’s line given by the program state parameter differs from \textit{initial-line} and the current function call stack depth is less or equal than \textit{initial-depth}.

This condition implements the expected behaviour of the step over method, which is to not pause execution unless the next line in the current call stack level is reached. The non-strictness of the depth comparison in the condition function accounts for the case where the current line is the last line in the function, so there is no next line in the current call stack and the execution is paused in the next line of the parent call instead.

### 11.4. Step into

This debugging action steps inside of a function if the current instruction is a function call, instead it works in the same way as the step over function.

In this case, the condition function is even simpler. The only thing we need to check is that the line of the current instruction differs from \textit{initial-line}. The informal proof of the correctness of this condition follows:

If the current line changes it could be for one of three reasons:

- The current line was not a function call, it was not the last instruction in the function, means we just jumped into the next line in the current call.
- The current line was not a function call and it was the last instruction of the function, means either execution finished or we reached the next instruction of the parent function call.
- The current line was a function call, means we jumped onto the first line in the function being called.
These proof works under the assumption that different functions are defined in different lines of code. This is a reasonable assumption, so the debugger does not support cases where this condition is violated.

11.5. Step out

This debugging action should keep running instructions until the call stack depth is less than the current one, i.e. the function returned.

The boolean condition in this case simply consists in checking whether the current call stack depth is less than initial-depth.

11.6. Step

This debugging action simply always stops in the next instruction.

The boolean condition in this case is as simple as always returning true, regardless of the current program execution state.

11.7. Keeping track of alive variables

Because the interface must display the current variables that are alive at a given instruction, during the compilation stage a property is added to every instruction which contains a set of the current alive variables.
The command line interface (CLI) was mainly built with the purpose to be able to quickly test programs without having to build the whole project for the website. It is simple yet useful.

12.1. Arguments

It accepts two arguments, both of them being optional.

The first argument is the program source file path. If it is not specified, it defaults to a simple hello world program.

The second argument is a file containing the input that will be sent to the program. If it is not specified the input is set to the empty string.

Format specification:
```
cmm [program-source-path] [input-file-path]
```

Example command:
```
cmm program.cc input.txt
```

12.2. Output

Because the main purpose of the interface is for developer use, its output must show more information than just the output of the program execution.

For a successful program execution, the interface outputs the following:

- The AST resulting from the parsing.
- The instruction list for every function defined in the program, including the compiler-generated entry function.
- The exit code of the program.
The `stdout`, `stderr`, and **interleaved** outputs.

Because the AST and the instruction list are complex structures, a printing function must be defined that prints them in a pretty way so that they can be easily visualized.

For this reason a tree printing function was implemented that recursively prints the AST. Figure 12.1 shows the output for the execution of `cmm` with no arguments, which defaults to a Hello World program with no input.

If an error occurs during compilation due to a parsing or semantic error in the program, the CLI simply outputs the compilation error.
This chapter will explain the development of the new website interface, including the tools that were used and the new features that were implemented.

13.1. Frameworks and Libraries

Building a website interface from scratch is a very hard task, and there currently exist a lot of frameworks, libraries and tools that can be used to ease the process. This section will explain the external tools that were used to develop the website interface.

13.1.1. MeteorJS

MeteorJS [21] is a website framework that allows to build full-stack web applications with a frontend and a backend server that seamlessly connect via a DDP protocol.

The following features make it attractive to quickly build website interfaces:

- There is no need to add include tags for the JavaScript and CSS files in the HTML. MeteorJS automatically scans the project folder structure looking for Javascript, CSS and HTML files and automatically minifiesootnote{Minification is a process that reduces the size of the code files by shortening variable names and removing spaces, comments and endlines, among other techniques. The reduction in file size effectively speeds up the load time of the webpage and reduces server load.} joins and adds the corresponding includes in the HTML header.

  This feature allows to minimize redundancies, which is one of the key principles in the MeteorJS framework. In this way, when a new JS, CSS, or HTML file is created or renamed we do not have to remember to add a new include or rename an existing one. This reduces errors and speeds up development.

- Serving the website is as easy as running the meteor command inside the project folder. During development and when working in the local machine, whenever any code file is
changed the website is updated immediately reflecting the changes. This avoids having to continuously restart the server and refresh the webpage manually during development.

- There are numerous packages available for the framework and it is compatible with the npm package repository, which contains tons of libraries.
- There is a tool built for Meteor called meteor-up that allows to quickly deploy the website to a server with minimal configuration.

13.1.2. AngularJS 1.0

AngularJS [18] is a website client framework that allows to develop Single Page Applications (SPA) with a View-Model-Controller architecture.

Single Page Application are webpages which are served with a single main HTML file. Whenever the user navigates within the website, the specific content that has to be updated is loaded dynamically via AJAX requests, without refreshing the whole page. This speeds up navigation and provides a much better User Experience.

The main use of Angular is to host the website interface logic. It is very easy to build an interface that automatically updates itself as the model changes thanks to the two-way data binding feature of AngularJS. This feature tracks changes in the model variables so that when a variable value is changed the corresponding view element that displays the information of that variable is immediately updated. The opposite is also true, when the user modifies some input in the HTML view the model variable associated to that input is automatically updated with the new input value.

13.1.3. Angular Material

Angular Material [17] is a plugin for AngularJS. It is a styling library that can be used to quickly build beautiful user interfaces that are based on the Google Material Design guidelines.

It does also have support for making responsive websites that automatically adapt to the screen size to be able to support mobile phones and tablets.

13.1.4. jQuery Terminal Emulator

The jQuery Terminal Emulator [32] is a library developed by Jakub Jankiewicz used to provide a terminal interface in the webpage. It is very easy to setup and supports setting a prompt and being able to receive commands, which are sent from the interface either by typing into it or by pressing keyboard combinations. It does also support being able to write output into the terminal from the Javascript code.

13.1.5. ACE Editor

The ACE Editor [22] is another library used to provide an editor in the interface. The editor supports syntax highlighting of C++ among other languages and it displays line numbers. It does also understand indentations and it shows a cursor and highlights the current line of code being edited.
It has two interesting features for the debugger, which include being able to set breakpoints at a specific line of code and being able to highlight an arbitrary line of code, which is useful to indicate the current line of code when the debugger pauses the execution of the program.

13.2. Improvements

This section will explain the improvements that were made over the previous interface, although it was re-implemented completely due to the change in the frameworks being used.

Figures 13.2 and 13.1 show pictures of the previous and new website interfaces respectively.

13.2.1. Organization

The two main components of the website are the terminal and the editor.

It is very important for those two elements to have a good height (rather than width) in order to avoid scrolling as much as possible. Usually, both the terminal and the editor grow more in height than in width. This is because normally programmers have the discipline to set a limit in the line width to avoid having to scroll horizontally. Also, the terminal automatically wraps too long lines, so it will never grow horizontally.

For this reason, in the new interface the editor spans all the height of the website, and the terminal shares half of the height with a tabbed view that is used for secondary UI elements. If we compare it to the previous interface, this organization is much more efficient.

Previously, the debugger variable section spanned almost half of the website, even when it was being unused. On the other hand, the terminal was too wide, occupying the whole width of the
website and having a very low height, which forced the user to continuously scroll.

Another key feature in the new interface which was already present in the previous implementation is being able to resize the different components of the website. This allows the user to customize the layout to match his needs at any time.

Finally, the tabbed view in the top right corner is used as a container for secondary UI components, which will be detailed in the next sections.

13.2.2. Terminal

The terminal was already present in the previous interface implementation. However, it was not a conventional terminal, as it could only be used for sending input and receiving output.

On the other hand there was no prompt separating outputs of different executions, which made it hard to distinguish between outputs of different executions.

