C++ simulation in the browser

Ricard Gascons

Supervisor: Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

This dissertation is submitted for the degree of
Informatics Engineering

January 2016

Abstract

The aim of this thesis is to design, develop and evaluate an open source tool to edit, run and debug C++ code inside a webpage without requiring a remote server to compile and execute the programs. C++ is widely used in education and industry, but to date no tools that run C++ code natively in the browser have been found. For this reason, it is been decided to develop an interpreter for a small subset of C++ that runs in a simple web interface. We call this tool C−−. This thesis parts from an older project able to interpret a small subset of C++ in the browser, and then designs and implements several features such as a debugger, non-blocking input/output operations, and a public API. In a second stage a web interface is built to allow users running and debugging code in a simple manner. After the first implementation of the web interface, feedback from testers is received and several adaptations for the improvement of the user experience are undertaken. Moreover, several efficiency tests have been performed to compare C−− execution times to other existing tools, such as GCC, JavaScript or Python. Finally, the tool is deployed to *cmm.jutge.org* where anyone can use it.

Resum

L'objectiu d'aquest projecte final de carrera és dissenyar, desenvolupar i evaluar una eina de codi obert capaç d'editar, executar i debugar codi C++ dins una pàgina web sense que es necessiti un servidor remot per compilar i executar els programes. C++ és molt popular en el mon educatiu i a la indústria, però a data d'avuí no s'ha trobat una eina que sigui capaç d'executar codi en C++ de forma nativa dins el navegador. Per aquesta raó s'ha decidit desenvolupar un intèrpret per un petit subconjunt de C++ que s'executi dins una interfície web. Aquesta eina l'hem anomenat C−−. Aquest projecte de final de carrera prové d'un projecte anterior capaç d'interpretar un petit subconjunt de C++ en el navegador, que posteriorment s'ha ampliat per donar suport a noves carecterístiques com un debugger, operacions d'entrada/sortida no bloquejants i una API pública. En la segona fase del projecte s'ha desenvolupat una interfície web per a què els usuaris puguin executar i debugar codi d'una forma molt senzilla. Després de la primera implementació de la interfície web, s'han realitzat proves amb usuaris on s'ha recollit el seu feedback i s'han fet les modicacions pertinents per tal de millorar l'experiència d'usuari. A més, s'han realitzat diversos testos d'eficiència on es mesura el temps d'execució de programes en C−− i es comparen els resultats amb altres eines existents, com GCC, JavaScript o Python. Finalment, l'eina és desplegada a *cmm.jutge.org* on qualsevol persona hi pot accedir.

Table of contents

Chapter 1

Introduction

1.1 Overview

This project consists in designing, building and evaluating an open source interpreter able of running a subset of C++ natively in a browser client, without requiring a remote server that compiles and runs the program. The result can be seen at *https://cmm.jutge.org/*.

The project derives from a previous Compilers' class project proposed by Jordi Petit, the current advisor of the project. The goal is to allow running a small subset of the C++ language, which we call C−−, with expectations of expanding it in the future.

This tool will allow users to directly edit, run and debug inside a browser client. In this way, the user is able to program without worrying about configuring a proper environment that is needed for compiling and running C++ code. On top of that, the fact that the compilation and execution would be done directly in the browser frees the server from doing any work.

The system tool will also include a debugger, able of pausing, resuming and stopping the execution of the program, to watch and edit the variables and its values in the program memory.

We expect this project to be a success among young students and educational institutions who do not have the resources needed to prepare proper programming environments.

1.2 Problem description

As stated in the overview, this project is the result of trying to solve a number of issues:

Setting-up a C++ environment is not easy. Preparing an environment for students, schools and universities that is capable of compiling and executing C++ programs is a resource-consuming task that leads to an expense of monetary and human resources that not everybody can afford. On top of that, there's a continuous maintenance labor that requires someone with experience in the IT field. Again, not everyone can afford it.

Strong dependency with the server. Online compilers and Judges are dependent on a server that does the compilation and evaluation of the program. This is a slow process and is often limited by the online tools themselves.

Lack of professional solutions. Although similar initiatives have been found, such as Emscripten [12], Rushton [26] or Cherp [7], none of them are capable of running C++ natively in a browser client. A more detailed analysis is done in Chapter 2.1.

1.3 Solution overview

Given the previous issues, a number of solutions have been studied. Below lie their descriptions:

An open source C++ simulator within the browser. A compiler that simulates the behavior of a typical C++ program will be built in the browser client and without the necessity of having it connected to a server. Thanks to this, the user can run a program in a sandbox environment, repeatedly, and without worrying about server-side penalties.

A user-friendly debugger. To provide easy and reliable debugging tools, a debugger will be built in the simulator itself, giving the control of the program to the user.
The debugger will support pausing, stopping and resuming the program as well as putting breakpoints at any line, plus watching and editing the variables of the program.

1.4 Motivation

Personal motivation

I love programming languages. I still remember the days I was figuring out how to write an Android application, learning to think like a computer, looking at pieces of Java code, and tinkering with databases without even knowing what they really were. Ever since I was a teenager, I've been amazed by programming languages and how they can manage to do all sorts of complex tasks with a relatively easy syntax.

Helping others to become better Computer Scientists. I want to provide a better environment for the Computer Scientists of the future. I still remember installing my first Linux Distribution just to be able to compile a *Hello World* program. While I recommend installing a Linux Distribution as a means of personal growth and self-accomplishment, I also think we can do better in order to help the youngest ones.

Become a greater Computer Scientist and Engineer. Writing a compiler is a difficult task and involves applying CS concepts from the entire grade. From tree traversal to managing the memory of the program, this is a golden opportunity for me to grow as a Computer Scientist and further develop my Software Engineering skills.

A tool for teachers, kids and teenagers

Schools and high schools often lack of a proper IT department, and are not capable of setting up complicated and often unstable programming environments for the students to use. This ends up with the student not receiving a basic Computer Science education.

Trial and error without a high cost

During the first evaluation of the project it was considered building or using an already working online Judge, although in the end the idea was scrapped. The reason was that, although Online Judges are very useful when taking a programming course, they often set limits at how many programs one can upload to the server. The existing tools are not made for young students, who often learn better in a trial and error basis.

Building an extensible solution

Another important objective of this project is to build a software package taking future growth into consideration. Because of the open source nature of the project, components of the software will be developed by unrelated parties independently. Adaptable software components are necessary since components from external contributors are more unlikely to fit in the current system.

1.5 Summary

This report consists of four different parts. Each one of them describes a phase in the development of the project:

Planning. Details the time plan and the budget to develop the project, as well as identifying and analyzing the problem to solve following a proposed methodology.

Design. Details the design of the system, component by component.

Implementation. Describes the development of the different components that compose the solution.

Evaluation. Describes and executes the testing and evaluation methodology, and discusses usability principles through a number of usage tests.

Conclusion. The Conclusion closes the report with a summary of the accomplished objectives, followed by an overview of what the future expansions of the project would be, and ending with a personal evaluation of the thesis.

Part I

Planning

Chapter 2

Project plan

2.1 State of the art

Most of the compilers written today, such as GCC or CLANG, are based on the Dragon book [2] and thus many of them share the same design principles. We can find these similarities in the syntax across languages, or how their type system behaves. We call such languages "C-like languages", because most of them inherit features from the C language. One can find out these features in the C Programming language book [22]. C++, as a superset of C, is no exception to this rule.

This family of languages was not designed with the web and browsers in mind, and there's a lack of tools that offer the possibility to execute native code from a website.

There are many projects that attempt to provide support for desktop-like languages in the browser: Emscripten, Rusthon and Cheerp.

Rusthon. Rusthon [26] is an integrated Python transpiler that targets multiple backend languages, like JavaScript and C++. The JavaScript backend implements most of the dynamic and some builtin functions of Python.

Cheerp. Cherp [7] is a C++ compiler that generates JavaScript code that can run on any browser.

Compile LLVM bytecode to asm.js through Emscripten. Emscripten [12] is an LLVM based project that compiles C and C++ into highly performant JavaScript in the asm.js format. This process could be used to compile GCC or CLANG to JavaScript but due to the big size of these projects, browsers are not able to execute them smoothly. Among all these options, this is the one that gets closer to what we want to accomplish in this project. Emscripten could be used to compile an already existing C++ compiler like GCC [15] or Clang [8] and run it on the browser. One could argue this approach *could* work, but the loading times would skyrocket and the compiler alone would weight too much, making it impractical to use it.

Given that there are no pre-made solutions for our project, we have to build our own solution to the problem.

2.2 Specification

This chapter specifies the different stakeholders of the project and the main and secondary objectives of the thesis' project.

2.2.1 Stakeholders

A stakeholder is a group that has an interest in a company, project or institution, and can either affect or be affected by the business. The primary stakeholders in a typical project are its investors, employees and customers.

Freshman college undergraduates. They will learn algorithms and data structures by writing programs of all difficulties.

High school students. They want to start learning the basic principles of Computer Science, while also being introduced to a wide range of algorithms and techniques.

Primary school students. They will develop logic and abstract thinking by developing programs that accomplish simple tasks.

Schools and High Schools. They will use the software to teach programming to their students.

2.2.2 Main objectives

The existing compiler, initially built in the Compilers' class, will be extended to offer support to new features. Moreover, the website will be built to accommodate the extensions of the compiler.

Develop a debugger. Developing a debugger so that the stakeholders described in section 2.2.1 are able to debug their code easily.

Keep the code base maintainable and scalable. The code base must be testable and reliable; the existing code base must not break when new features are implemented.

Design a new UI. The original website of the project was not meant for scaling. A new website will be be made to match the current status of the project.

Create test programs and documentation. The project will be expanding in the feature. Documentation and proper testing are key for the success of the project.

2.2.3 Secondary objectives

Additionally, extra features are built.

Asynchronous I/O operations. The first version of the compiler did not support I/O operations properly: each time a I/O was fired, a buffer was filled and only in the end of the execution of the program was emptied. Thanks to the nature of NodeJS, now it's possible to perform I/O instructions asynchronously.

IDE-like features for the website. A proper debugger demands a proper editor. The website will be reworked to support features of modern editors and IDEs.

2.3 Methodology

This section specifies the software development methodology and techniques used in this project in order to ensure the best product quality.

2.3.1 Scrum

The project development of this thesis follows Scrum [27] principles. As stated by the Scrum Alliance Organisation, Scrum is an agile methodology for completing complex projects. Scrum is a simple and powerful set of principles and practices that help teams deliver products in short cycles, enabling fast feedback, continual improvement, and rapid adaptation to change. Its main features are:

Approach. A project managed under the Scrum methodology starts with objectives that have been previously outlined by the client and the company. The first task is to split it into partial deliveries, so that the client can rethink aspects to which he did not give enough importance or simply did not know.

Task lists. The working team creates a task list of things to do for the next sprint. It is essential to do a reasonable estimation of the required effort to perform these tasks.

Meetings. it is ideal to perform daily meetings to keep track of the state of each task. Note: due to time constraints, we communicated via email.

Demo. Once the tasks of each phase have been executed, the resulting product is shown to the client. The client then decides if the results are sufficient or if a rethinking of the task is needed.

2.3.2 Development cycle

According to the Scrum methodology, weekly meetings between the thesis advisor and the student will be established to keep track of what's done and to set new milestones to be completed within the next 7 days.

2.3.3 Validation methods

Different tools for code validation will be used:

Unitary tests. Unit testing involves breaking the program into different parts, and subjecting each part to a series of tests. Mocha framework [24] will be used for unitary testing.

Test programs. Programs that test a set of features of the compiler, or evaluates its efficiency.

Continuous Integration. Continuous Integration (CI) is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. [1] The webpage Travis CI [29] will be used for this task.

2.4 Time plan

This section aims to describe the tasks that are going to be executed in order to do the project, along with a time table and a timeline describing the dependencies of each task.
However, we have to take into account that the planning described here is subject to modifications depending on the development of the project.

2.4.1 Project duration

The estimated project duration is approximately 6 moths. The project starts on September 1st, 2016 and the deadline is on January 6th, 2017.

2.4.2 Project planning

This section describes the tasks that planned to do in order to build the project.

