Universidade do Minho
Escola de Engenharia
Departamento de Informática

Sofia Guilherme Rodrigues dos Santos

ACE Grader
Automatic Grading of Programming Exercises

August 2023

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Sofia Guilherme Rodrigues dos Santos

ACE Grader
Automatic Grading of Programming Exercises

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Prof. Pedro Rangel Henriques
Prof. Alda Gancarski

August 2023

STATEMENT OF INTEGRITY

Sofia Guilherme Rodrigues dos Santos

AGRADECIMENTOS

Primeiro de tudo, preciso de agradecer ao meu orientador, o professor Pedro Rangel Henriques, pela oportunidade de desenvolver este projeto e pelo apoio que me deu ao longo de todo este processo. Não poderia ter escolhido um melhor orientador. De igual modo, agradeço à professora Alda Gancarski pelo seu acompanhamento e mentoria. Quero agradecer também à minha família, especialmente aos meus pais, à minha irmã e à minha Titi por todo o seu suporte e por me terem aturado sempre que precisava de desabafar sobre a dissertação. Por último, obrigada ao Clube Rainbow da Universidade do Minho por me terem acolhido e por me terem introduzido a pessoas incríveis, a quem tenho a sorte de poder chamar amigos.

ABSTRACT

Despite their rising usage in classrooms, most automatic grading tools for programming exercises are quite simple, using only output comparison or unit tests to evaluate a solution, in contrast with manual grading methods used by teachers, which also look at the code itself, even if it doesn't produce a correct solution. Static analysis methods for code have been around for a while, but largely ignored in assessment software.

The Master's project here reported proposes an automatic grading method for programming exercises that, in addition to dynamic analysis, uses static analysis to evaluate submissions. This method benefits both teachers and students, since, by scoring solutions that produce the wrong output, it provides a more comprehensive evaluation of student-submitted programs while also making it easier to see exactly what needs to be improved. Moreover, it makes evaluation more rigorous, by requiring more than just a program that solely produces the correct result. A prototype application called ACE Grader was created to demonstrate the efficacy of this grading strategy.

This dissertation describes a bibliographic review of existing automatic grading tools, proposes and introduces ACE Grader through an overview of its architecture and its development process. As an initial version of the application was deployed in the middle of the second semester of university classes, experiments with students in a real classroom setting are also presented and discussed.

Keywords:
Automatic assessment
Assessment software
Programming exercises
Dynamic analysis
Static analysis
Education technology

RESUMO

Apesar do seu uso crescente em salas de aula, a maioria das ferramentas de avaliação automática de exercícios de programação são relativamente simples, usando apenas comparação de *output* ou testes unitários para avaliar uma solução, em contraste com métodos de avaliação manual usados por professores, que também têm em conta o código em si, mesmo que este não produza uma solução correta. Métodos de análise estática para código, que tentam preencher esta lacuna, já existem há algum tempo, mas são na sua maioria ignorados em software de correção.

O projeto de mestrado aqui descrito propõe um método de avaliação automática de exercícios de programação que, para além de análise dinâmica, usa análise estática para avaliar submissões. Este método beneficia tanto professores como estudantes, visto que, ao pontuar soluções que produzem o *output* errado, avalia de forma mais compreensiva os programas submetidos por estudantes e permite saber exatamente o que pode ser melhorado. Para além disso, torna a avaliação mais rigorosa, ao exigir mais do que um programa que apenas produz o resultado correto. Um protótipo de uma aplicação chamado ACE Grader foi desenvolvido para demonstrar a eficácia desta estratégia de correção.

Esta dissertação descreve uma revisão bibliográfica de ferramentas de avaliação automática existentes, propõe e introduz o ACE Grader através de uma visão geral da sua arquitetura e do seu processo de desenvolvimento. Uma versão inicial da aplicação foi disponibilizada para testes a meio do segundo semestre letivo, o que permitiu que esta fosse usada em sala de aula com estudantes, em experiências que são aqui apresentadas e discutidas.

Palavras-chave:
Avaliação automática
Programa de avaliação
Exercícios de programação
Análise dinâmica
Análise estática
Tecnologia educacional

CONTENTS

LIST OF FIGURES

ACRONYMS

A

Abstract Syntax Tree.

C

Computer Science.

D

Domain Specific Language.

M

Model View Controller.

U

User Interface.

V

Virtual Machine.

1 INTRODUCTION

This chapter describes the motivations behind this Master's project, its objectives and an overview of the research hypothesis and method.

1.1 MOTIVATION

Throughout the last couple of years, teachers and educators have started to rely more and more on software capable of aiding them in grading programming exercises, especially during the confinement periods brought about due to the COVID-19 pandemic.