Command prompt

In order to cope with these problems, a command prompt (“>”) was included in the terminal. A welcome message was also included at the top of the terminal that explains which commands are allowed to be typed in it. Those include the compile, execute and debug commands, with their corresponding shortcuts c, e and d.

This allows for users that are used to type in the terminal to compile and run their programs to be able to keep using the same workflow. Also, because the prompt appears before and after
each execution output, it is very easy to tell apart outputs from different executions.

Sending commands is not the only input that the terminal can receive. During execution, it is used to send input to the program, and the Ctrl+D and Ctrl+C key combinations can be used respectively to send an end of input character to the program or kill it immediately.

**Endlinses problem**

Another problem that was solved in the new implementation of the terminal is the output of strings without endlines. In the previous implementation, when a program did not write any endline character the output never appeared in the terminal, even if some characters were written. This was due to a missing feature in the terminal library, which does not have any function to print in the terminal without printing an endline.

In the new implementation, this issue was worked around by changing the terminal prompt. Whenever the program outputs a character, it is added to the prompt string, effectively changing the current line. As soon as the program writes an endline character, the current prompt is written as a new line and the new prompt is set to the substring that remains after the endline character. When the execution finishes, the prompt string ”> ” is added to the end of the current prompt. This effectively implements the expected behaviour.

**Compilation errors**

Finally, in the previous implementation the compilation errors were highlighted with bold characters. This still didn’t make it easy enough to distinguish them from conventional output. The current implementation uses a feature of the terminal library to output the compilation errors in red text. In this way, the compilation error easily pops up over the rest of the terminal text, making it easier for the user to quickly realize that something went wrong.

13.2.3. Fixed input

This tab allows the user to specify a fixed input that will always be send to the program at the beginning of its execution. It avoids having to write the input again for every new execution.

13.2.4. Compilation help

This tab appears only when a compilation error occurs, and the interface automatically navigates to it when that happens.

It contains the compilation help, which is a detailed description of the compilation error that explains common situations where it could happen and offers a minimal example of a piece of code with the compilation error.

It is useful for cases where the information provided in the compilation error message is not enough to understand what is failing, specially for programming begginers which are not used to compilation errors.
13.2.5. AST view

This tab view shows a string representation of the AST resulting from the program parsing. It only appears after hitting the compile, run or debug button and only if the program compilation succeeded.

13.2.6. Instructions view

This tab shows a list of the instructions that were generated for each function of the program. Similarly to the AST view tab, it only appears after a successful compilation.

13.2.7. Buttons

The buttons used to control debugging, program compilation, execution start and killing the program have been redesigned to be bigger. This facilitates for the user to easily reach them without having to be very precise in pointing the mouse to them.

When the screen is wide enough, each button includes a text that describes the action it performs, which makes it even clearer for the user. On top of that, every button shows a tooltip text with an explanation of what it does whenever the user hovers the mouse over it.

In order to save space and make everything clearer for the user, only the buttons that are available for use at the moment are shown. For instance, the kill button only appears when the program is debugging or running, and the debugging buttons only appear when the program is paused in debug mode.

Finally whenever the compile, run or debug buttons are clicked the terminal is updated as if the corresponding command had been written on it, so that the user can keep track of its actions.

We must note that the compilation button was not present in the previous interface implementation. Although it is not strictly necessary, given that the run and debug already compile the code automatically, it is useful for when the user is still developing the program and wants to check that so far there is no compilation error, but doesn’t want to check the execution yet. Frequent compilation is a good programming practice and prevents accumulating compilation errors, which can slow down development.

13.2.8. Loading code examples

This is a feature that was already present in the previous website implementation, but it has been made more accessible and it has been improved.

The Download button at the right of the button bar can be used to display a menu with the available program examples that can be loaded. Two special examples, the MatrixMul and the Sudoku programs have been added to show off and demonstrate the new features supported by the compiler.

Also, not only the editor is updated with the code of the example, but also the fixed input tab content is set to an example input for the program, so that it can be executed rightaway.
13.2.9. Debugging

The debugging experience has been improved by using more clear breakpoint indicators and current line highlighting. Figures 13.3 and 13.4 show respectively the old and new breakpoint style and current line highlighting.

The new styling makes it much clearer and visible for the user at a glance what is the current line and where are the breakpoints in its program.

Variable value changing

When the debugger is paused, a new tab is shown in the tabbed view that displays the value of each currently visible variable. Figure 13.5 shows the mentioned view.
This view also allows to edit the values of the variables. The only two cases were a variable edit is not allowed are:

- **Arrays**: Currently not supported because of the extra interface elements that would be required to be able to see and edit individual elements. Currently the address of its first array value is shown instead of its elements.

- **Const variables**: An error message is shown indicating that the variable is declared const and cannot be modified.

On top of that, if the value that was entered by the user is not appropriate for the type of the variable, an error is shown indicating the situation. This ensures that the user cannot insert an incorrect value by mistake, compromising the execution of the program.

### 13.2.10. Status

In order for the user to be able to know what is the current state of the execution at any given moment, a status text and icon is included in the button bar. The status is updated whenever one of the following events changes occur:

- Compilation error
- Compilation success
- The program starts running
- The program is waiting for input
- The program is paused (only when debugging)
- The program has been killed

To help emphasize the state change, the status icon flashes repeatedly whenever the state is updated.
Part III

Evaluation
CHAPTER 14

CORRECTNESS EVALUATION

This chapter will explain the methodology that was followed and the tools that were developed and used during the project to ensure its correctness and automate the testing process.

14.1. Testing framework

Testing compiler software is hard and laborious, because it usually consists of a lot of components with complex interconnections. Additionally, having a strong and exhaustive test suite is essential for developers to be able to trust and rely on it.

For this reason, and with the hope to reduce the complexity of creating new tests and encourage them, a specialized testing framework was built to test the compilation and interpretation stages of the C– environment.

14.1.1. Mocha

The specialized framework is built on top of Mocha [11], a JavaScript test framework. Mocha works by scanning all JavaScript files within a special test folder, usually named test, where all the tests are implemented. It then proceeds to run each file and check whether its execution produces an exception, in which case the test fails.

Every test has an associated description, so that when tests are run it is easy to identify which ones failed and succeeded. Tests can also be grouped and organized in a hierarchical manner, being able to associate a description or name to each test group which is also displayed when the tests are run.

Figure [14.1] shows an example output when running a test suite with mocha. In the example there are two test groups, #indexOf() and #pop(), which both belong to the Array group and contain tests within them.
Test groups are specified by using the `describe` method that mocha exposes, which receives a description as a first parameter and a function as the second parameter. The function must contain all the code of the tests within the group, which includes possible nested `describe` calls that specify children test groups.

A test can be specified by using the `it` function, which also receives a description of what is being tested as the first parameter, and a function as the second parameter, which performs the actual test that should throw an exception if it fails.

Listing 11 shows an example of how the tests shown in Figure 14.1 could be implemented. Note that `assert.equal` throws an exception if the values specified are not equal.

```javascript
assert = require 'assert'

describe('Array', ->
    describe('#indexOf()', ->
        it('should return -1 when the value is not present', ->
            assert.equal([2,3].indexOf(25), -1)
        )
    )

    it('should return the correct index when the value is present', ->
        assert.equal([2,3].indexOf(3), 1)
    )

    describe('#pop()', ->
        it('should remove and return the last value', ->
            array = [1,2,3]
            assert.equal(array.pop(), 3)
            assert.equal(typeof array[2], 'undefined')
        )
    )
)
```

Listing 11: Mocha tests description example
14.1.2. C– test framework

The design of the specialized test framework was inspired by looking at how the GCC compiler describes its tests.

By looking inside its public repository, I found out that each test was described by a C++ program, which itself contained metadata about how the test should behave in the form of code comments.