Project management course

The goal of the project management course is to help lead the project in the right direction. In this course, different parts of the project are defined:

- Context and Scope

[1]Source: https://martinfowler.com/articles/continuousIntegration.html

- State of the art
- Planning
- Economic viability

During this period I will use LaTeX related tools such as the Overleaf [25] online editor and the Mendeley [23] reference feed.

Software design

In this task, an analysis of the requirements is done and a solution is designed to satisfy them.

Plain whiteboard will be used and meetings with the advisor will be done to ensure the best design will be developed.

It is expected to have a solid design by the end of this task. This is why there is no alternative plan if the design problems are not solved.

Main development

This is the main task of the project. It covers all tasks that are related to the implementation and testing of the virtual machine.

Backend for the debugger. A refactor from the first version of the project will be needed in order to implement a step-by-step debugger.

It is probable that additional features for the language will be cut out of the project in case the debugger takes longer time than expected.

Asynchronous Input and Output operations. This task consists on implementing Input and Output operations that work in the middle of the execution of a program.

It is probable that additional features for the language will be cut out of the project in case the debugger takes longer time than expected.

Implement IDE-like features to the client. A client that handles the endpoints of the compiler API, and provides an interface to the user that helps the coding, execution and debugging of a program.

It is probable that additional features for the language will be cut out of the project in case the debugger takes longer time than expected.

Implement C++ language features. Features that are missing from the first versions of the compiler will be implemented.

Write programs showcasing the new features. Sample programs will be developed to test that the underlying compiler and virtual machine work as intended.

Testing and polishing. In this task, the system will be tested with complex and simple test cases to ensure its stability.

The tests will be written and executed with the help of the Mocha[24] testing suite.

Project report

During this task, a document explaining how the project has been developed will be written. Once again, I will use make extensive use of Overleaf and Mendeley to write the memory.

Oral presentation

Once the memory of the project is finished, the student will prepare for the final oral presentation.

The presentation will be build using the Office PowerPoint software or Google Slides suite.

2.4.3 Human resources

For all the tasks listed above a product manager, a developer, a designer and a tester will be needed. All this roles will be assumed by the student.

2.4.4 Time table

Figure 2.1 summarizes the duration of every task described in the previous section. The total duration of the project is expected to be **440 hours**. The student will need to work 440/16 $\approx$ 28 hour/week to finish the project before the deadline.

Task	**Expected duration (h)**
Project management course	75
Software design	15
Debugger backend	**70**
Analysis	15
Implementation	40
Testing	10
Integration	5
Async I/O	**40**
Analysis	5
Implementation	20
Testing	10
Integration	5
Client	**50**
Analysis	5
Implementation	30
Testing	15
Other language features	**65**
Analysis	10
Implementation	40
Testing	15
Sample programs	**35**
Analysis	5
Implementation	15
Testing	15
Testing and polishing	40
Project memory	40
Oral presentation	10
Total	**440**

Fig. 2.1 Planning time table

2.4.5 Project timeline

Figure 2.2 shows the expected timeline of the project, taking into consideration the dependencies between tasks.

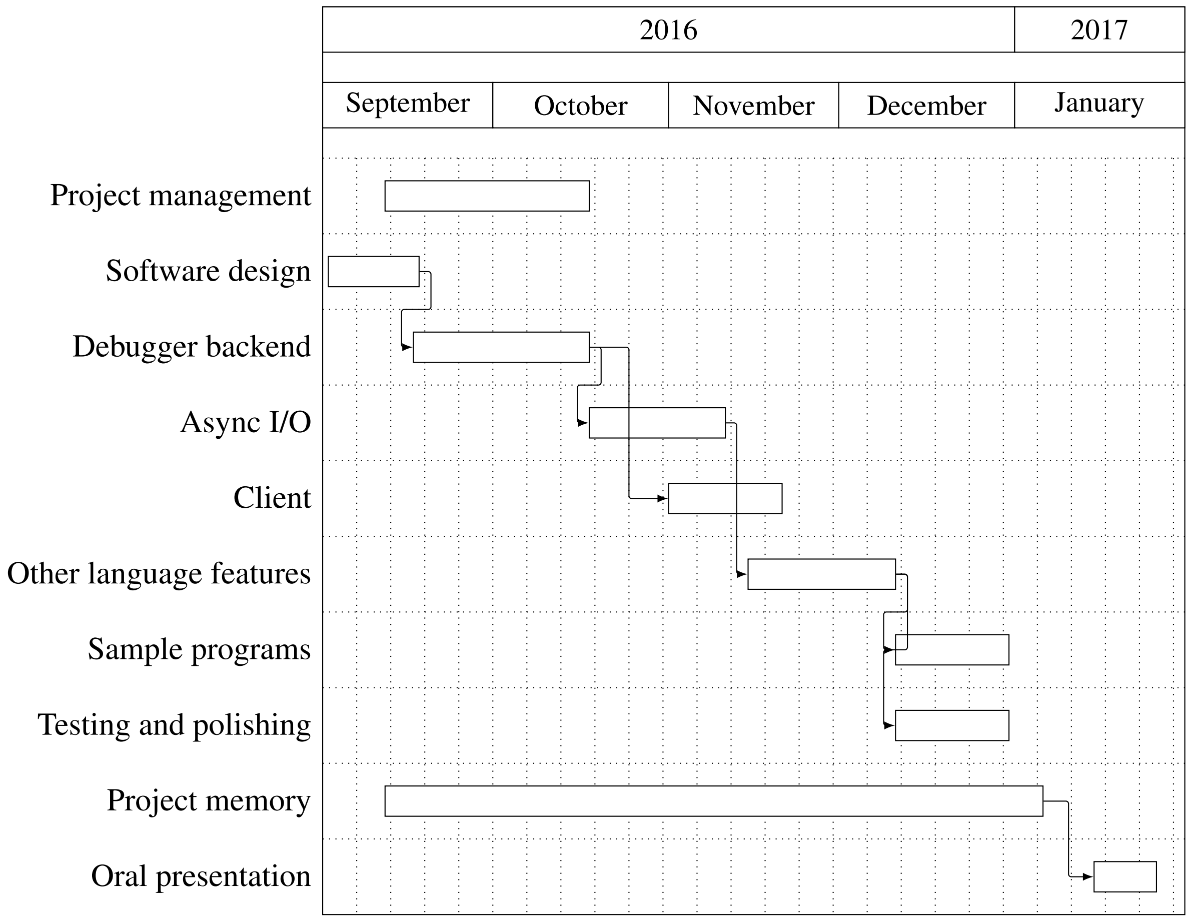


Fig. 2.2 Planning timeline

2.5 Economic plan

2.5.1 Hardware resources

The project will be developed using a laptop. A mouse is also used with the laptop computer. There is no other hardware needed.

Hardware	Cost (€)	Useful life (years)	Amortized cost (€)
Laptop	1000.00	4	13.13
Wireless Optical Mouse	60.00	4	2.23
Total	1060.00		15.36

Table 2.1 Hardware budget

2.5.2 Software resources

The software used to develop the project is: Ubuntu, Atom, a browser, LaTeX, gulp, git, nodejs, npm, Javascript and Coffeescript.

All the software listed above can be used for free. Table 2.2 shows the licenses of most of the software needed.

Software	License
Ubuntu	http://www.ubuntu.com/about/about-ubuntu/our-philosophy
Atom	https://github.com/atom/atom/blob/master/LICENSE.md
LaTeX	https://latex-project.org/lppl/
gulp	https://github.com/gulpjs/gulp/blob/master/LICENSE
git	https://git-scm.com/about/free-and-open-source
nodejs	https://github.com/nodejs/node/blob/master/LICENSE
npm	https://github.com/npm/npm/blob/latest/LICENSE
Coffeescript	https://github.com/jashkenas/coffeescript/blob/master/LICENSE
Mozilla Firefox	https://www.mozilla.org/en-US/foundation/licensing/

Table 2.2 Software licenses

2.5.3 Human resources

Table 2.3 shows the expected salary per project role and table 2.4 shows the human resources budget allocated to the project.

Role	Payment (€/h)
Project manager	40.00
Software engineer	45.00
Software developer	35.00

Table 2.3 Salary per role

Role	Task	Time (h)	Cost (€)
Project manager	Project management course	75	3000.00
	Project memory	40	1600.00
	Oral presentation	10	400.00
Software engineer	Analysis and design	15	675.00
Software developer	Debugger backend	70	2450.00
	Async I/O	40	1400.00
	Client	50	1750.00
	Other language features	65	2275.00
	Sample programs	35	1225.00
	Testing and polishing	40	1400.00
Total	-	-	16175.00

Table 2.4 Human resources budget

2.5.4 Other resources

Electricity

Electricity is needed to power the hardware. Table 2.5 shows the power consumption, the estimated time of usage, and the cost of every hardware that needs to be powered.

Hardware	Consumption (Wh)	Time of usage (h)	Cost (€)
Laptop	400.00	400.00	23.58
Total			23.58

Table 2.5 Electricity costs

Internet connection

Internet connection is necessary to perform all the tasks. The student uses the university connection most of the time.

2.5.5 Total

Table 2.6 shows the total budget needed to develop the whole project. A 10% of the total cost is added to face any possible contingencies.

Resource	**Total cost (€)**
Hardware	15.36
Software	0.00
Human resources	16175.00
Electricity	23.58
Internet	0.00
Subtotal	16213.94
Contingency (10%)	1621.40
Total	17835.34

Table 2.6 Total budget

2.6 Sustainability report

This section contains a basic sustainability report of the project based on the method described by Christian Felber in ’The economy of the common good’ [13]. This book is used in other sustainability reports with great success, as it provides a realistic analysis of an economic scenario.

2.6.1 Economic analysis

The economical analysis will be completed and all costs are assessed and reasonable. Resources will be used efficiently, and the greater part of the time will be spent in the most important tasks. A contingency budget was added to take care of any unexpected problems.

However, the project is not aimed to be profitable, so there will be no direct economic benefit from it.

2.6.2 Social impact analysis

The project will make the learning process more fun, it will also reduce the amount of work for teachers, and it will make freshman programmers able run programs easily.

However, the project will not affect the mainstream consumer directly. It will only be relevant inside a specific community in computer science.

2.6.3 Environmental impact analysis

The project uses 124kWh of electricity, which is approximately 51 kg of CO_2 [6]. This is the amount of CO_2 that a 13W lightbulb during 163 days. On top of that, most of the information to write this project will be accessed through the Internet, with a few exceptions.[2] Refer to table 2.7 for details.

Moreover, most of the hardware used in this project will eventually have to be retired and decomposed into different contaminants.

Task	**Consumption (kWh)**
Charge computer	74
Print thesis copies	50
Total	124

Table 2.7 Costs detail

[2]See References section

Chapter 3

Starting point

Before working on this project, my classmates P. Oliver, A. Segarra and me built a little prototype of the project in the Compilers' class (called C−− 1.0 from now on).

This 1.0 implementation supported features we considered to be basic enough in order to write small C++ programs. Therefore, it was decided that the first prototype should implement the first working versions of the following features:

1. A **compiler** implementing basic language features, such as a bare bones type system, functions, recursion and a simplified stack.
2. A **web client** implementing a code editor and simple Compile and Run actions.

It is important to note that during the 1.0 implementation, the compiler and the website were tied together under the same package. This monolithic architecture, while simple to build, can be the source of a lot of complexity in the architecture of the program. So for the next release (namely 1.1) both backend and frontend should act independently.

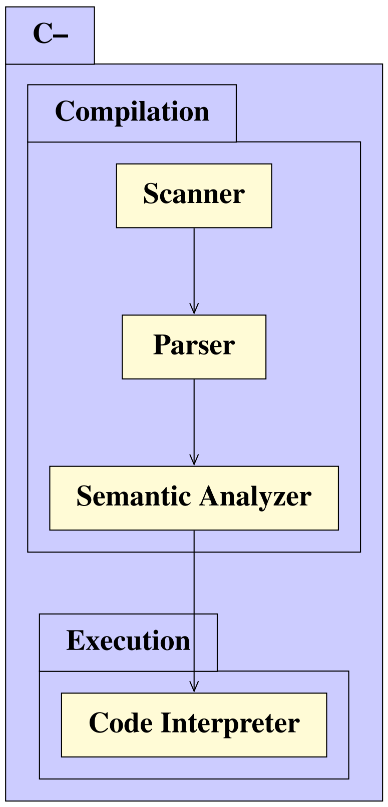


Fig. 3.1 C– 1.0 compiler phases

3.1 The first compiler

3.1.1 Architecture of the system

The Virtual Machine is composed of 3 distinct components: the parser, the semantics module and the interpreter. Figure 3.1 pictures the compiler phases of C−−.