Automatic grading tools for programming exercises have been around for years, most notably in programming contests, where they can be used by judges to aid in the evaluation process. Such contests include IOI [1] and SWERC [2]. Their software provides participants with an overview of their submissions and if they are valid and/or correct. Mooshak [3] and CMS [4] are two examples of grading tools used in programming contests.

However, most of the existing solutions don't offer a detailed analysis of the evaluation and fail to evaluate incomplete solutions, for example, since their primary objective is to be used in contests.

Despite the existence of some tools designed for classroom usage, like Codeboard [5], they're mostly based on dynamic analysis. In other words, they analyze and validate the submitted piece of code or program by running a series of tests and checking if the program's output matches the expected values [6].

This approach, although very useful, has its limitations, since a direct comparison between expected and real values could fail due to something as simple as a difference in spacing or items ordered differently if the value is a list or another complex data structure. These problems can be alleviated by using regular expressions or more complex comparisons, but, for some exercises, it's impossible to cover every possible output. Dynamic analysis also fails to evaluate a program that doesn't compile, for example, seeing as it doesn't generate a result.

On the other hand, static analysis relies on a program's correctness, i.e., the way it is written, the functions or variables it uses, among other factors that can be measured without running the program itself [6].

Static analysis can be used to check if a submission obeys specific parameters [7]. For example, if an exercise asks students to use a recursive function in order to solve it, without statically analyzing student submissions, it would be impossible to verify this parameter.

Despite its usefulness, static analysis is still relatively rare in automatic assessment tools, especially in tools designed for classroom usage.

1.2 OBJECTIVES

The main objective of this Master's Thesis is to demonstrate that an automatic grading tool that uses a combination of static and dynamic analysis methods to grade an exercise could be a preferable solution for teachers to use in their classrooms, as well as for students to study and learn. For this purpose, a prototype application is to be developed.

This application can be used by teachers to create programming exercises and share them with their students. Teachers can define custom tests and parameters for the tool to evaluate submissions, including partial grading. For example, if a problem consists of recursively removing negative elements from a list of numbers and a student does not define a recursive function, they would be penalized. On the other hand, if a student's solution removes every negative number, but also removes the number 0 from a list, it could still have a partial grade. Most dynamic assessment tools are unable to do either of these things.

Students are able to submit answers to problems shared with them and to see detailed feedback for each submission. In addition to student feedback, teachers can view general statistics for their classes. Finally, students are able to search for exercises to solve on their own, instead of only being able to solve the exercises proposed by their teachers.

1.3 RESEARCH HYPOTHESIS

The main hypothesis made in this Master's project is that an assessment application that relies on both dynamic and static analysis is a preferable tool for grading programming exercises than one without these features, and a better tool for students to learn and practice their programming skills.

1.4 RESEARCH METHOD

The development of this Master's Thesis follows a methodology described in the following paragraphs.

First, a bibliographic search is conducted, where existing assessment methods are identified, as well as current assessment tools.

Then, the results from this search are read, synthesized and compared. The main goal of this step is to assess the current state of assessment methods and tools.

After this step, development of a prototype tool commences. Here, the goal is to prove or disprove the research hypothesis through this application. The research results are used to identify which features this tool shall possess, as well as the main points of improvement over existing solutions.

After the development and testing of the tool is complete, the final research step consists of evaluating the prototype and discussing the obtained results.

1.5 DOCUMENT STRUCTURE

This document is divided into logical sections, each corresponding to one research step.

After this introductory chapter, the State of the Art chapter summarizes the research made on this topic. A general state of current tools is presented, along with a more in-depth description of assessment methods.

The Proposed Approach chapter describes in more detail the proposal for the development of an automatic assessment tool and goes over its features. It also presents some diagrams that reflect the architecture of the application, along with a practical example of the tool's main features.

The Development chapter describes the development process for the application, going over its main features and some of the decisions that had to be made during this process.

In the Case Studies chapter, some illustrative scenarios are used to expose and describe the main features of the application. It also explains how to use the automatic assessment tool.

Finally, the Conclusion chapter summarizes this document and concludes if the research hypothesis has been proven or not, in addition to discussing potential future work.

2

STATE OF THE ART

This chapter provides an in-depth examination on the topic of automatic assessment tools. The first two sections cover assessment methods, while the third section covers existing assessment tools, with a subsection dedicated to comparing some of these applications.

2.1 PROGRAM ANALYSIS

As mentioned in the introduction, there are two main techniques to evaluate a program's correctness [6] [8].