I took that idea and designed a test framework with the same principles.

Advantages/disadvantages

A drawback related to this solution is that the tests are not completely isolated from each other. For instance, the function module will be partly checked in every test related to the interpreter runtime, because all valid C++ programs must define a main function (except for headers, which are not supported yet). This means that when a test fails it’s harder to track the origin of the bug because it could come from features other than the one that is mainly intended to check in the test. A partial solution to this problem is to test the most basic features (functions, input/output, etc.) first, so that if these features fail we can ignore more complex tests until the simpler ones are solved.

On the other hand, this solution has the advantage that it is easy to implement and maintain, because it is independent from the actual compiler implementation (language, code structure, etc.), so if the compiler’s codebase is re-factored there’s no need to update the tests. For this reason, and because easiness in the creation of tests is very important, this solution was chosen for the C–’s test framework.

Moreover, this does not prevent other kind of tests such as unit and integration tests targeted at specific modules, which do not have the mentioned disadvantages, to be implemented together with the testing framework in the future.

Test properties

In the C– test framework, every test is also a C++ program with .cc extension, which defines a header at the top of the code file with the following properties:

- **Description**: A description of what feature specifically does the test check. Compulsory for every test.

- **Expected compilation error**: An integer code which identifies the specific error that should be thrown during compilation. Zero is specified when no compilation error is expected. This property can be omitted, in which case it defaults to zero.

- **Expected execution exit status**: Again an integer code which identifies an exit status which is expected when running the program. Note that every execution error such as division by zero or stack overflow has a specific exit status code which identifies it. This property can be omitted, in which case defaults to zero, which means that no execution error is expected.

Listing 12 shows an example test with its properties and code.

The expected compilation error and expected execution exit status properties mark the distinction between compilation and runtime tests. Compilation tests check that the compiler outputs
Extending a browser C++ simulator

```cpp
/*
 * description{Checks that modifying const variables is not allowed}
 * compilation-error{5004}
 */

const int SIZE = 3;

int main() {
    SIZE = 5;
}
```

Listing 12: Example C– test for the const declaration attribute

an error when the program contains a semantic or parsing error, whereas runtime tests check that the execution of the program behaves as expected, producing the correct output and/or exit status.

Input/output

As I just mentioned, runtime tests can also have associated pairs of (input, expected output) files. Input files have the .in extension, whereas output files have the .out extension. The test program is run for each input/output pair associated with it, setting its input as the content of the input file and checking that its output matches the content of the output file.

Associations

Inputs are associated with outputs by their base name, which should be equal. Each input/output pair is associated with a program (test) by its name. For instance, if a test file is named const.cc then its associated input/output files should be named const-*.in and const-*.out. There is an exception: tests are grouped with folders, and if there is only one program within the folder and it is named program.cc then all input/output files within that folder are assumed to be associated with it, regardless of their name.

Of course, output files do not need to have an input file associated with it, because sometimes the program does not need any input. In that case the input of the program is set to the empty string. The same is true for input files, in which case the output of the program is not checked.

Implementation on top of Mocha

The testing framework works by scanning the structure of the test folder. It maps the folder structure into a mocha test structure, in such a way that every folder is mapped into a mocha test group with its description being the folder name, and a test file is mapped into a mocha test with its description being the test description property that was previously mentioned.

The content of each test file is then is parsed to obtain the description, expected compilation error and expected exit status, which is later used inside the test function to assert the correctness of the compilation and/or execution.
14.1.3. Errors

The testing framework checks for errors in the definition of tests, such as missing a description property, not defining a header, having duplicated test names or having input/output files with no associated test. This makes it easy for new people getting involved in the project to spot mistakes in their test definitions.

14.2. Test suites

The test framework detailed in the previous section allows to define test suites with ease and speed, avoiding redundancies. This section will briefly expose the actual test suites that were implemented to test the environment.

14.2.1. Compilation tests

This test suite is used to check that the semantic and parsing stages of the compilation work correctly.

Parsing

No parsing tests have been implemented during this thesis, although they are obviously important and should be included in next versions of the compiler. This would probably require adding some extra features to the testing framework to be able to assert parsing errors at specific lines.

Semantics

A total of 86 semantic tests have been implemented, which include at least one test for each compilation error that is contemplated by the compiler.

14.2.2. Runtime tests

Two distinct types of runtime tests have been implemented, which are detailed in the following sections.

Targeted tests

These tests use the testing framework described in section 14.1. There are a total of 55 tests, including all the executions for every input/output pair of each program. This tests are targeted at testing specific features, such as pointers, arrays, arithmetic operations, etc.

Real-world program tests

These are generic tests, which are not targeted at a specific feature. They are used to complement the targeted tests with real world programs that may spot bugs which they still do not cover.
They do not use the testing framework specified above, because they all compile and produce no execution errors. They are simply defined by three files: the program, the input file and the expected output file.

Currently there are two sample tests, a Sudoku solving program and a Matrix multiplication program.

On the other hand, there is a test suite with 988 test programs which were taken from my solutions to Jutge.org problems. A script was used to download all the solutions and input/output samples, from which the input/output/program file triples were generated. Because some of the programs use C++ language features which are not supported yet by C–, when a program does not compile it is ignored by the test script.

14.3. Summary

The test suites were extremely useful and managed to spot a large number of bugs, including initial bugs that were present before the start of the project and bugs that were introduced during the implementation of new features.

The fact that the tests were updated and run very often right after implementing new features allowed to quickly spot and fix mistakes, which lead to a fast and reliable development process.
While execution performance is not the main objective of the C– project, it wouldn’t be wise to ignore it completely, because there might be some weak performance points which can be spotted by performance tests. Also, it is interesting to see how it compares to previous versions of the project to detect whether a performance bottleneck was introduced.

In this chapter the performance of the improved compiler and interpreter developed in this thesis (C– 2.0 from now on) will be compared to that of other compiled and interpreted languages. Specifically, it will be compared to the following:

- **C– 1.0**, the initial version of C– that this thesis extended (See [https://github.com/jutge-org/cmm/releases](https://github.com/jutge-org/cmm/releases) for the exact Git commit that is tested).
- **Python v3.5.2**, the popular interpreted scripting language.
- **NodeJS v7.7.3**, a Javascript runtime environment which uses the *Chrome V8* just-in-time compiler.
- **C++11** compiled with *GCC v6.2.0* and no optimizations (-O0).

C– release 1.1 as it figures in the C– repository releases will not be compared because it crashed on one of the benchmark problems, and it was already shown to have a very similar (slightly worse) performance than C– 1.0 by Ricard Gascons [12].

In order to carry out the comparison, the implementation of the solution to four different problems is used. Those include:

- **The primality test** problem, with a simple loop solution.
- **The Fibonacci** problem with the straightforward recursive solution.
- **The Collatz sequence** problem.
- **The matrix multiplication** problem for square matrices, with the straightforward $\theta(n^3)$ solution.

Note that the first three problems were taken from [12], although the solutions are different here because the size parameter $N$ is an input to the program.
For every problem, the execution time of the whole solution program is measured for different values of the input parameter \( N \). The measurement for Python, C++11 and NodeJS is made using a program similar to Linux’s `/usr/bin/time` that I developed for other projects, which has microsecond precision. For C– 1.0 and 2.0, the measurement is made using node’s `process.hrtime()` function, with nanosecond precision, and only the execution time is measured, the compilation time is not included, as with C++11. All execution time measures are averaged over three runs.

Someone might argue that having different timing methods for the C– versions and the other languages might affect the results precision, but since the benchmarks have large enough execution times those discrepancies in time measurement are too small to affect the result in a meaningful way.