The parser: the parsing of the source-code is done with the help of an external module called Jison [18]. Jison is very similar to YACC and Bison, and integrates perfectly in a JavaScript environment. This module provides an API very similar to the Yacc [31] or Bison [3] parser generators. Its ease of use and simplicity helped developing the project faster.

The semantics module: the semantics module runs just before the interpreter. Typical compile-type checks and AST pre-processing tasks are done in this module such as type checks and instruction coherence.

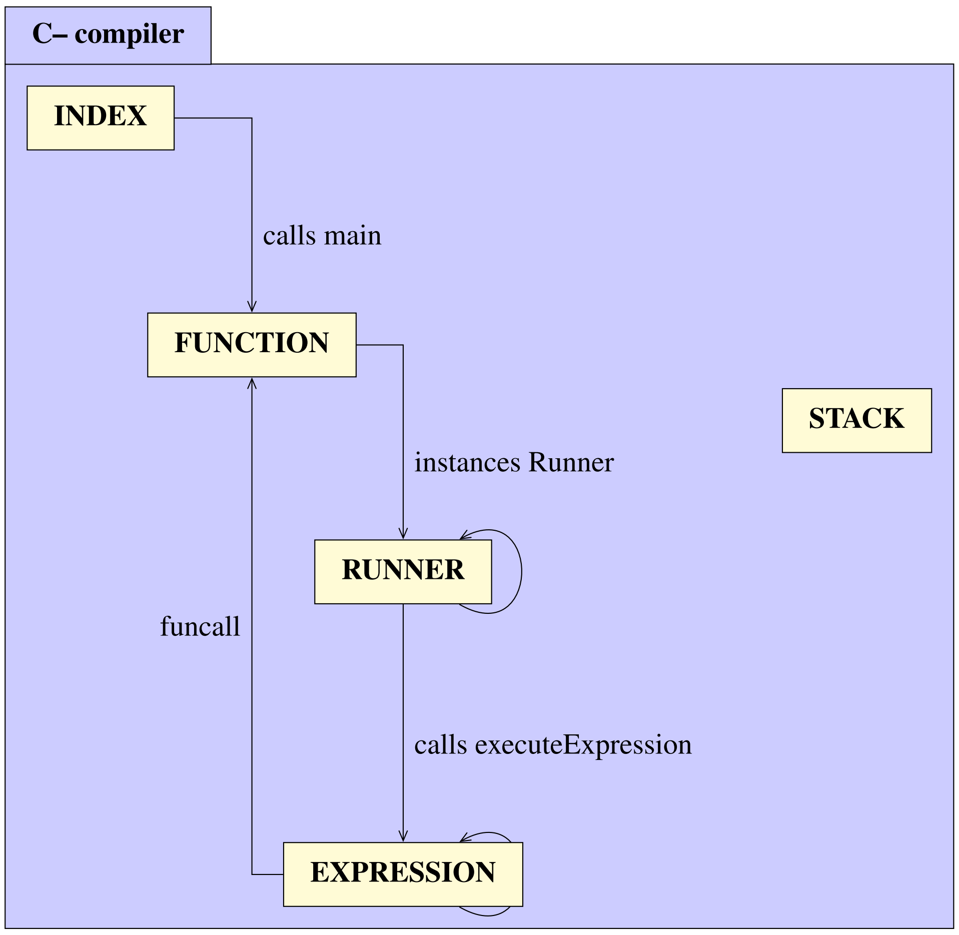


Fig. 3.2 C−− 1.0's architecture

The interpreter: the interpreter is in charge of interpreting and executing the code under the set of rules of the C++ programming language.

Figure 3.2 shows the compiler's 1.0 architecture of the system, with the most important calls.

We can clearly see multiple levels of recursion, between the components themselves and between multiple components. The calls to the Stack component are not included for visualization purposes, as all modules call Stack at some point to access or save variables.

3.1.2 Programming language

Due to the nature of this project, the only suitable language for simulating C++ programs in the browser is Javascript, because is the only language understood by all modern browsers. However, there are also some languages that are capable of transpiling themselves to JS, such as Coffeescript [9], Typescript [30], Dart [11]... For this project the Coffeescript programming language has been chosen, for its ease of use, flexibility, and good tooling support.

3.1.3 Type system

We emulated the basic C++ type system in this first prototype. Users may expect the usual behavior and interactions between the **int**, **char** and **bool** types.

Strings are introduced as types as well. We are aware that **string** typing is a class in C++, but for design purposes we kept it as a C-style type.

Arrays, pointers, references and structs were not supported in the 1.0 version.

3.1.4 Instructions and expressions

All the common statements were supported in the language. Such statements include the **if**, **while** and **for** conditional blocks.

The standard **arithmetic** and **boolean operators** are included, respecting the C/C++ operator precedence. **Compound assignments** and **pre/post increments** are supported as well.

3.1.5 Functions and recursion

Functions and function calls are supported among all types as well as the special **void** type. Recursive calls are supported as well.

Functions are distinguished purely by identifier, so function overloading will result in a compilation error.

3.1.6 Stack

The memory stack implemented in this prototype follows a simple set of rules:

- Before a function call, a new activation record is pushed and initialized with the parameters from the call. The **activation record** is popped after the function returns.
- Before entering in a conditional block (see 3.1.4), a new **scope** is created. The scope is closed as soon as the conditional block has ended. Variable scoping follows the set of rules of the C language, as expected.
- All variables are validated at compile time. A program will not compile if at least one variable is not properly referenced in the stack or the value assigned to the variable does not comply with the type of the variable.
- Global variables are not supported.

3.2 The first client

3.2.1 Programming language

The web client is written in **Coffeescript**, as well as HTML [17] and CSS [10] for the UI. The jQuery [20] and Bootstrap [4] libraries are used as well. We chose this stack because it is very easy to use and provided enough flexibility.

3.2.2 Concurrency and Web Workers

The compiler might perform heavy calculations so a form of concurrency must be achieved to prevent blocking the UI.

The only option that JavaScript offers are **Workers**, which provide a simple messaging API between the main worker and its siblings. Please note it is not possible to achieve true parallelism under Javascript, Workers function thanks to the event loop that all JS programs have.

3.2.3 Summary

In this chapter, we have described how the first iteration of the project was built. C−− 1.0 implemented the most basic features of every component:

- The **Virtual Machine**, that emulated the execution of a C++ program.
- The **client**, that let the users interact with the compiler by running C++ programs.

The next chapter will talk about the system design of the new version of the compiler.

Part II

Design

Chapter 4

System design

This chapter will cover the changes for C−− 1.1, the most recent version of the compiler.

The system is compounded of two distinct parts: the frontend and the backend. The **frontend** is a web interface that acts as a bridge between the user and the backend of the system. The **backend** is the compiler, which runs within the browser. Figure 4.1 shows these components.

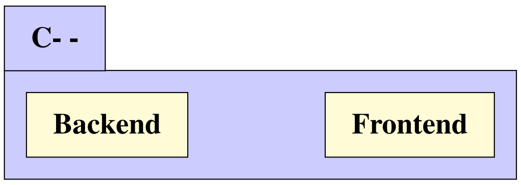


Fig. 4.1 System overview

4.1 Backend architecture

This section considers the backend of the C−− compiler version 1.1. As shown in Figure 4.2, the backend is composed by four distinct parts: the parser, the semantics, interpreter and debugger packages. The order of the packages reflects the sequence of its execution during the compilation of a program.

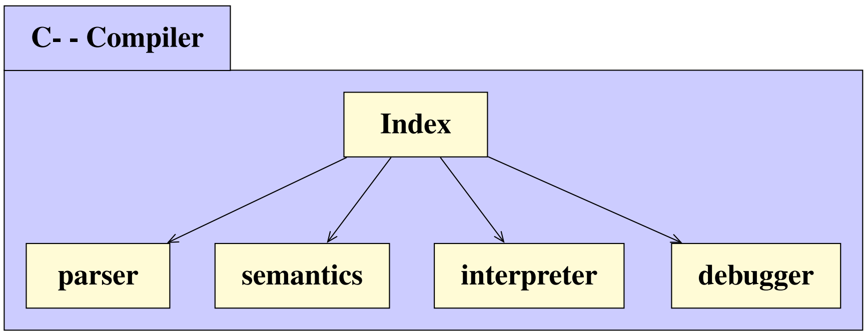


Fig. 4.2 Architecture of the C−− compiler

The following subsections will explain in more detail the role of each of the components.

4.1.1 Parser package

The parser package converts a C−− source code into a set of tokens, organized under an Abstract Syntax Tree (AST). Figure 4.3 shows the compiler's architecture.

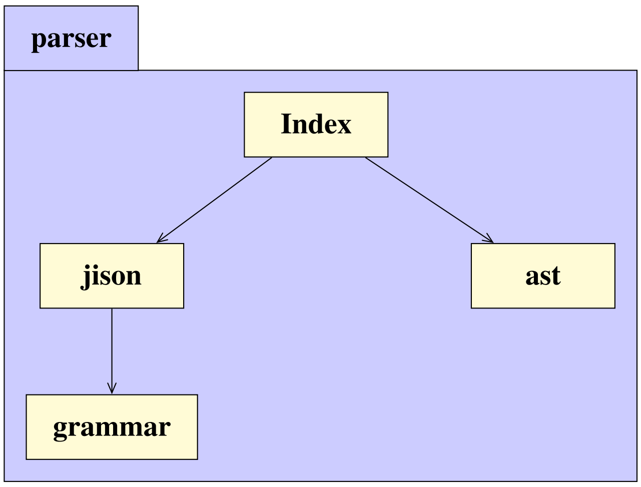


Fig. 4.3 Architecture of the parser package

The **lexical analysis** and **parsing** are done by the jison[18] library. It follows a similar interface to bison or yacc. Here's a toy example showing the grammar of a simple calculator, accepting expressions of the form $4+2-1$, $4*2$ and so on:

```
/* description: Parses end executes mathematical expressions. */

/* lexical grammar */
%lex

%%
\s+                   /* skip whitespace */
[0-9]+("."[0-9]+)?\b  return 'NUMBER';
"*"                   return '*';
"-"                   return '-';
"+"                   return '+';
<<EOF>>               return 'EOF';

/lex

/* operator associations and precedence */

%left '+' '-'
%left '*'

%start expressions

%% /* language grammar */

expressions
    : e EOF
        {print($1); return $1;}
    ;

e
    : e '+' e
        {$$ = $1+$3;}
    | e '-' e
        {$$ = $1-$3;}
    | e '*' e
        {$$ = $1*$3;}
    | NUMBER
        {$$ = Number(yytext);}
    ;
```

Once the parsing finished successfully, it outputs an AST of the program, ready to be treated.

The $C--$ parser is given in Appendix A.

4.1.2 Semantics package

This package runs just before the interpreter. Typical compile-type checks and AST pre-processing tasks are done in this module such as type checks and instruction coherence. The execution of the compiler can end here if the user specifies it.

4.1.3 Interpreter package

This module is in charge of running the C−− program once the parse and semantics phases have finished successfully. The execution follows the same flow as a typical C++ program.

First, the *main* function is called and is immediately initialized by the Function module. Once the function is initialized the Runner begins executing the instructions of the program, calling the Expression module to interpret the Expressions contained within the instructions. This process goes on until the end of the program is reached. Here's a representation of the dependencies within the interpreter package. Figure 4.4 shows a graphical representation of the module.

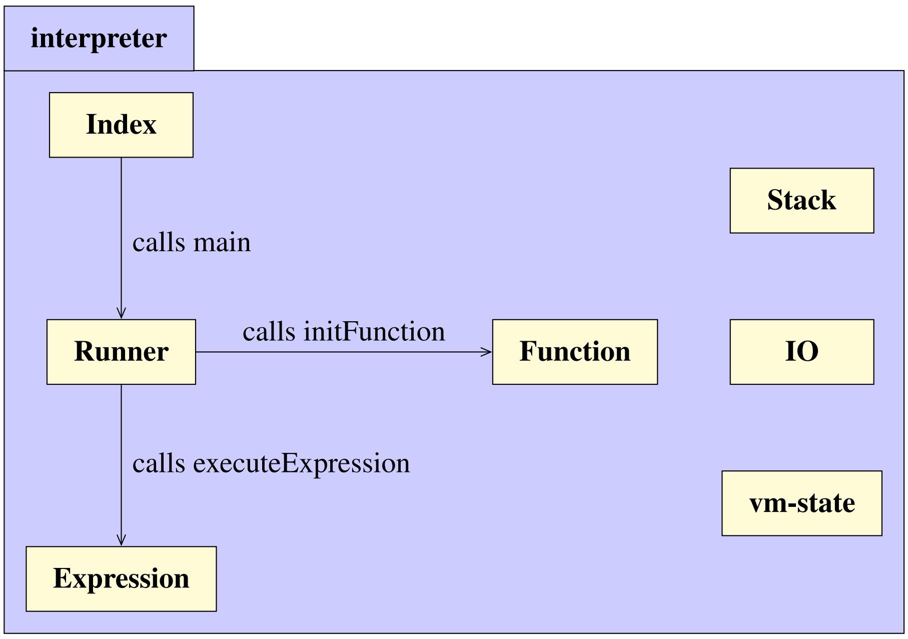


Fig. 4.4 Architecture of the interpreter package

A more detailed description is written below.

Runner: the runner module executes the instructions of the program.

Function: the function module maps the function IDs to the corresponding ASTs and initializes the Stack Activation Record when a new function call is hit.

Expression: this module computes the expressions of the program.

Stack: contains the memory of the program, segmented by Activation Records and Scopes.

IO: This module is in charge of the Input/Output operations within C−− VM.

vm-state: Handles internal data of the Virtual Machine: *instruction queue* — used by the Runner to consume the instructions, *data stack* — used by the Expression module do consume the expression tokens, *call stack* — used by Expression and Runner to exchange information about the result of the expressions, *return stack* — once a function returns a value it is pushed here, *cin stack* — used to handle the input data of the program.

4.1.4 Debugger package

This is an auxiliary module that handles debugger features, such as stepping in, out or over a function. This module is called once the user runs the debugger mode for a program.

4.2 Frontend architecture

The web page acts as an interface between the compiler and the user. It is organized under a number of external libraries to handle the multiple functionalities the backend offers, plus two scripts that interface the backend with the fronend. Figure 4.5 pictures the architecture of the webpage.

jQuery Library: jQuery [20] is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling and animation much simpler with an easy-to-use API that works across a multitude of browsers. This library is used to handle the events of the web page, such as buttons.

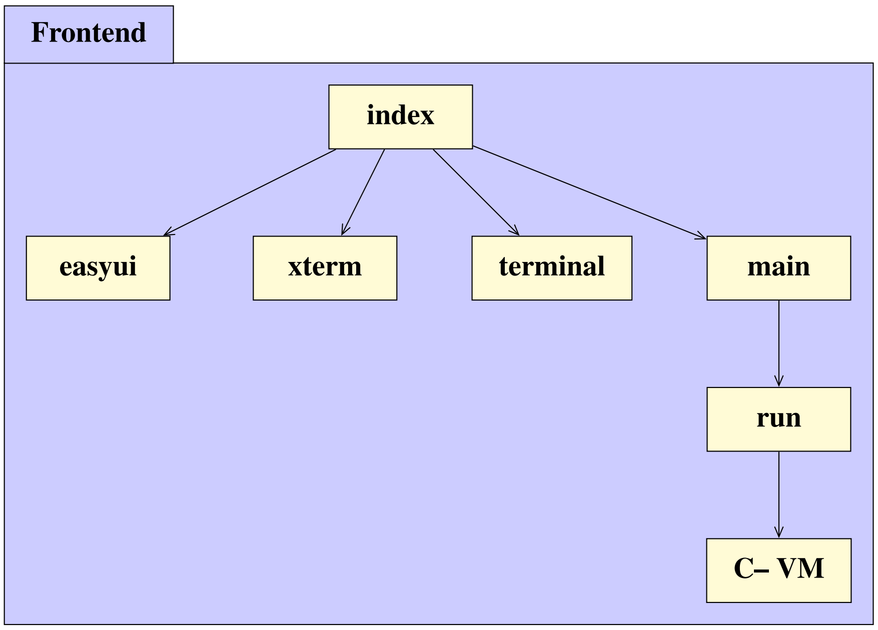


Fig. 4.5 Frontend architecture

EasyUI Library: EasyUI [19] is a collection of user-interface plugins based on jQuery. EasyUI provides essential functionality for building modern, interactive, javascript applications.

Ace Editor Library: Ace.js [1] is a customizable editor that allows for syntax highlighting and breakpointing lines of code.

jQuery Terminal Library: jQuery Terminal Emulator [21] is a plugin for creating command line interpreters. It is used to interface the input and output of the program to the user.

main script: interfaces the web page and the compiler.

run script: loads the C−− compiler.

4.3 Summary

This chapter showcased the design of the architecture of the C−− 1.1 system, from the most general components to the most particular one.

Part III

Implementation

Chapter 5

Memory stack

This chapter explains how the stack will work in relation to the debugger.

While reading this section, please refer to section 3.1.6 for a more detailed explanation of how the stack behaves in a C−− program. Moreover, this chapter introduces two new concepts:

Activation Record. it is the data structure that composes a call stack, created when a new function call is executed. It is generally composed of local variables, parameters passed by the caller and the return address to the caller. The Stack is composed by multiple Activation Records.

Scope. refers to the visibility of variables. In other words, which parts of your program can see or use it. Generally, a new scope is created when entering in a conditional block or loop. Scopes live inside an Activation Record.

Having defined these two concepts, we can now talk about the operations supported by the stack:

- Declare a variable: declare a new variable in the current activation record. In case of not setting a value to the newly declared variable, it is left uninitialized.
- Set a value to an existing variable.
- Push/Pop an Activation Record.
- Open/Close an scope.

5.1 Data visualization

The stack will be used by the user to visualize and modify the variables of the program while debugging the program.

Before an instruction is executed, a copy of the stack is sent via the public API to the frontend of the application. This public view of the stack includes all the activation records of the running program, as well as the open scopes.

If the user edits a value of the stack, the changes will be reflected in the memory of the running program once the execution is resumed.

5.2 User malfunctioning and protections

There exists the possibility that the user modifies variable values that can cause an error to the program, causing it to stop its execution.

The responsibility of modifying the stack is left to the user and a minimal number of checks is done by the system.

5.3 Future modifications

The stack will need to handle memory references and pointers in futures releases of the C–VM, so a better encapsulation of the values will be needed.

By the end of the thesis the system will support variable encapsulation to facilitate better memory management for the future releases of the VM.

5.4 API

pushActivationRecord: (funcName, paramIds)

Description: creates a new Activation Record and pushes it on top of the stack

Parameters:

- funcName [String]: name of the function
- paramIds [String]: array with the name of the function parameters

popActivationRecord

Description: pops the most recently pushed Activation Record from the stack

defineVariable: (name, type, value = null)

Description: defines a new variable

Parameters:

- name [String]: name of the variable
- type [String]: type of the function
- value [Number or String]: value of the variable. If no value is provided defaults to *null*

getVariable: (name)

Description: gets a variable from a given name

Parameters:

- name [String]: name of the variable

Return value: variable value [Number or String]

Throws: when the variable hasn't yet been assigned

setVariable: (name, value)

Description: sets the value of a variable

Parameters:

- name [String]: name of the variable
- value [Number or String]: value of the variable

openNewScope: ()

Description: opens a new scope in the current Activation Record

closeScope: ()

Description: closes a new scope in the current Activation Record

5.5 Summary

In this chapter, it is been explained the additions to the stack module to accommodate the new features of the system. A first iteration that implemented the most basic features of every component:

- An API of **editing the content** of the stack.
- **Refactoring** the Stack module to better accommodate new features.

Chapter 6

The debugger

The debugger is the main component being developed in this thesis. Several challenges appeared when building the debugger. The most notorious one is that JavaScript (and CoffeeScript) offers no control over other processes, nor has proper multithreading.

6.1 Javascript lack of multithreading

Traditionally, Javascript (JS) was intended for short and computational-light pieces of code. If someone needed to have expensive calculations going on, those were deployed on a server. The idea of a JS+HTML app that ran in the browser for long periods of time doing non-trivial things was absurd.

Of course, the panorama has changed a lot in the recent years as non-traditional web apps are flooding the market. But browsers are taking their time to catch up, most of them are designed around a single-thread model, and changing that is hard.

For this reason, writing a debugger in JS requires a well thought design. In the following sections I am going to explain the process of implementing a debugger-friendly architecture in great detail.

6.2 Architectural changes

As stated in section 6.1, the project's architecture has to go under a series of changes in order to build a reliable debugger.

The goal of these changes is to preserve the internal state of the Virtual Machine when the execution of a program pauses for long periods of time. For example, waiting for the user to input data, or pausing the program when a breakpoint is reached.

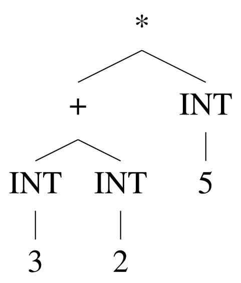
6.2.1 Expressions

Expressions are composed by numbers, operators, parenthesis and function calls. They are the minimum component in the language grammar and they are generally included as part of an instruction (see 6.2.2).

Internal representation of expressions

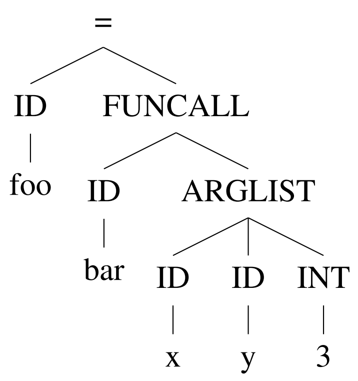
Expressions are represented by an Abstract Syntax Tree (AST). The following examples are valid expressions in the language, with the subsequent ASTs:

$$(3+2)*5$$



A slightly more complex expression would go as follows:

$$foo = bar(x, y, 3)$$



Computation of expressions

Expressions, due to the nature of the AST, are usually computed recursively. However, due to our project constraints this computation must be done iteratively with an explicit stack instead of using the Javascript one.

6.2.2 Instructions

After refactoring the Expressions module, similar modifications on the Runner module will be required. The Runner module is in charge of running instructions line by line (refer to section 3.1.1 for more details).

Unwrap of instruction blocks

Instructions are grouped under instruction blocks. Instructions blocks are just simple arrays of instructions. Here's a block of instructions inside an if statement:

```
if (condition) {
    BLOCK_INSTRUCTIONS;
}
```

Please note that *BLOCK_INSTRUCTIONS* is the name of the token used to define an instruction block. The same principle applies for all flow control statements as well as functions.

In the 1.0 version of the Virtual Machine, function blocks were accessed in a recursive manner, so unwrapping the instructions contained in the block was unnecessary.

When transitioning the Runner from a recursive to an iterative approach the blocks of instructions will need to be unwrapped and added to the instructions queue.

However, this approach has some edge cases that need to be taken into account:

- **Conditional statements:** the block of instructions will only be unwrapped if the condition of the *IF* clause has a true value. An internal instruction will be added to at the end of the instruction block to mark the end of the scope (see Chapter 5).
- **Loops:** they behave similar to the conditional statements in the sense that the instruction block is unwrapped if the initial condition is fulfilled. However, it will be taken into account that the scope will be opened once.

A single Runner

The Runner class will have only once instance across the project to ensure the state of execution of a program is preserved.

6.2.3 Function calls

Functions are the outermost component in the VM architecture. All functions in a C++ program are called from an expression, with the *main* function being the only exception. Due to the nature of the function calls, the initial architecture had to recursively call the function module. In the end, a program had multiple $Function-> Instruction-> Expression$ recursive levels (see figure 3.2).

Making instructions the main component

Converting the instructions into the main unit of a C−− program will remove an unnecessary layer of abstraction and better decouple the function calls from expressions.