The benchmarking script and the algorithm implementations for every language can be found in the appendix. Also, for easy reproducibility you can find the whole benchmark setup in the following Git repository [https://github.com/albertsgrc/c--benchmark](https://github.com/albertsgrc/c--benchmark). The computer that was used for the benchmarks is a 2016 15’ Macbook Pro with an Intel Core i7 6700HQ Processor and 16GB of LPDDR3 2133MHz RAM.

With regards to benchmarking methodology, for precision and reproducibility maximization the minimal number of applications necessary to run the benchmark (OS, Terminal) were executing during the tests, in order to minimize its effect on the execution time of the benchmark process, which could be otherwise affected by the OS taking it out of the CPU, causing overhead due to context changes and cache misses due to cache pollution by other processes.

### 15.1. Primality test

The primality test algorithm is used to test whether a given number is prime. In this benchmark, for an input parameter \( N \) we perform the primality test for all numbers between 1 and \( N \cdot 100000 \).

The primality test algorithm used in the implementation has complexity \( \theta(\sqrt{n}) \) in the worst case for an input number \( n \), performing a loop until the square root of \( n \) is reached or a divisor of \( n \) is found.

Table 15.1 shows the results of the benchmark, with the execution times shown in milliseconds. Figure 15.1 shows a plot of the execution times as a function of \( N \).

---

1 Note that \( n \), the input of the algorithm is not the same as \( N \), the input of the program.
Table 15.1: Execution times [ms] for the primality test problem

<table>
<thead>
<tr>
<th>N</th>
<th>C– 2.0</th>
<th>C– 1.0</th>
<th>C++11</th>
<th>NodeJS</th>
<th>Python3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3528</td>
<td>3738</td>
<td>7</td>
<td>52</td>
<td>266</td>
</tr>
<tr>
<td>2</td>
<td>8965</td>
<td>9357</td>
<td>15</td>
<td>63</td>
<td>624</td>
</tr>
<tr>
<td>3</td>
<td>15425</td>
<td>16063</td>
<td>26</td>
<td>74</td>
<td>1058</td>
</tr>
<tr>
<td>4</td>
<td>23004</td>
<td>23783</td>
<td>37</td>
<td>84</td>
<td>1620</td>
</tr>
<tr>
<td>5</td>
<td>31113</td>
<td>32016</td>
<td>47</td>
<td>95</td>
<td>2169</td>
</tr>
<tr>
<td>6</td>
<td>40118</td>
<td>40879</td>
<td>60</td>
<td>110</td>
<td>2821</td>
</tr>
<tr>
<td>7</td>
<td>49452</td>
<td>50547</td>
<td>72</td>
<td>123</td>
<td>3453</td>
</tr>
<tr>
<td>8</td>
<td>59803</td>
<td>60344</td>
<td>84</td>
<td>137</td>
<td>4174</td>
</tr>
<tr>
<td>9</td>
<td>70864</td>
<td>70971</td>
<td>100</td>
<td>151</td>
<td>4951</td>
</tr>
<tr>
<td>10</td>
<td>80771</td>
<td>82009</td>
<td>112</td>
<td>175</td>
<td>5695</td>
</tr>
</tbody>
</table>

Figure 15.1: Execution time vs parameter N comparison for the primality test problem

As we can see in Figure 15.1, the performance of C– 2.0 and C– 1.0 is very similar for all input values, with the C– 2.0 performance being marginally better.

We can also see that its performance compared to the other languages is very poor, although very decent considering that C– is interpreting C++ and is itself running on NodeJS, which uses just-in-time compilation, so there are two levels of abstraction over the machine code that produce a very high overhead.

15.2. Fibonacci

The straightforward Fibonacci algorithm implementation has a complexity of $O(2^n)$ for an input value $n$, producing a very large number of function calls. This means it can be very useful to test the function call performance of the interpreter.
In this case, the program simply runs the straightforward Fibonacci algorithm with input $N$.

Table 15.2 shows the results of the benchmark, with the execution times shown in milliseconds. Figure 15.2 shows a plot of the execution times as a function of $N$. Note that for the C–implementations and the Python implementation large input values are not tested because the behaviour can already be seen from the included values and the execution time would be very large, given the exponential complexity.

<table>
<thead>
<tr>
<th>$N$</th>
<th>C– 2.0</th>
<th>C– 1.0</th>
<th>C++11</th>
<th>NodeJS</th>
<th>Python3</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>582</td>
<td>1229</td>
<td>2</td>
<td>49</td>
<td>72</td>
</tr>
<tr>
<td>30</td>
<td>6289</td>
<td>13375</td>
<td>6</td>
<td>58</td>
<td>379</td>
</tr>
<tr>
<td>35</td>
<td>69855</td>
<td>148412</td>
<td>64</td>
<td>154</td>
<td>3663</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>644</td>
<td>1207</td>
<td>40569</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td></td>
<td>7096</td>
<td>12579</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>78370</td>
<td>154234</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 15.2: Execution times [ms] for the Fibonacci problem

As we can see in Table 15.2, C– 2.0 has a much better performance than C– 1.0, with it being at least twice as fast. This can also be noted from Figure 15.2, where we can see that the execution time starts increasing quickly a bit later. Again, the performance compared to other languages is poor but pretty decent, as we can see that Python’s execution time starts increasing quickly only for $N \approx +7$ with respect to C– 2.0.

This increase in performance seen in C– 2.0 may be due to the fact that C– 1.0 had a very inefficient interpretation of function calls, where a new activation record was pushed in the stack on every function call, and every argument and local variable had to be defined again in the stack for every function call. In C– 2.0 the argument declarations and local variable declarations are translated to memory addresses and do not produce any instruction, which reduces the overhead in function calls.
15.3. Collatz sequence

This benchmark computes the Collatz sequence, a convergent series which is conjectured to always converge to one. The sequence is computed starting from a value \( n \) until it reaches 1. In this case, we repeat this process for values of \( n \) from 1 to \( N \), the input value of the benchmark.

Although the length of the Collatz sequence for a starting number \( n \) is unknown, because it has not yet been shown whether the sequence reaches 1 for all \( n \in \mathbb{N} \), for the values tested in the benchmark we’ll assume for practical purposes that it is constant with respect to \( n \), given that the largest number of steps to reach 1 for \( n \leq 50000 \) is 323 [2]. This means that the complexity of the benchmark for input \( N \) can be assumed to be \( \theta(N) \), because the Collatz sequence is computed \( N \) times.

This program, similar to the Fibonacci program has a very large number of function calls, because the \texttt{collatz} function is called for each step of the series.

Table 15.3 shows the results of the benchmark, with the execution times in milliseconds. Figure 15.3 shows a plot of the execution times as a function of \( N \).