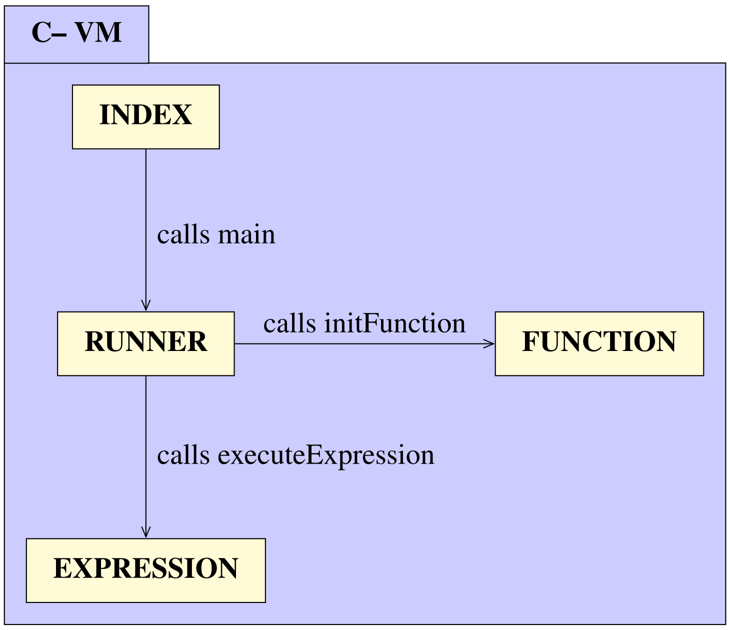


Fig. 6.1 C−− 1.1 system's architecture

The following UML Diagram shows the architecture after applying all the changes stated above.

The Function module is now operating as an auxiliary module that helps inject the new instructions from a function call into the Runner, avoiding creating new instances of Runner every time a function call is made.

Decoupling function calls from expressions

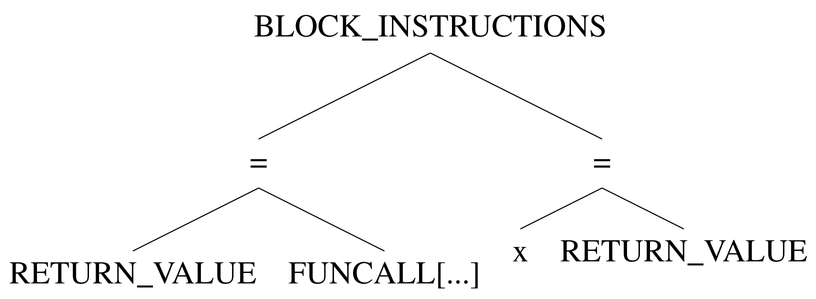
Function calls are taken out of expressions and new a instruction is generated. For example, the following expression:

```
x = foo(y);
```

is converted to:

```
RETURN_VALUE = foo(y);
x = RETURN_VALUE;
```

resulting in the following AST (simplified):



The *RETURN_VALUE* variable is an internal representation of the return value of a given function, and is never shown to the user.

With this modification function calls are essentially instructions and can be called from the Runner class. Each time a function is called, the block of instructions contained of the function is added to the instruction queue of the Runner, letting the system to be paused by the user at any time and without loss of data.

6.2.4 Polishing the details

During the design of the new architecture some language features behaved differently from the original iteration.

Please note that both cases treated in this section do not currently impact the outcome of a C−− program. However, as stated in earlier chapters, this project is meant to be continued by other students and I considered as best practices to address the following cases.

Lazy evaluation with function calls

Before commenting on the issue, let's consider the following example:

```
bool b = foo() && bar();
```

Imagine both functions foo and *bar* modify the state of the program in some manner. In standard C++ the function *bar* would be executed if, and only if foo returned a $true$ value.

Now let's consider the internal representation of the above code in our system (read section 6.2.3):

```
RET_VAL_1 = foo();
RET_VAL_2 = bar();
bool b = RET_VAL_1 && RET_VAL_2;
```

As we can see, both functions foo and *bar* are be executed before evaluating the boolean expression. This behavior is critical for functions that modify the state of the program outside of their own scope, so safety measures must be applied to contemplate this case.

After fixing the mentioned issue, this is the resulting internal code:

```
RET_VAL_1 = RET_VAL_2 = false;
RET_VAL_1 = foo();
if (RET_VAL_1) {
    RET_VAL_2 = bar();
}
bool b = RET_VAL_1 && RET_VAL_2;
```

A similar algorithm takes place for the *OR* operator.

References and function calls

The following program would perform very differently between the new and old implementations of the Expression module:

```
int foo(int &x) {
    x = 1;
    return x;
}

int x = 0;
x = foo(x) + x;
```

Modifying the value of a variable and immediately accessing it in the same expression is considered an undefined behavior under the *The C++ programming language* book[28], page 492. Thus, it was decided to not correct the change of behavior.

6.2.5 AST Flags and Line Counting

Additional meta-data will need to be stored in the nodes of the AST:

- *a* boolean indicating if the node is the root of an instruction.
- *a* boolean indicating the node does not have more AST child nodes.

These new attributes will help to better process the AST during the interpretation of a program.

Additionally, the Jison[18] locations API is being used to map the original source code to its AST representation.

6.2.6 Iterators and its relationship with the public API

After refactoring the project, a **minimal API** is to be built to interact with the frontend of the application. This API involves the use of **iterators** and **callbacks** to access the VM information.

6.3 API

6.3.1 Public API

hooks.modifyVariable: (varName, value)

Description: modifies the content of a variable

Parameters:

- varName [String]: name of the variable
- value [Number or String]: new value of the variable

actions.stepOut

Description: steps out of a function

actions.stepOver

Description: steps over a function

actions.stepInto

Description: steps into a function

6.4 Summary

In this chapter, it is described the implementation of the debugger. The implementation process featured:

- **Refactor** the Expression, Runner and Function modules to facilitate the pausing of a program in the middle of its execution.
- **Solve** problems that raised from this new implementation.
- Modification of the **AST** to **show useful information**.
- Build a minimal **API to interact with the frontend** of the VM.

Chapter 7

Non-blocking I/O

This chapter will talk about the operations of input and output and how they change between the versions 1.0 and 1.1 of the project.

7.1 Initial problem

In the 1.0 of the project the I/O calls were handled synchronously. The **input** of the program was given **before its execution**. On the other hand, the **output** was delivered once the program **finished running**.

This simplistic approach is far from how real C++ programs behave and it will be improved in the 1.1 version. I will be using different approaches when handling an input and an output. Both approaches will be explained in the upcoming sections.

7.2 Asynchronous outputs

The output of the program is handled by passing a callback to the interpreter. The following subsections explain in detail what callbacks are and how they are used in the project.

7.2.1 CoffeeScript callbacks

Before diving to the implementation of asynchronous outputs I am going to explain how functions, and more precisely callbacks, work in CoffeeScript.

In CoffeeScript, **functions** act as **first class citizens**. That is, functions are treated as a type just like Numbers, Strings, Arrays and Objects. For this reason functions can be passed as parameters in other functions, and functions can be assigned to variables. A function passed as a parameter is called a **callback**, and is often used when writing libraries and want the user to perform some action after an amount of work.

One example is showing an alert after pressing a button. Here's the example written in code:

```
$("#button").click(() -> alert("Button Clicked"))
```

As you can see, a function is passed by parameter and it will be fired when the button is pressed. This approach works really well when handling events of all sorts, just like the example above.

7.2.2 The Event endpoints

An API was developed using the native concurrency in CoffeeScript. In essence, the endpoints accept a callback function by parameter that handle the output of the program.

The code below shows an schematic of how it is done:

```
events = {
    onstdout: (callback) -> interpreter.onstdout(callback)
}
```

The interpreter then handles the callback when needed: when running a C−− program and a **cout** instruction is reached, the callback is fired.

7.3 Asynchronous inputs

Building an asynchronous approach for inputs was a process much more complex than the done with outputs. CoffeeScript has no threading support nor is capable of managing multiple processes. In essence, a program in CoffeeScript can't be paused, so major software components of the Virtual Machine had to be refactored in order to implement *async* inputs.

7.3.1 Keeping the state

In Chapter 6, an instruction queue had to be implemented in order to support debugging functionality. The reason was that the state of the program had to be kept in order to pause the virtual machine at any given time.

I will take advantage of this feature to also pause the execution of the virtual machine when the C−− program requires input to proceed with the execution. When a **cin** instruction is reached the interpreter looks at the input buffer and if it's empty it pauses its execution waiting for the buffer to be refilled.

The following section will talk about the input buffer in more detail.

7.3.2 Input buffering

When the user sends input data to the interpreter, it is kept in a buffer waiting to be consumed. Doing it this way, the user can introduce all the input at the beginning of the program, even though not all the input is consumed at the same time.

Imagine the following toy program:

```
int square_sequence() {
    cin >> n;
    for(int i = 0; i < n; ++i) {
        int in;
        cin >> in;
        cout << in*in << endl;
    }
}
```

The user can enter all the data in the beginning of the program and the input calls will keep consuming the data until the program has ended or the buffer gets empty.

7.4 API

7.4.1 I/O module

reset

Description: resets all the streams. These streams are the stdout, stdint, stderr and interleaved streams

output: (stream, string)

Description: fills the stream *stream* with content from *string*. If *stdoutCB* 7.4.1 is not null, a callback is fired.

Parameters:

- stream [String]: name of the stream
- string [String]: content to put in the stream

setInput: (stream, input)

Description: sets data from the input function into a stream

Parameters:

- stream [String]: name of the stream
- input [String]: content to put in the stream

getWord: (stream)

Description: gets a word from a stream

Parameters:

- stream [String]: name of the stream

Return value: a word [String]

unshiftWord: (stream, word)

Description: used to refill when there is some leftover that was not parsed from the word

Parameters:

- stream [String]: name of the stream
- word [String]: leftover

getStream: (stream)

Description: used to refill when there is some leftover that was not parsed from the word

Return value: a stream [Array]

setStdoutCB: (cb)

Description: sets a callback to the standard output stream

Parameters:

- cb [Function]: callback

isInputBufferEmpty: (stream)

Description: checks if the input buffer is empty

Parameters:

- stream [String]: name of the stream

Return value: returns a [Boolean] depending on the state of the stream

7.4.2 Public API

hooks.setInput: (stream, input)

Description: sets data from the input function into a stream

Parameters:

- stream [String]: name of the stream
- input [String]: content to put in the stream

hooks.isInputBufferEmpty

Description: checks if the input buffer is empty

Parameters:

- stream [String]: name of the stream

Return value: returns a [Boolean] depending on the state of the stream

events.onstdout: (cb)

Description: sets a callback that will fire after an standard output event.

Parameters:

- cb [Function]: callback

7.5 Summary

In this chapter, *async* I/O operations were implemented. The implementation process featured:

- Use **callbacks** to fire output events.
- **Keeping the state** of the virtual machine to **stop the execution** of a program at any time.
- Having an **input buffer** where the input data goes before it is used.

Chapter 8

Frontend

The frontend of the project is a User Interface meant to connect the user with the C−− interpreter.

8.1 Overview

We will need the following for building a webpage able to connect the user with the interpreter:

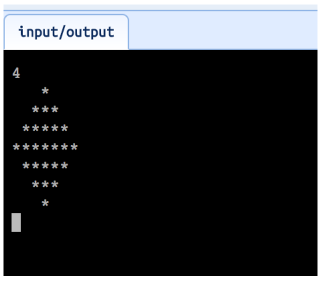
- A code editor that lets put breakpoints at any line.
- A terminal to treat input and output operations.
- A contextual menu for advanced options.
- A quick access menu for common actions.
- A grid to display the debugger information.
- A status bar showing the state of the execution of a program.

8.2 Components

The following subsections will explain in more detail the sections of the frontend.

8.2.1 Terminal

The terminal serves as an input and output source for the programs. The terminal is just a text processor that reads the input from the user and sends it to the compiler. The Terminal is built with the help of jquery.Terminal [21] library.



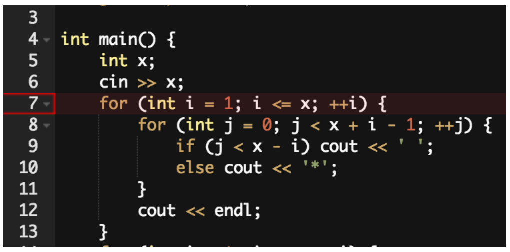
```
4
   *
  ***
 *****
*******
 *****
  ***
   *
```

Fig. 8.1 Input and Output terminal

Compilation and execution errors are displayed in bold, to differentiate themselves from the actual output of the program.