<table>
<thead>
<tr>
<th>( N )</th>
<th>C- 2.0</th>
<th>C- 1.0</th>
<th>C++11</th>
<th>NodeJS</th>
<th>Python3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>2943</td>
<td>4952</td>
<td>7</td>
<td>52</td>
<td>288</td>
</tr>
<tr>
<td>15000</td>
<td>4516</td>
<td>7718</td>
<td>12</td>
<td>54</td>
<td>424</td>
</tr>
<tr>
<td>20000</td>
<td>6204</td>
<td>10618</td>
<td>19</td>
<td>55</td>
<td>585</td>
</tr>
<tr>
<td>25000</td>
<td>8019</td>
<td>13635</td>
<td>19</td>
<td>56</td>
<td>735</td>
</tr>
<tr>
<td>30000</td>
<td>9724</td>
<td>16573</td>
<td>28</td>
<td>58</td>
<td>872</td>
</tr>
<tr>
<td>35000</td>
<td>11401</td>
<td>19687</td>
<td>27</td>
<td>59</td>
<td>1039</td>
</tr>
<tr>
<td>40000</td>
<td>13071</td>
<td>22731</td>
<td>32</td>
<td>61</td>
<td>1186</td>
</tr>
<tr>
<td>45000</td>
<td>14875</td>
<td>25922</td>
<td>36</td>
<td>62</td>
<td>1345</td>
</tr>
<tr>
<td>50000</td>
<td>16631</td>
<td>29078</td>
<td>40</td>
<td>64</td>
<td>1499</td>
</tr>
</tbody>
</table>

Table 15.3: Execution times [ms] for the Collatz problem

<table>
<thead>
<tr>
<th>( N )</th>
<th>C- 1.0</th>
<th>C++11</th>
<th>NodeJS</th>
<th>Python3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.59</td>
<td>420.43</td>
<td>56.6</td>
<td>10.22</td>
</tr>
<tr>
<td>15000</td>
<td>0.59</td>
<td>376.33</td>
<td>83.63</td>
<td>10.65</td>
</tr>
<tr>
<td>20000</td>
<td>0.58</td>
<td>326.53</td>
<td>112.8</td>
<td>10.61</td>
</tr>
<tr>
<td>25000</td>
<td>0.59</td>
<td>422.05</td>
<td>143.2</td>
<td>10.91</td>
</tr>
<tr>
<td>30000</td>
<td>0.59</td>
<td>347.29</td>
<td>167.66</td>
<td>11.15</td>
</tr>
<tr>
<td>35000</td>
<td>0.58</td>
<td>422.26</td>
<td>193.24</td>
<td>10.97</td>
</tr>
<tr>
<td>40000</td>
<td>0.58</td>
<td>408.47</td>
<td>214.28</td>
<td>11.02</td>
</tr>
<tr>
<td>45000</td>
<td>0.57</td>
<td>413.19</td>
<td>239.92</td>
<td>11.06</td>
</tr>
<tr>
<td>50000</td>
<td>0.57</td>
<td>415.78</td>
<td>259.86</td>
<td>11.09</td>
</tr>
<tr>
<td>( \mu )</td>
<td>0.582</td>
<td>394.71</td>
<td>163.46</td>
<td>10.85</td>
</tr>
<tr>
<td>( \sigma )</td>
<td>0.008</td>
<td>36.02</td>
<td>70.17</td>
<td>0.30</td>
</tr>
<tr>
<td>( \sigma_{\text{norm}} )</td>
<td>0.014</td>
<td>0.091</td>
<td>0.429</td>
<td>0.028</td>
</tr>
</tbody>
</table>

Table 15.4: Speedup vs C- 2.0 for the Collatz problem
The resulting plot in Figure 15.3 again shows that C– 2.0 has a better performance compared to C– 1.0 with function-call intensive programs. The slope of the execution time as a function of N is significantly smaller for C– 2.0.

When comparing it with other languages, the results are very similar to the other benchmarks, with C– 2.0 having a significantly worse performance as is expected.

Because for practical purposes this benchmark can be assumed to have linear complexity with respect to N, as seen in the plot and contrary to all other benchmarks, it is suitable to compare and evaluate the performance of C– 2.0 with respect to the other languages by looking at their speedup relative to C– 2.0 for every value of N. Table 15.4 shows the speedup relative to C– 2.0 for all other languages and for all values of N.

As we can see, C– 2.0 is almost twice as fast as C– 1.0, and is only 11 times slower than Python3.

In general, the speedups are more or less constant for C– 1.1 and Python3. C++11 shows varying speedups because its execution time is very low and thus is subject to more variability due to external factors, as can be seen from its $\sigma_{\text{norm}} = \frac{\sigma}{\mu}$. In the case of NodeJS, it has an increasing speedup because it uses just-in-time compilation, and thus has an initial overhead which pays off only when the program performs heavy computations (larger values of N).

### 15.4. Matrix multiplication

This benchmark consists in computing the matrix multiplication of two square matrices with size $N \times N$. The complexity of the solution’s algorithm is $\theta(n^3)$, performing three nested loops each with length $N$.

The main thing that differentiates this benchmark from the others is that it is heavy in memory accesses, because it is continuously accessing the matrix structure. Note that because the pro-

---

2This value is useful to compare standard deviations when the mean of the variables being compared cannot be assumed to be equal, as in this case.
gram uses pointers and memory allocation in order to store the matrix, it will not be compared against C– 1.0 because those features are not supported in such version of the compiler.

Note also that the matrices are all initialized with zeroes, but this does not matter for the benchmark because the execution time does not depend on the actual values stored in the matrix.

Table 15.5 shows the results of the benchmark, with the execution times in milliseconds. Figure 15.4 shows a plot of the execution times as a function of $N$.

<table>
<thead>
<tr>
<th>$N$</th>
<th>C– 2.0</th>
<th>C++11</th>
<th>NodeJS</th>
<th>Python3</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>389</td>
<td>2</td>
<td>56</td>
<td>232</td>
</tr>
<tr>
<td>100</td>
<td>2976</td>
<td>6</td>
<td>77</td>
<td>802</td>
</tr>
<tr>
<td>150</td>
<td>9927</td>
<td>28</td>
<td>135</td>
<td>2318</td>
</tr>
<tr>
<td>200</td>
<td>23399</td>
<td>55</td>
<td>246</td>
<td>5292</td>
</tr>
<tr>
<td>250</td>
<td>46755</td>
<td>99</td>
<td>410</td>
<td>10187</td>
</tr>
<tr>
<td>300</td>
<td>80282</td>
<td>146</td>
<td>662</td>
<td>17463</td>
</tr>
<tr>
<td>350</td>
<td>127603</td>
<td>245</td>
<td>1019</td>
<td>27757</td>
</tr>
</tbody>
</table>

Table 15.5: Execution times [ms] for the Matrix multiplication problem

The resulting plot (Figure 15.4) shows similar results to the previous experiments, although in this case differences are amplified due to the cubic complexity of the algorithm.

We must note from table 15.5 and looking at the algorithm that, for instance with $N = 100$ a total of $3 \times 10^8 = 3 \times 10^6$ memory accesses are performed, and the execution time of C– 2.0 is only of approximately 3s. This gives an approximate average of $\frac{3s}{3 \times 10^8 \text{acc}} = 1 \mu s$ per memory access, or a million memory accesses per second. This is far enough for the main intended purpose of C–, which is educational use.

$^3$Note that the GCC compiler does not perform any optimization with O0, so it cannot optimize the loops assuming values are zero.

Figure 15.4: Execution time vs parameter $N$ comparison for the Matrix multiplication problem
This goes to show that the new features introduced in C– 2.0 do not have any major performance bottlenecks related to memory accesses.

15.5. Summary

This results demonstrate that C– 2.0, despite being a more general implementation of a C++ simulation than previous versions of C–, does not show worse performance. Not only that, but it does even show much better performance for programs with a large number of function calls.

This can be attributed to the fact that some computation that was previously performed during runtime is now performed at the compilation stage, specially due to the instruction generation and the translation of variable declarations into memory addresses, without any instruction being generated for them. This compensates the performance penalty that other changes incur, such as the interpretation of loops with branches instead of actual Javascript loops, or the memory simulation.

By looking at its performance with respect to other languages, we can also see that even with the new features that were implemented in this thesis, its performance is good enough for its target of use, which is mainly educational.
This project does not have any specific law/regulation with regard to its actions, given that C– is mainly a software development project, and its website does not use cookies or deal with user passwords or personal user data.

The software is released under the MIT License:

**Copyright 2017 Albert Segarra Roca, Ricard Gascons Gascón and Pau Oliver Farreny**

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
This final chapter presents a summary of the thesis and possible future work that could be done to improve the C– development environment. Finally, the author briefly exposes his personal conclusions about the project.

17.1. Goal validation

This section will review the goals that were set at the beginning of the project, to see how the project development accomplished them.

- **Extend the subset of supported C++ features**: The project implemented several new features, which include pointers, arrays, global variables, variable shadowing, the const attribute and the new and delete operators for memory allocation and deallocation.

- **Increase the maintainability and extensibility of the compiler**: A major compiler architecture restructuring was performed to modularize its components, thus improving its maintainability and extensibility. The addition of an automated testing framework with big test suites also improved the extensibility by providing an easy way to check the correctness of the compiler.

- **Provide better compilation feedback**: Line and column reporting was implemented for the compilation errors, making it easier for the user to quickly find which part of the code originated the error. Extended error descriptions were written for each compilation error to provide a better explanation of the errors for beginners.

- **Improve the website interface**: Numerous improvements were made to the website interface, which include a reorganization of its components, an enlargement of the buttons for better access and user experience and the implementation of new features such as fixed input and terminal commands.

- **Improve the validation tests to ensure the correctness of the compiler**: A specialized test framework was built for the compiler to be able to easily create new tests. A total of 1131 tests were added, including a test suite with 988 Jutge.org problem solutions, 57 runtime tests and 86 compiler semantic tests.
17.2. Future work

Looking into future thesis and people that might get involved in this project, I would list the following features and improvements as the most important and interesting to implement:

- **Structs and classes**: This is crucial to be able to implement other features of the language such as vectors, strings and properly implement the cin and cout objects.

- **Libraries**: Having libraries is also important to be able to modularize C– programs, and implement the C++ standard library.

- **Other primitive types**: Implementing the short type, the float type and the unsigned variants should not be hard now that a typed memory has been implemented.

- **Compilation warnings**: These would be very useful to warn of potential bugs in the program to programming begginers. Examples of warnings could be out-of-bound array accesses or variable shadowing.

17.3. Personal conclusions

This project supposed my second contact to the C– project and the compiler construction branch of computer science. I have always been fascinated by computer languages and wondered how compilers were implemented, specially during the first courses in the University, where they looked to me like something I would never be able to understand because of its apparent complexity.

Thanks to being part of this project and having developed this thesis, I have been able to gain vast knowledge about compiler theory and use concepts from a lot of different subjects in the degree to build creative solutions to the problems that I faced during its development.

On the other hand, I have learned the hard way that building compilers is a very hard task, which requires being very meticulous and researching on language reference documents to find corner cases that need to be taken in account during the compilation and interpretation stages. By doing that, I have learned about a lot of constructions of the C++ language that I didn’t even know existed, and that made me realize how big a language it is.

Finally, I have realized that I probably have been to generous when setting the objectives of this project, and I have done way more work than expected for a final thesis. This has mainly been because of my motivation for the development of the project and my perfectionist personality, but I am very positive about the extra experience that I gained and that I will be able to use for future projects in my life.


Extending a browser C++ simulator

Albert Segarra Roca

[31] Intel Core i7 6700HQ. URL: http://ark.intel.com/products/88967/Intel-Core-i7-6700HQ-Processor-6M-Cache-up-to-3.50-GHz (visited on 06/20/2017).
[34] NPM Malloc. URL: https://www.npmjs.com/package/malloc (visited on 06/20/2017).
unwrap = /^function\s*\(/s*/{\s*return\s*\{\s*\[\s*\S*\]};\s*\}/

addLocationData = (first, last) ->
  (obj) ->
    if obj isnt null and typeof obj is "object"
      obj.locations =
        lines:
          first: first.first_line
          last: last.last_line
        columns:
          first: first.first_column
          last: last.last_column
        obj

o = (patternString, action, options) ->
  # Remove extra spaces
  patternString = patternString.replace /\s{2,}/g, ' '  
  return [patternString, '$$ = $1;\', options] unless action
  patternCount = patternString.split(' ').length

  action = if match = unwrap.exec action then match[1] else "(#(action)())"
  
  # All runtime functions we need are defined on "yy"
  action = action.replace /\bnew /g, '"yy."'
  action = action.replace /\b(?:Ast.copyOf)\b/g, 'yy.$&'
  # Also objects
  action = action.replace /\b(?:PRIMITIVE_TYPES)/g, 'yy.$&'
[  
    patternString,  
  
    (  
      if action.indexOf('$$') >= 0  
        action  
      else  
        "$$ = #{addLocationData}(@1, @#{patternCount})(#{action});"  
    ),  
  
  options  
]  

r = (pattern, value) ->  
  [  
    pattern.toString()[1...-1],  
    (if value.match(/\/*.+/)? then value else "return '#{value}'")  
  ]  

lexRules = [  
  r /\/\*/.*/, "/* ignore comment */"  
  r /\/\*(\[\n\n|\r)*?\*\//, "/* ignore multiline comment */"  
  r /\s+/, "/* skip whitespace */"  

  r /\+\+/, '++'  
  r /\-\-/, '--'  
  r /\+=/, '+='  
  r /-=/, '-='  
  r /\*=/, '*='  
  r /\%=/, '%='  
  r /\*/, '*'  
  r /\//, '/'  
  r /-/, '-'  
  r /%/, '%'  
  r /\+/, '+'  
  r /&/, '&'  
  r /\!=/, '!='  
  r /\or\b/, '||'  
  r /\and\b/, '&&'  
  r /not\b/, '!'  
  r /not_eq\b/, '!='  
  r /\<%/, '{'  
  r /%>/, '}'  
  r /%:/, '#'  
]
APPENDIX A. LANGUAGE GRAMMAR

85  \tau /\|\|/\,  \quad '||'
86  \tau /&&/\,  \quad '&&'
87  \tau /!\,  \quad '!'  
88  \tau /<<\,  \quad '<<<'
89  \tau />>\,  \quad '>>'
90  \tau />=\,  \quad '>='  
91  \tau /<=\,  \quad '<='  
92  \tau /\>\,  \quad '>'  
93  \tau /\<\,  \quad '<'
94  \tau /=\,  \quad '=='
95  \tau /;\,  \quad ';'
96  \tau /\{\,  \quad '{'
97  \tau /\}\,  \quad '}'  
98  \tau /\(/\,  \quad '('  
99  \tau /\)/\,  \quad ')'
100  \tau /\\[\,  \quad '['  
101  \tau /\]\,  \quad ']'
102  \tau /,\,  \quad ','  
103  \tau /\#\,  \quad '#'
104  \tau /new\b/\,  \quad 'NEW'
105  \tau /delete\b/\,  \quad 'DELETE'
106  \tau /return\b/\,  \quad 'RETURN'
107  \tau /cin\b/\,  \quad 'CIN'
108  \tau /cout\b/\,  \quad 'COUT'
109  \tau /endl\b/\,  \quad 'ENDL'
110  \tau /int\b/\,  \quad 'INT'
111  \tau /double\b/\,  \quad 'DOUBLE'
112  \tau /char\b/\,  \quad 'CHAR'
113  \tau /bool\b/\,  \quad 'BOOL'
114  \tau /string\b/\,  \quad 'STRING'
115  \tau /void\b/\,  \quad 'VOID'
116  \tau /include\b/\,  \quad 'INCLUDE'
117  \tau /using\b/\,  \quad 'USING'
118  \tau /namespace\b/\,  \quad 'NAMESPACE'
119  \tau /std\b/\,  \quad 'STD'
120  \tau /if\b/\,  \quad 'IF'
121  \tau /else\b/\,  \quad 'ELSE'
122  \tau /while\b/\,  \quad 'WHILE'
123  \tau /for\b/\,  \quad 'FOR'
Extending a browser C++ simulator