8.2.2 Editor

The code editor allows the user to write C−− code. The editor supports syntax highlighting for better readability, plus offers auto-completion of parenthesis and brackets. The editor supports breakpoints as well, which are used to debug programs. The editor is a customized version of Ace Editor [1] library, which highlights C++ code.

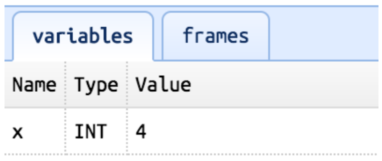


```
int main() {
    int x;
    cin >> x;
    for (int i = 1; i <= x; ++i) {
        for (int j = 0; j < x + i - 1; ++j) {
            if (j < x - i) cout << ' ';
            else cout << '*';
        }
        cout << endl;
    }
```

Fig. 8.2 The code editor

8.2.3 Variables and Frames grid

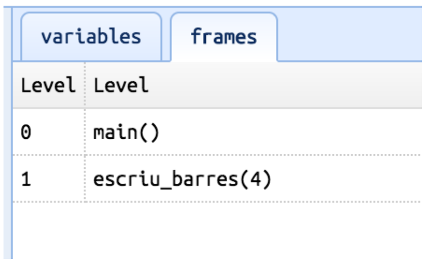
The grid allows for a quick variable edition when the program execution is paused at a breakpoint. With a double click in the **Value** cell, the user can modify the value of the variable, provided the new value is of the same type.



Name	Type	Value
x	INT	4

Fig. 8.3 Variables grid, which allows editing the value of a variable

Furthermore, the user can see the function frames and the initial value of the parameter of a function call.



Level	Level
0	main()
1	escriu_barres(4)

Fig. 8.4 Frames grid

For the grids EasyUI [19] framework is used.

8.2.4 Controls and Status Bar

All the controls of the webpage are displayed in the contextual menu. The menu is divided in 4 categories: **File**, **Execute**, **Debug** and **About**.

Fig. 8.5 Contextual menu, with all the user options

The most common controls are displayed in a long frame of buttons as well. Next to the buttons there is the status bar, which displays the current status of the compiler. The possible statuses are the following: Running, Waiting for input, Paused, Program finished successfully, Compilation error, Execution error and Program killed.

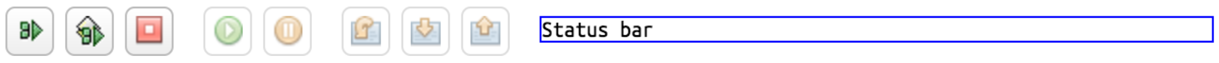


Fig. 8.6 Quick access bar (left) and status bar (right)

Both contextual and quick access menu are components of EasyUI [19].

8.3 Summary

This chapter deals with the different components that make up the frontend of the project.

Chapter 9

Deployment

The deployment of the entire system is held under two stages: first releasing an stable version of the compiler and second, deploying the frontend to a HTTP [14] server. Gulp [16] is used in both parts of the project. Gulp is an streaming system that's capable of performing a series of synchronous tasks, such as compiling a project or resolving dependencies.

9.1 Compiler deployment

Gulp

Gulp is configured with two modes here. The *default* mode is meant for production only, and minifies the resulting JavaScript. The *dev* mode does not minify the code, which improves readability and debugging. Both modes compile the code from CoffeeScript to JavaScript, and apply the Browserify [5] plugin.

Browserify

As stated in their webpage, Browserify lets you import modules NodeJS modules in the browser by bundling up all of your dependencies.

Github releases

After compiling and browserifying the resulting code, Github is used to host the releases of the project. Stable releases are hosted in the following url.

9.2 Frontend deployment

Gulp

Just like in the backend, Gulp is configured with two modes. The *default* mode downloads the latest release of the compiler from Github and compiles the whole project, which is now ready to be statically served in a HTTP server. With the *dev* mode however, the programmer has to put the compiler's JS code in the project folder. This mode opens up a local server to watch the changes.

Hosting the website

Once the frontend is compiled, it can be served as an static webpage in a server. In this project, we chose to run it on a Linux machine. The frontend is hosted at https://cmm.jutge.org.

9.3 Summary

This chapter talked about the deployment steps needed to compile the project. The compilation process involved two separate tasks: backend and frontend.

Part IV

Evaluation

Chapter 10

Testing and Evaluation

This chapter tests and evaluates the C−− compiler. For the testing, a testing framework is used and for the evaluation a set of algorithms is executed across multiple languages. Moreover, a set of informal usability interviews have been made.

10.1 Testing

Multiple sample programs evaluate different features of the compiler. These programs can be found in the webpage of the project.

10.1.1 Continuous Integration

CI practices are used for the project. More precisely Travis CI [29] is used as a tool for merging all developing working copies several times a day. All tests must be passed before each merge. For unit testing, Mocha [24] is used.

10.2 Evaluation

The Fibonacci and primality algorithms are tested in these sections. Appendix B has the specific implementations of each algorithm for each language.

10.2.1 Is Prime

The first algorithm to test is an Is Prime detector. The algorithm consisted in delivering if a number is prime or not, in intervals between one and a few millions:

N	C– 1.0	C– 1.1	C++	NodeJS	Python 3.5
1	4.25	5.75	0.13	0.11	6.33
2	15.39	17.56	0.34	0.33	17.11
3	23.45	25.43	0.58	0.59	29.43
4	49.31	52.74	0.86	0.89	44.31
5	75.16	79.14	1.17	1.21	60.12
6	101.01	106.65	1.51	1.53	82.60
7	139.87	143.82	1.88	2.02	97.01
8	174.72	160.71	2.27	2.33	116.28
9	207.58	197.42	2.67	2.76	147.68
10	240.43	241.67	3.09	3.22	184.52
11	276.28	271.02	3.72	3.65	208.02
12	311.14	314.72	4.08	4.18	226.79
13	339.39	342.10	4.76	4.56	256.42
14	371.05	381.06	5.10	5.06	288.00
15	405.10	414.52	5.68	5.54	311.24

Table 10.1 isPrime timetable (*N* in millions, times in seconds)

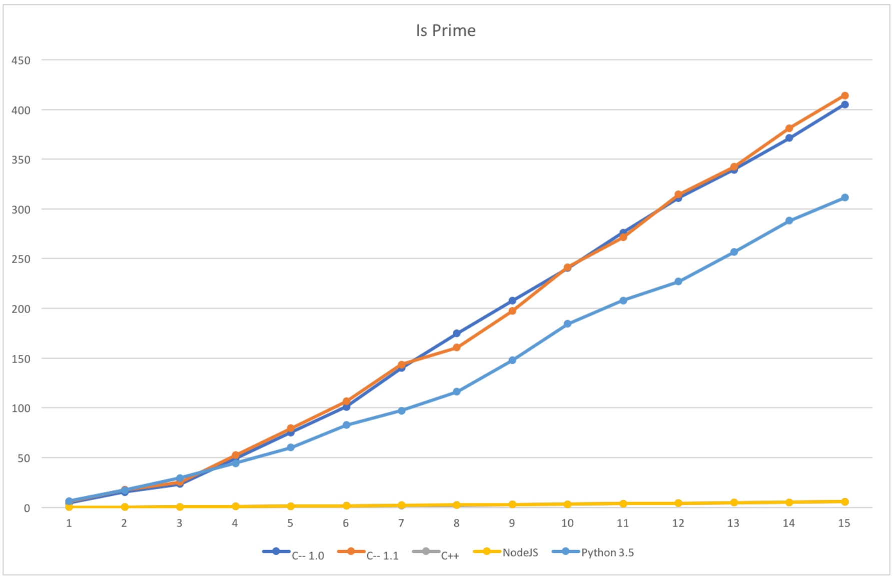


Fig. 10.1 Time (sec) - Y axis, N - X axis

Here we can appreciate that the C−− implementation is slightly more stable than the Python implementation. This can be due to optimizations triggered by the Python interpreter, than can often speed up the execution of the code. Again, the 75-80 ratio of C−− execution time respect the C++ implementation holds. The JavaScript (Node.js) implementation is as fast as the C++ one.

10.2.2 Fibonacci

The second algorithm to test is the Fibonacci sequence. The algorithm consisted in computing the Fibonacci numbers from 25 to 50, in intervals of 5. These are the results:

N	C– 1.0	C– 1.1	C++	NodeJS	Python 3.5
25	2.05	2.45	0.01	0.01	0.03
30	61.29	67.75	0.01	0.01	0.37
35			0.09	0.11	4.15
40			0.75	1.22	44.91
45			8.24	13.72	
50			89.42	164.41	

Table 10.2 Fibonacci timetable (time in seconds)

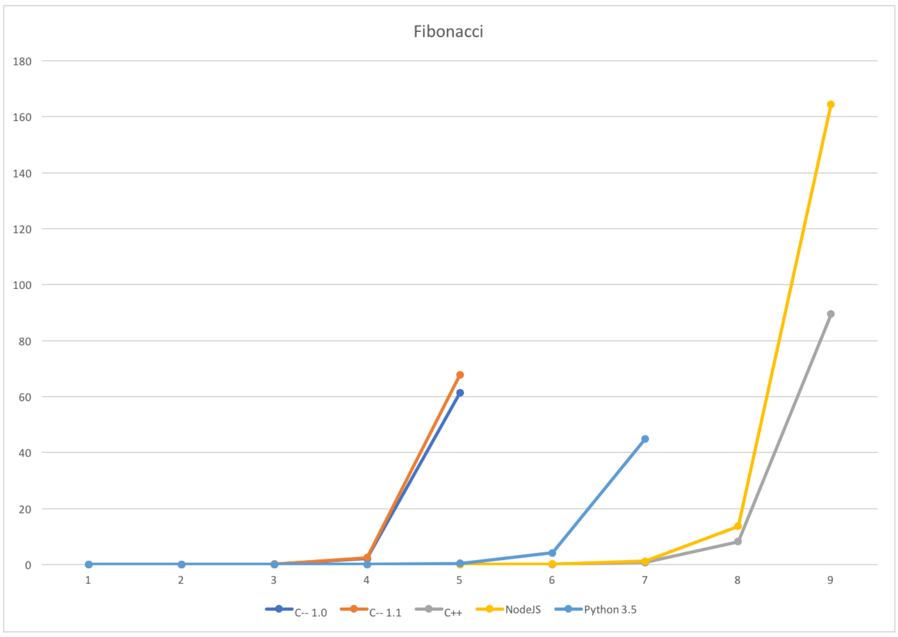


Fig. 10.2 Time (sec) - Y axis, N - X axis

As we can observe, the C−− implementation is significantly slower than the other languages. To be more precise, the C−− implementation is 75 to 80 times slower than the C++ implementation, which is the same ratio than in the previous experiment.

On top of that, we can see that the 1.1 version is a bit slower than the 1.0. The reason would be the added overhead of using explicit data-structures instead of built-in features offered by JavaScript, which would be more optimized.

Another interesting aspect about this data is the discrepancy between the JavaScript and the C++ implementations, unlike the *Is Prime* algorithm. After researching over the Internet, I concluded that the function calls are the responsible of this discrepancy. Function calls in JavaScript are more expensive than in C++, and given that this Fibonacci implementation uses a recursive approach , the execution time of the JavaScript implementation is greater than expected.

10.2.3 Collatz sequence

The last algorithm testes is the Collatz Sequence. The Collatz Sequence is a convergent series that is conjectured to always reach one:

N	C– 1.0	C– 1.1	C++	NodeJS	Python 3.5
10000	198.29	215.42	19.38	4.4	164.52
15000	596.56	657.21	27.67	5.13	247.58
20000	971.03	1156.19	37.92	8.46	354.97
25000	2101.89	2302.79	43.59	9.72	455.09
30000	2934.91	3236.58	48.56	12.77	545.98
35000	4085.61	4044.32	53.41	14.74	675.46
40000	4725.67	4853.73	63.31	15.98	732.12
45000	5653.82	5876.82	75.27	19.3	828.85
50000	6157.45	6289.31	81.74	20.87	926.22

Table 10.3 Collatz sequence timetable (time in milliseconds)

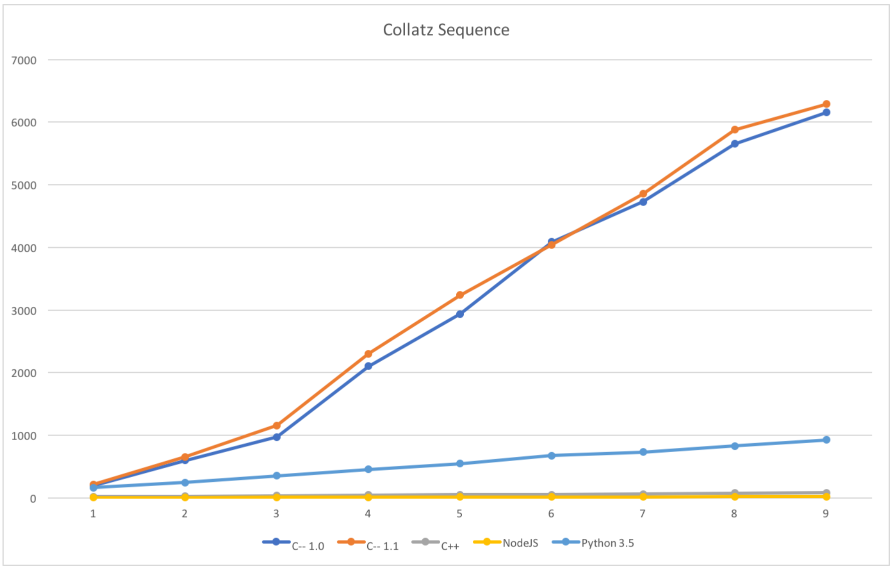


Fig. 10.3 Time (msec) - Y axis, N - X axis

We can appreciate that the C—— implementation is a bit unstable compared to the other tests. This can be due the difference in the time scale between this test and the other two. The 75-80 ratio of C—— execution time respect the C++ implementation stays with higher values of N. The JavaScript (Node.js) implementation is slightly faster than the C++ one, due to the GCC compiler not being in O1/O2/O3 mode.

10.3 Usability

For the web interface, several usability interviews to different people were made: four students and two teachers. Most of the feedback mentioned that it was difficult to access the controls to run and debug a program. Moreover, the testers were confused by the use of the word 'Terminal' inside the website as they thought they could use UNIX commands. Another complain pointed the lack of feedback of the state of the program. From the feedback obtained in the interviews, several improvements were made:

Quick Access menu. The buttons from this menu are now deactivated when they can't be used.

Terminal tab. The tab 'terminal' has been renamed to 'output/output' to better represent what it is.

Status bar. The status bar has been moved from the lower side to just above the terminal, next to the Quick Access menu.

10.4 Summary

This chapter consisted in analyzing algorithm execution times across multiple languages and compare the times to the C– implementation, and performing usability tests across different people and improving the website from the feedback collected from them.

The conclusions we can extract from this tests is that the execution time of a program in C−− is consistent across the different tests performed. More precisely, C−− 75 times slower than a C++ implementation, which is a positive result. However, C−− execution times are very close to the Python ones, which is a positive surprise . The second point I conclude is that the 1.1 version of the compiler is generally a bit slower than the 1.0 version, due to the implementation of the debugger.

Chapter 11

Conclusion

11.1 Validation summary

This section studies whether the project's main and secondary objectives described in chapter 2.2 have been fulfilled after the implementation process.

11.1.1 Main objective

Develop a debugger. After several modifications in the VM's architecture, the Virtual Machine was able to run step-by-step and perform critical operations, such as I/O, asynchronously. The implementation details are given in Chapter 6.

Keep the code base maintainable and scalable. The platform was designed with scalability in mind:

- The Virtual Machine can work on different frontend systems.
- The infrastructure is composed of modules, which favors the addition of new features.

Moreover, the code base is maintainable after successfully applying the methodology of Chapter 2.3.

Design a new UI. The new website can be found at *https://cmm.jutge.org*. Chapter 8 explains how it has been developed in detail.

Create test programs and documentation. Test programs can be found in the project's website *https://cmm.jutge.org*. Chapters 5, 6 and 7 include the API documentation.

11.1.2 Secondary objectives

Asynchronous I/O operations. The Virtual Machine was able to perform I/O operations without freezing the UI after successfully implementing asynchronous operations in chapter 7.

IDE-like features for the website. A new website with many features was implemented in chapter 8. Also, it has been tested constantly as described in chapter 10.

Overall, all goals have been achieved, without significant delays from the original plan.

11.2 Future work

Looking into future versions of C−−, these are the most interesting features that could be implemented:

Structs and arrays: Having structs and arrays is crucial for the maturity of the language as well as for writing more complex programs. This addition would be of first priority for future versions.

References: currently all variables are passed by value, and arrays are not allowed. References would provide better memory management and a deeper insight into the language.

Better scaling: with larger and larger programs the language does not scale well. This can be a problem when trying to emulate large code bases, although it is not the primary use of C−−. Improving in this part could led to more complex program executions.

Libraries: the language does not provide an out-of-the-box modularity feature. Another useful feature to look for would be having STD libraries, implemented in our own language.

11.3 Personal conclusions

This project represented my second contact with the compiler ecosystem. Thus, I spent the first weeks of the project researching about compiler architectures and getting used to the common software patters that are used.

Also, thanks to my project's supervisor I gained advanced knowledge about compiler theory that helped me develop a better product. Moreover, the weekly meetings have helped me advance with the project faster and more efficiently.

There are only two courses in the official Computer Science curriculum from the Barcelona School of Informatics that even mention Compiler theory, and just one of them is compulsory. While it is clear that this topic is not needed by most jobs, I believe it should be covered more extensively in the CS Curriculum.

During the project, I learned many software patterns that are far from the traditional models we use every day (like Callbacks and Asynchronousny), and I scratched the surface of micro-code design when optimizing the AST to implement the debugger.

References

[1] *Ace is an embeddable code editor written in JavaScript*. https://ace.c9.io/#nav=about (visited on 01/11/2017).

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. “Compilers: Principles”. In: *Techniques, and Tools* (1986).

[3] *Bison - GNU Project - Free Software Foundation*. https://www.gnu.org/software/bison/ (visited on 12/31/2016).

[4] *Bootstrap is a popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web*. http://getbootstrap.com/ (visited on 01/11/2017).

[5] *Browserify*. http://browserify.org/ (visited on 12/29/2016).

[6] *Calculation of CO2 emissions*. http://www.sunearthtools.com/tools/CO2-emissions-calculator.php (visited on 10/21/2016).

[7] *Cheerp - C++ for the Web*. http://leaningtech.com/cheerp/ (visited on 10/20/2016).

[8] *Clang: a C language family frontend for LLVM*. http://clang.llvm.org/ (visited on 10/20/2016).

[9] *CoffeeScript*. http://coffeescript.org/ (visited on 12/10/2016).

[10] *CSS is a style sheet language used for describing the presentation of a document written in a markup language.* https://en.wikipedia.org/wiki/Cascading_Style_Sheets (visited on 01/11/2017).

[11] *Dart programming language*. https://www.dartlang.org/ (visited on 12/10/2016).

[12] *Emscripten*. http://kripken.github.io/emscripten-site/ (visited on 10/20/2016).

[13] Christian Felber. *The economy of the common good*. Deusto SA Editions, 2011. ISBN: 9788423412808.

[14] Roy T. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. 1999. https://tools.ietf.org/html/rfc2616 (visited on 01/25/2017).

[15] *GCC - GNU Compiler Collection*. https://en.wikipedia.org/wiki/GNU_Compiler_Collection (visited on 10/20/2016).

[16] *gulp.js - the streaming build system*. http://gulpjs.com/ (visited on 12/29/2016).

[17] *HTML is the standard markup language for creating web pages and web applications.* https://en.wikipedia.org/wiki/HTML (visited on 01/11/2017).

[18] *Jison*. http://zaa.ch/jison/ (visited on 11/09/2016).

[19] *jQuery EasyUI framework helps you build your web pages easily*. http://www.jeasyui.com/ (visited on 01/11/2017).

[20] *jQuery is a fast, small, and feature-rich JavaScript library to work with the DOM*. https://jquery.com/ (visited on 01/11/2017).

[21] *jQuery Terminal emulator plugin*. http://terminal.jcubic.pl/ (visited on 01/11/2017).

[22] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. 1988. ISBN: 0201889544.

[23] *Mendeley*. https://www.mendeley.com (visited on 10/20/2016).

[24] *Mocha - the fun, simple, flexible JavaScript test framework*. https://mochajs.org/ (visited on 10/20/2016).

[25] *Overleaf: Real-time Collaborative Writing and Publishing Tools with Integrated PDF Preview*. https://www.overleaf.com/ (visited on 10/20/2016).

[26] *Rusthon*. http://rusthon.github.io/Rusthon/ (visited on 10/20/2016).

[27] *Scrum*. https://www.scrumalliance.org/why-scrum (visited on 10/20/2016).

[28] B Stoustrup. *The C++ programming language*. 1997. ISBN: 0201889544.

[29] *Travis CI*. https://travis-ci.org/ (visited on 10/20/2016).

[30] *TypeScript - JavaScript that scales*. https://www.typescriptlang.org/ (visited on 12/10/2016).

[31] *Yacc: Yet Another Compiler-Compiler*. http://dinosaur.compilertools.net/yacc/ (visited on 12/31/2016).

Appendix A

Language grammar

```
/* description: C-- language grammar. */

/* lexical grammar */
%lex
%%

"//".*                  /* ignore comment */
"/*"(.|\n|\r)*?"*/"     /* ignore multiline comment */
\s+                     /* skip whitespace */
"++"                                        return '++'
"--"                                        return '--'
"+="                                        return '+='
"-="                                        return '-='
"*="                                        return '*='
"/="                                        return '/='
"%="                                        return '%='

"*"                                         return '*'
"/"                                         return '/'
"-"                                         return '-'
"%"                                         return '%'
"+"                                         return '+'

"!="                                        return '!='

"or"                                        return '||'
"and"                                       return '&&'
```

```
"not"                                      return '!'

"||"                                       return '||'
"&&"                                       return '&&'
"!"                                        return '!'

"<<"                                       return '<<'
">>"                                       return '>>'

">="                                       return '>='
"<="                                       return '<='
">"                                        return '>'
"<"                                        return '<'
"=="                                       return '=='

"="                                        return '='

";"                                        return ';'
"{"                                        return '{'
"}"                                        return '}'
"("                                        return '('
")"                                        return ')'
","                                        return ','
"#"                                        return '#'

"return"                                   return 'RETURN'

"cin"                                      return 'CIN'
"cout"                                     return 'COUT'

"endl"                                     return 'ENDL'

"int"                                      return 'INT'
"double"                                   return 'DOUBLE'
"char"                                     return 'CHAR'
"bool"                                     return 'BOOL'
"string"                                   return 'STRING'
"void"                                     return 'VOID'

"include"                                  return 'INCLUDE'
'using'                                    return 'USING'
'namespace'                                return 'NAMESPACE'
'std'                                      return 'STD'