Albert Segarra Roca

138  r /const\b/, 'CONST'
139  r /true\b/, 'BOOL_LIT'
140  r /false\b/, 'BOOL_LIT'
141  r /\[0-9\]*\.\[0-9\]+\b/, 'DOUBLE_LIT'
142  r /\([1-9]\[0-9\]*\|0\)/, 'INT_LIT'
143  r /\"[\"\"]\"\b/,'CHAR_LIT'
144  r /\"[^-\"]*\"/,'STRING_LIT'
145  r /nullptr\b/, 'NULLPTR'
146  r /\NULL\b/, 'NULLPTR'
147  r /([a-z]|(\[0-9\]+), 'DOUBLE_LIT'
148  r /$/\b/, 'EOF'
149  r /\./, 'INVALID'
150
151 # Operators at the top of this list have higher precedence than the ones lower
152 operators = [
153   ['right', 'THEN', 'ELSE'],
154   ['left', '!' ],
155   ['nonassoc', '++++', '+-+-']
156   ['right', '!', '++', '+-', 'u+', 'u-', 'deref', 'ref', 'NEW', 'DELETE'],
157   ['left', '*', '/', '%'],
158   ['left', '+', '-'],
159   ['left', '>>', '<<'],
160   ['left', '<', '<=', '>', '>='],
161   ['left', '==', '!='],
162   ['left', '&&'],
163   ['left', '||'],
164   ['right', '+=', '-=', '*=', '/=', '%=', '='],
165   ['left', 'CIN'],
166   ['right', 'type_decl']
167 ]
168
169 bnf = {
170  prog: [
171    o 'top_level_decl_seq EOF', -> new ProgramAst $1
172  ]
173
174  top_level_decl_seq: [
175    o 'top_level_decl_seq top_level_decl', -> $$ addChild $2
176    o '\', -> new List
177  ]
178
179  top_level_decl: [
180    o 'include', -> null
181    o 'function'
182    o 'declaration ;'
183  ]
include: [  
  o '# INCLUDE < id >', ->  
  o 'USING NAMESPACE STD ;', ->  
  ]

function: [  
  o 'type_specifier_seq decl_var_reference ( arg_seq ) { block_instr }',  
  -> new Function new DeclarationGroup($1, [$2]),$4,$7  
  ]

arg_seq: [  
  o 'arg_seq , arg', -> $$.addChild $3  
  o 'arg', -> new List $1  
  o '', -> new List  
  ]

dec: [  
  o 'type_specifier_seq decl_var_reference', -> new FuncArg $1, $2  
  ]

block_instr: [  
  o 'block_instr instruction', -> $$$.addChild $2  
  o 'block_instr { block_instr }', -> $$$.addChild new ScopedList($3)  
  o '', -> new List  
  ]

instruction: [  
  o 'basic_stmt ;'  
  o 'if'  
  o 'while'  
  o 'for'  
  o 'return_stmt ;'  
  o ';', -> null  
  ]

basic_stmt: [  
  o 'block_assign'  
  o 'declaration'  
  o 'cout'  
  o 'expr'  
  ]

return_stmt: [  
  o 'RETURN expr', -> new Return $2  
  o 'RETURN', -> new Return  
  ]

funcall: [  
  o 'id ( param_seq )', -> new Funcall $1,$3  
  o 'id ( VOID )', -> new Funcall $1, new List  
  ]
param_seq: [
  o 'param_seq , param', -> $$$.push $3
  o 'param', -> [$1]
  o ' ', -> []
]

param: [
  o 'expr', -> $1
]

if: [
  o 'IF ( expr ) instruction_body', (-> new IfThen $3, $5), prec: "THEN"
  o 'IF ( expr ) instruction_body else', -> new IfThenElse $3, $5, $6
]

while: [
  o 'WHILE ( expr ) instruction_body', -> new While $3, $5
]

optional_expr: [
  o 'expr'
  o ' ', -> no
]

optional_basic_stmt: [
  o 'basic_stmt'
  o ' ', -> no
]

for: [
  o 'FOR ( optional_basic_stmt ; optional_expr ; optional_expr ) instruction_body',
    -> new For $3, $5, $7, $9
]

else: [
  o 'ELSE instruction_body', -> $2
]

cin: [
  o 'CIN block_cin', -> $2
]

block_cin: [
  o 'block_cin >> expr', -> $$$.addChild $3
  o '>> expr', -> new Cin $2
]

cout: [
  o 'COUT block_cout', -> $2
]

block_cout: [
  o 'block_cout << expr', -> $$$.addChild $3
Extending a browser C++ simulator

Albert Segarra Roca

APPENDIX A. LANGUAGE GRAMMAR
Extending a browser C++ simulator

Albert Segarra Roca

APPENDIX A. LANGUAGE GRAMMAR 108
expr: [
  expr + expr', -> new Add $1, $3
  expr - expr', -> new Sub $1, $3
  expr * expr', -> new Mul $1, $3
  expr / expr', -> new Div $1, $3
  expr % expr', -> new Mod $1, $3
  expr && expr', -> new And $1, $3
  expr || expr', -> new Or $1, $3
  expr < expr', -> new Lt $1, $3
  expr > expr', -> new Gt $1, $3
  expr <= expr', -> new Lte $1, $3
  expr >= expr', -> new Gte $1, $3
  expr == expr', -> new Eq $1, $3
  expr != expr', -> new Neq $1, $3
  expr += expr', -> new AddAssign $1, $3
  expr -= expr', -> new SubAssign $1, $3
  expr *= expr', -> new MulAssign $1, $3
  expr /= expr', -> new DivAssign $1, $3
  expr %= expr', -> new ModAssign $1, $3
  expr = expr', -> new Assign $1, $3
  expr ++', -> new PostInc $1
  expr --', -> new PostDec $1
  - expr', -> new Usub $2, prec: "u-"
  + expr', -> new Uadd $2, prec: "u+
  !! expr', -> new Not $2
  \++ expr', -> new PreInc $2
  \-- expr', -> new PreDec $2
  & expr', -> new AddressOf $2, prec: "ref"
  * expr', -> new Dereference $2, prec: "deref"
  funcall
  id
  expr accessor', -> new ArraySubscript($1, $2)
  ( expr ), -> $2
  literal
  cin
  new_expr
  delete_expr
]

id: [
  ID', -> new Id $1
]

start = "prog"

bnf[start][0][1] = "return #{bnf[start][0][1]};"
APPENDIX

B

PERFORMANCE EVALUATION PROGRAMS AND SCRIPT

B.1. Main script

```coffee
#!/usr/bin/env coffee

fs = require 'fs'
path = require 'path'
spawnSync = require 'child_process'
cmm1 = require './cmm1'
cmm2 = require './cmm2'
memory = new cmm2.Memory

OUTPUT_DIR = './out'
MS_PER_SEC = 1e3
NANOSECONDS_PER_MS = 1e6
REPETITIONS = 3

fs.mkdirSync(OUTPUT_DIR) unless fs.existsSync(OUTPUT_DIR)

range = (b, e, i) -> (x for x in [b..e] by i)

TESTS =
  primality: range(1, 10, 1)
  fibonacci: range(25, 50, 5)
  collatz: range(10000, 50000, 5000)
  matrixmul: range(50, 350, 50)

useTimer = (file, value) ->
```

110
# See https://github.com/albertsgrc/c--benchmark/blob/master/timer/timer.cc
# for the implementation of timer

{ stderr: output } = spawnSync "timer", ['-ni', file, value], encoding: 'utf-8'