```

```
"if"                                         return 'IF'
"else"                                       return 'ELSE'
"while"                                      return 'WHILE'
"for"                                        return 'FOR'

"true"                                       return 'BOOL_LIT'
"false"                                      return 'BOOL_LIT'
[0-9]+("."[0-9]+)\b                          return 'DOUBLE_LIT'
([1-9][0-9]*|0)                              return 'INT_LIT'
\'([^\\\']|\\.)\'                            return 'CHAR_LIT'
\"([^\\\"]|\\.)*\"                           return 'STRING_LIT'

([a-z]|[A-Z]|_)([a-z]|[A-Z]|_|[0-9])*        return 'ID'

<<EOF>>                                      return 'EOF'

.                                            return 'INVALID'

/lex

/* operator associations and precedence */
%right '+=' '-=' '*=' '/=' '%=' '='
%left '||'
%left '&&'
%left '==' '!='
%left '<' '>' '<=' '>='
%left '>>' '<<'
%left '+' '-'
%left '*' '/' '%'
%right '!' 'u+' 'u-'
%right '++a' '--a'
%left 'a++' 'a--'
%right THEN ELSE

%start prog

%% /* language grammar */

prog
    : block_includes block_functions EOF
        { return new yy.Ast('PROGRAM', [$1, $2]); }
    ;

block_includes
```

```
    : block_includes include
        {$$.addChild($2);}
    |
        {$$ = new yy.Ast('BLOCK–INCLUDES', []);}
    ;

include
    : '#' INCLUDE '<' id '>'
        {$$ = new yy.Ast('INCLUDE', [$4]);}
    | USING NAMESPACE STD ';'
        {$$ = new yy.Ast('NAMESPACE', [$4]);}
    ;

block_functions
    : block_functions function
        {$$.addChild($2);}
    |
        {$$ = new yy.Ast('BLOCK–FUNCTIONS', []);}
    ;

function
    : type id '(' arg_list ')' '{' block_instr '}'
        {$$ = new yy.Ast('FUNCTION',[$1,$2,$4,$7]);}
    ;

arg_list
    : arg_list ',' arg
        {$$.addChild($3);}
    | arg
        {$$ = new yy.Ast('ARG–LIST', [$1]);}
    |
        {$$ = new yy.Ast('ARG–LIST', []);}
    ;

arg
    : type id
        {$$ = new yy.Ast('ARG', [$1, $2]);}
    ;

block_instr
    : block_instr instruction
        {$$.addChild($2, instr=true, instrNumber=@2.first_line);}
    |
        {$$ = new yy.Ast('BLOCK–INSTRUCTIONS', []);}
    ;

instruction
    : basic_stmt ';'
    | if
    | while
    | for
    | return_stmt ';'
    | ';'
        {$$ = new yy.Ast('NOP', []);}
    ;

basic_stmt
    : block_assign
    | declaration
    | cout
    | expr
    ;

return_stmt
    : RETURN expr
        {$$ = new yy.Ast('RETURN', [$2]);}
    | RETURN
        {$$ = new yy.Ast('RETURN', [])}
    ;

funcall
    : id '(' param_list ')'
        {$$ = new yy.Ast('FUNCALL', [$1,$3], leaf=true);}
    ;

param_list
    : param_list ',' param
        {$$.addChild($3);}
    | param
        {$$ = new yy.Ast('PARAM-LIST', [$1]);}
    |
        {$$ = new yy.Ast('PARAM-LIST', []);}
    ;

param
    : expr
        {$$ = $1;}
    ;
```

```

if
    : IF '(' expr ')' instruction_body %prec THEN
        {$$ = new yy.Ast('IF-THEN', [$3, $5]);}
    | IF '(' expr ')' instruction_body else
        {$6.setIsInstr(true); $6.setInstrNumber(@6.first_line); $$ = ↩
            new yy.Ast('IF-THEN-ELSE', [$3, $5, $6]);}
    ;

while
    : WHILE '(' expr ')' instruction_body
        {$$ = new yy.Ast('WHILE', [$3, $5]);}
    ;

for
    : FOR '(' basic_stmt ';' expr ';' basic_stmt ')' instruction_body
        {$$ = new yy.Ast('FOR', [$3, $5, $7, $9])}
    ;

else
    : ELSE instruction_body
        {$$ = $2;}
    ;

cin
    : CIN block_cin
        {$$ = $2;}
    ;

block_cin
    : block_cin '>>' id
        {$$.addChild($3);}
    | '>>' id
        {$$ = new yy.Ast('CIN', [$2], leaf=true);}
    ;

cout
    : COUT block_cout
        {$$ = $2;}
    ;

block_cout
    : block_cout '<<' expr
        {$$.addChild($3);}
    | block_cout '<<' ENDL
        {$$.addChild(new yy.Ast('ENDL', []));}
    | '<<' expr
        {$$ = new yy.Ast('COUT', [$2]);}
    | '<<' ENDL
        {$$ = new yy.Ast('COUT', [new yy.Ast('ENDL', [])]);}
    ;

instruction_body
    : instruction
        {$1.setIsInstr(true); $1.setInstrNumber(@1.first_line); $$ = ↩
            new yy.Ast('BLOCK-INSTRUCTIONS', [$1]);}
    | '{' block_instr '}'
        {$$ = $2;}
    ;

direct_assign
    : id '=' expr
        {$$ = new yy.Ast('=', [$1, $3]);}
    ;

declaration
    : type declaration_body
        {$$ = new yy.Ast('DECLARATION', [$1, $2]);}
    ;

declaration_body
    : declaration_body ',' direct_assign
        {$$.push($3);}
    | declaration_body ',' id
        {$$.push($3);}
    | direct_assign
        {$$ = [$1];}
    | id
        {$$ = [$1];}
    ;


type
    : INT
        { $$ = 'INT' }
    | DOUBLE
        { $$ = 'DOUBLE' }
    | CHAR
```

```
        { $$ = 'CHAR' }
    | BOOL
        { $$ = 'BOOL' }
    | STRING
        { $$ = 'STRING' }
    | VOID
        { $$ = 'VOID' }
    ;

expr
    : expr '+' expr
        {$$ = new yy.Ast('+', [$1,$3]);}
    | expr '-' expr
        {$$ = new yy.Ast('-', [$1,$3]);}
    | expr '*' expr
        {$$ = new yy.Ast('*', [$1,$3]);}
    | expr '/' expr
        {$$ = new yy.Ast('/', [$1,$3]);}
    | expr '%' expr
        {$$ = new yy.Ast('%', [$1,$3]);}
    | expr '&&' expr
        {$$ = new yy.Ast('&&', [$1,$3]);}
    | expr '||' expr
        {$$ = new yy.Ast('||', [$1,$3]);}
    | '-' expr %prec 'u-'
        {$$ = new yy.Ast('u-', [$2]);}
    | '+' expr %prec 'u+'
        {$$ = new yy.Ast('u+', [$2]);}
    | '!' expr
        {$$ = new yy.Ast('!', [$2]);}
    | expr '<' expr
        {$$ = new yy.Ast('<', [$1,$3]);}
    | expr '>' expr
        {$$ = new yy.Ast('>', [$1,$3]);}
    | expr '<=' expr
        {$$ = new yy.Ast('<=', [$1,$3]);}
    | expr '>=' expr
        {$$ = new yy.Ast('>=', [$1,$3]);}
    | expr '==' expr
        {$$ = new yy.Ast('==', [$1,$3]);}
    | expr '!=' expr
        {$$ = new yy.Ast('!=', [$1,$3]);}
    | DOUBLE_LIT
        {$$ = new yy.Ast('DOUBLE_LIT', [$1], leaf=true);}
    | INT_LIT
        {$$ = new yy.Ast('INT_LIT', [$1], leaf=true);}
    | CHAR_LIT
        {$$ = new yy.Ast('CHAR_LIT', [$1], leaf=true)}
    | BOOL_LIT
        {$$ = new yy.Ast('BOOL_LIT', [$1], leaf=true);}
    | STRING_LIT
        {$$ = new yy.Ast('STRING_LIT', [$1], leaf=true);}
    | direct_assign
    | '++' id %prec '++a'
        {$$ = new yy.Ast('=', [$2, new yy.Ast('+', [yy.Ast.copyOf($2), new yy.Ast('INT_LIT', ['1'], leaf=true)])]);}
    | '--' id %prec '--a'
        {$$ = new yy.Ast('=', [$2, new yy.Ast('-', [yy.Ast.copyOf($2), new yy.Ast('INT_LIT', ['1'], leaf=true)])]);}
    | id '++' %prec 'a++'
        {$$ = new yy.Ast('a++', [$1]);}
    | id '--' %prec 'a--'
        {$$ = new yy.Ast('a--', [$1]);}
    | id '+=' expr
        {$$ = new yy.Ast('=', [$1, new yy.Ast('+', [yy.Ast.copyOf($1), $3])]);}
    | id '-=' expr
        {$$ = new yy.Ast('=', [$1, new yy.Ast('-', [yy.Ast.copyOf($1), $3])]);}
    | id '*=' expr
        {$$ = new yy.Ast('=', [$1, new yy.Ast('*', [yy.Ast.copyOf($1), $3])]);}
    | id '/=' expr
        {$$ = new yy.Ast('=', [$1, new yy.Ast('/', [yy.Ast.copyOf($1), $3])]);}
    | id '%=' expr
        {$$ = new yy.Ast('=', [$1, new yy.Ast('%', [yy.Ast.copyOf($1), $3])]);}
    | id
    | cin
    | funcall
    | '(' expr ')'
        {$$ = $2}
    ;

id
    : ID
        {$$ = new yy.Ast('ID', [$1], leaf=true);}
    ;
```

Appendix B

Evaluation programs

Note that the C−− code does not have any time measures, the reason being the time has been measured by editing the compiler's code.

B.1 Fibonacci

B.1.1 C– 1.0 and 1.1

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

int main() {
  for (int N = 10; N <= 50; N+=5) {
    fib(N);
  }
}
```

B.1.2 C++ 11

```cpp
#include <iostream>
#include <chrono>

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

int main() {
  for (int N = 10; N <= 50; N+=5) {
    auto begin = std::chrono::steady_clock::now();
    fib(N);
    auto end = std::chrono::steady_clock::now();
    std::cout << "Time difference in seconds for N = " << N << " " <<
      (std::chrono::duration_cast<std::chrono::microseconds>(end - ↩
          begin).count()) /1000000.0 << std::endl;
  }
}
```

B.1.3 JavaScript

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

for(var N = 10; N <= 50; N+=5) {
  console.time('Time difference in seconds for N = ' + N + ' ');
  fib(N);
  console.timeEnd('Time difference in seconds for N = ' + N + ' ');
}
```

B.1.4 Python 3

```python
import time

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

for N in range(10,55,5):
    start = time.clock()
    fib(N)
    end = time.clock()
    print('Time difference in seconds for N = ' + str(N) + ' ' + str(↩
       end-start))
```

B.2 Is Prime

B.2.1 C– 1.0 and 1.1

```c
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() {
  for (int limit = 1; limit < 16; ++limit) {
    for (int N = 1; N <= limit*1000000; ++N) {
      isPrime(N);
    }
  }
}
```

B.2.2 C++ 11

```
#include <iostream>
#include <chrono>

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() {
  for (int limit = 1; limit < 16; ++limit) {
    auto begin = std::chrono::steady_clock::now();
    for (int N = 1; N <= limit*1000000; ++N) {
      isPrime(N);
    }
    auto end = std::chrono::steady_clock::now();
    std::cout << "Time difference in seconds for N = " << limit << " " ↩
        <<
      (std::chrono::duration_cast<std::chrono::microseconds>(end - ↩
          begin).count()) /1000000.0 << std::endl;
  }
}
```

B.2.3 JavaScript

```
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;
}

for (var limit = 1; limit < 16; ++limit) {
  console.time('Time difference in seconds for N = ' + limit + ' ');
  for (var N = 1; N <= limit*1000000; ++N) {
    isPrime(N);
  }
  console.timeEnd('Time difference in seconds for N = ' + limit + ' ');
}
```

B.2.4 Python 3

```
import time

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

for limit in range(1,16):
    start = time.clock()
    for N in range(1,limit*1000000+1):
        isPrime(N)
    end = time.clock()
    print('Time difference in seconds for N = ' + str(N) + ' ' + str(↩
        end-start))
```

B.3 Collatz Sequence

B.3.1 C– 1.0 and 1.1

```cpp
#include <iostream>
#include <chrono>
#include <math.h>

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() {
  for (int N = 10000; N <= 50000; N+=5000) {
      sequences_collatz(N);
  }
}
```

B.3.2 C++ 11

```cpp
#include <iostream>
#include <chrono>
#include <math.h>

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() {
   for (int N = 10000; N <= 50000; N+=5000) {
    auto begin = std::chrono::steady_clock::now();
    sequences_collatz(N);
    auto end = std::chrono::steady_clock::now();
    std::cout << "Time difference in ms for N = " << N << " " <<
      (std::chrono::duration_cast<std::chrono::microseconds>(end - ↩
          begin).count()) /1000.0 << std::endl;
  }
}
```

B.3.3 JavaScript

```
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);
  }
}

```

```
for(let N = 10000; N <= 50000; N+=5000) {
  console.time('Time difference in ms for N = ' + N + ' ');
  sequences_collatz(N);
  console.timeEnd('Time difference in ms for N = ' + N + ' ');
}
```

B.3.4 Python 3

```
import time

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)


for N in range(10000,55000,5000):
    start = time.clock()
    sequences_collatz(N)
    end = time.clock()
    print('Time difference in ms for N = ' + str(N) + ' ' + str((end-↩
        start)*1000))
```