{ elapsed } = JSON.parse output

elapsed/1000 # Elapsed is in microseconds, result in milliseconds

cmmMeasure = (file, value, compileFn, compileTransform, runFn) ->
  compilation = compileTransform compileFn(fs.readFileSync(file, 'utf-8'))
  start = process.hrtime()
  runFn(compilation, value.toString())
  diff = process.hrtime(start)
  [ seconds, nanoseconds ] = diff
  seconds*MS_PER_SEC + nanoseconds/NANOSECONDS_PER_MS

measureTime =
  cmm2: (file, value) ->
    cmmMeasure(file, value,
      cmm2.compile, ((x)-> x.program.attachMemory(memory); x.program), cmm2.runSync
    )
  cmm1: (file, value) -> cmmMeasure(file, value, cmm1.compile, ((x)->x), cmm1.execute)
  cc: (file, value) -> useTimer file[...-3], value # Remove .cc extension
  js: useTimer
  py: useTimer

extNames = Object.keys(measureTime)

results = {}

shouldPerformTest = (language, test, value) ->
  if test is 'fibonacci'
    if ((language.indexOf('cmm') >= 0 and value > 35) or (language is 'py' and value > 40))
      return no
  if test is 'matrixmul'
    if language in ['cmm11']
      return no
  return yes

for test, values of TESTS
  languages = {}
  for file in fs.readdirSync("./#{test}") when (extension = path.extname(file)[1..]).length > 0
    languages[extension] = path.join "./#{test}", file
  languages.cmm1 = languages.cmm11 = languages.cmm2 = languages.cmm
B.2. Primality test

B.2.1. C++11

```cpp
#include <cstdlib>

int isPrime(int n) {
    if (n < 2) return false;
    if (n == 2) return true;
    if (n%2 == 0) return false;

    for(int i = 3; (i*i) <= n; i += 2) {
        if (n%i == 0) return false;
    }

    return true;
}

int main(int argc, char* argv[]) {
    int N = atoi(argv[1]);
    for (int i = 1; i <= N*100000; ++i) {
        isPrime(i);
    }
}
```
B.2.2. C–

```cpp
#include <iostream>
using namespace std;

int isPrime(int n) {
    if (n < 2) return false;
    if (n == 2) return true;
    if (n%2 == 0) return false;
    for(int i = 3; (i*i) <= n; i += 2) {
        if (n%i == 0) return false;
    }
    return true;
}

int main() {
    int N; cin >> N;
    for (int i = 1; i <= N*100000; ++i) {
        isPrime(i);
    }
}
```

B.2.3. NodeJS

```javascript
#!/usr/bin/env node

function isPrime(n) {
    if (n < 2) return false;
    if (n == 2) return true;
    if (n%2 == 0) return false;
    for (var i = 3; (i*i) <= n; i += 2) {
        if (n%i == 0) return false;
    }
    return true;
}

var N = parseInt(process.argv[2]);
for (var i = 1; i < N*100000; ++i) {
    isPrime(i);
}
```
B.2.4. Python3

```python
#!/usr/bin/env python3

import sys

def isPrime(n):
    if n < 2: return False;
    if n == 2: return True;
    if n%2 == 0: return False;
    i = 3
    while (i*i) <= n:
        if n%i == 0: return False
        i +=2
    return True

N = int(sys.argv[1])

for i in range(1, N*100000 + 1): isPrime(i)
```

B.3. Fibonacci

B.3.1. C++11

```cpp
#include <cstdlib>

int fib(int n) {
    if (n < 2) return n;
    else return fib(n - 2) + fib(n - 1);
}

int main(int argc, char* argv[]) {
    int N = atoi(argv[1]);
    fib(N);
}
```

B.3.2. C–

```c
#include <iostream>

using namespace std;

int fib(int n) {
    if (n < 2) return n;
    else return fib(n - 2) + fib(n - 1);
}
```
Extending a browser C++ simulator

Albert Segarra Roca

B.3.3. NodeJS

```javascript
#!/usr/bin/env node

function fib(n) {
    if (n < 2) return n;
    else return fib(n - 2) + fib(n - 1);
}

var N = parseInt(process.argv[2]);
fib(N);
```

B.3.4. Python3

```python
#!/usr/bin/env python3

import sys

def fib(n):
    if n < 2: return n
    else: return fib(n - 2) + fib(n - 1)

N = int(sys.argv[1])
fib(N)
```

B.4. Collatz sequence

B.4.1. C++11

```cpp
#include <cstdlib>

int collatz(int n) {
    if (n%2 == 0) return n/2;
    return 3*n+1;
}
```
Extending a browser C++ simulator

void sequence(int n) {
    while (n != 1) {
        n = collatz(n);
    }
}

void sequences_collatz(int n) {
    for (int i = 1; i <= n; ++i) {
        sequence(i);
    }
}

int main(int argc, char* argv[]) {
    int N = atoi(argv[1]);
    sequences_collatz(N);
}

B.4.2. C—

#include <iostream>
using namespace std;

int collatz(int n) {
    if (n%2 == 0) return n/2;
    return 3*n+1;
}

void sequence(int n) {
    while (n != 1) {
        n = collatz(n);
    }
}

void sequences_collatz(int n) {
    for (int i = 1; i <= n; ++i) {
        sequence(i);
    }
}

int main() {
    int N; cin >> N;
    sequences_collatz(N);
}

B.4.3. NodeJS

#!/usr/bin/env node

#APPENDIX B. PERFORMANCE EVALUATION PROGRAMS AND SCRIPT 116
Extending a browser C++ simulator

Albert Segarra Roca

function collatz(n) {
  if (n%2 == 0) return n/2;
  return 3*n+1;
}

function sequence(n) {
  while (n !== 1) {
    n = collatz(n);
  }
}

function sequences_collatz(n) {
  for (let i = 1; i <= n; ++i) {
    sequence(i);
  }
}

var N = parseInt(process.argv[2]);

sequences_collatz(N);

---

B.4.4. Python3

#!/usr/bin/env python3

import sys

def collatz(n):
  if (n%2 == 0): return n/2
  return 3*n+1

def sequence(n):
  while n != 1: n = collatz(n)

def sequences_collatz(n):
  for i in range(1, n+1): sequence(i)

N = int(sys.argv[1])

sequences_collatz(N)

---

B.5. Square matrix multiplication

B.5.1. C++11

#include <cstdlib>

---
void matrix_mul(int N) {
    double **A = new double*[N], **B = new double*[N], **R = new double*[N];
    for (int i = 0; i < N; ++i) {
    }
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
                R[i][j] = A[i][k]*B[k][j];
}

int main(int argc, char* argv[]) {
    int N = atoi(argv[1]);
    matrix_mul(N);
}

B.5.2. C–

#include <iostream>
using namespace std;

void matrix_mul(int N) {
    double **A = new double*[N], **B = new double*[N], **R = new double*[N];
    for (int i = 0; i < N; ++i) {
    }
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
                R[i][j] = A[i][k]*B[k][j];
}

int main() {
    int N; cin >> N;
    matrix_mul(N);
}

B.5.3. NodeJS

#!/usr/bin/env node

function matrixMul(N) {
    var A = new Array(N), B = new Array(N), R = new Array(N);
    for (var i = 0; i < N; ++i) {

A[i] = new Array(N); B[i] = new Array(N); R[i] = new Array(N);
}

for (var i = 0; i < N; ++i)
    for (var j = 0; j < N; ++j)
        for (var k = 0; k < N; ++k)
            R[i][j] = A[i][k]*B[k][j];

var N = parseInt(process.argv[2]);

matrixMul(N);

B.5.4. Python3

#!/usr/bin/env python3

import sys, numpy

def matrixMul(N):
    A = numpy.zeros((N, N))
    B = numpy.zeros((N, N))
    R = numpy.zeros((N, N))

    for i in range(0, N):
        for j in range(0, N):
            for k in range(0, N):
                R[i][j] = A[i][k]*B[k][j]

    N = int(sys.argv[1])

    matrixMul(N)