Dynamic analysis is the most common technique among automatic assessment tools, since it's the easiest to implement and arguably the most important one [6] [9]. Dynamic analysis relies on checking if the program is executing correctly, in other words, if, given some input, the program produces the expected output.

This can be verified through simple output comparison, where a small program or script compares the expected values with the actual values returned by the program. However, problems start to arise when the output is more than a simple number or string. If a program outputs a list of values, a difference in the order of the elements or just in spacing could lead to a correct solution being marked as wrong. Some solutions have been found and implemented for this problem, including the usage of regular expressions for comparison, instead of simple text strings [10]. Despite not being perfect, regular expressions allow for a certain degree of freedom in the format of the output, at the risk of inaccurate assessment, if the regular expression were to be too general and classify wrong output as correct [10].

Another common method of dynamic analysis is through unit tests. Unlike output comparison tests, which validate an entire program, unit tests, as their name implies, validate units. These can be functions, methods, classes or any other piece of code that can be logically isolated in a system [11]. Since these tests need to be able to access and interact with a program's logical units, they are usually coded in the same programming language as the units they are testing. For example, unit tests for programs written in Haskell tend to rely on Haskell frameworks, like HUnit [12].

While unit tests allow for more granularity of evaluation, they may be harder to implement than simple output comparison tests [9], especially on automatic assessment tools, where each exercise might require a different set of unit tests, which need to be manually programmed and verified.

On the other hand, **static analysis** does not need to execute a program in order to evaluate it. The common approach in static analysis is to build an abstract syntax tree (AST) from the original program [9]. An AST is nothing more than a graph representation of source code [13], which can be used by automatic tools to find syntax errors and bad patterns in the code without actually executing it [14]. ASTs can also be used to detect plagiarism through pattern comparison, instead of directly comparing source code [9]. Control flow graphs and data flow graphs are also used often to find unused variables or unreachable code, among other issues [9] [15].

2.2 PARTIAL GRADING

Despite being something commonly done by teachers when manually grading exercises, **partial grading** is a feature that is mostly absent in automatic assessment tools [16]. In other words, when a student fails to correctly answer a question, but part of the solution is valid, the solution is given partial credit by whoever is correcting the test. However, when the test is corrected by a computer, this is not usually the case, since most tools only consider answers to be completely correct or completely incorrect. There has been some discussion on implementing partial grading in automatic assessment tools [16], including in the field of Computer Science [17] [18], but it's mostly conceptual. After some research, I was not able to find any publicly available program capable of partially assessing programming exercises.

2.3 ANALYSIS OF EXISTING TOOLS

A recent article published by ACM Transactions on Computing Education by Paiva et al. [9] offers a very detailed state-of-the-art review on the current state of automatic assessment tools in Computer Science (CS) education. This review focuses on five main aspects: the domains in CS that can have their skills automatically assessed, the testing techniques, the approaches to executing untrusted code, the feedback provided to students and the collected analytics.

The article highlights 30 tools and compares them on the basis of the five aspects mentioned above. Since the final application to be developed in this master's project will only focus on traditional programming exercises, the first aspect of the review, which covers alternative domains and automatic assessment tools and techniques within those domains, is not relevant for this state-of-the-art review and will therefore not be mentioned any further.

In regards to testing techniques, all of the aforementioned tools use either output comparison, unit tests, or a combination of both to test a program's functionality. Static assessment is not as common, with many of these tools not supporting it at all or relying on external tools and/or custom scripts to evaluate parameters like code quality or plagiarism. Partial grading appears to have none or limited support in these tools.

When executing a submitted program, a sizable portion of the analyzed tools execute it as a different system user, one without most permissions. This prevents an untrusted program from deleting system files or causing other problems on the machine where the program is running. Some tools rely on Virtual Machines (VMs) and a growing number of them are starting to use containers to execute programs in an isolated environment. In comparison with VMs, containers are faster, smaller and easier to use, since they don't create a copy of the operating system [[19]].

As for student feedback, almost every tool reports to the student which tests they have failed or why their program did not compile/run. A few also provide help on how to proceed in case of errors and a structured report of the evaluation, including metrics like memory usage or execution time.

Lastly, for data collection and analytics, every tool keeps a submission log, as would be expected, but some also store keystrokes and code snapshots, i.e., they save a program's state in fixed time intervals, even if a student did not submit any solution. Educators are then able to consult these logs, among other information. Some tools are capable of displaying this information in charts and tables, for a more efficient analysis.

The study concludes that, although there have been numerous advancements in static analysis of programming exercises, very few tools actually provide this kind of analysis, and it's still only mostly found in prototypes [9]. Therefore, this is one of the biggest areas where many improvements can be made. Other aspects, like containerization for untrusted code execution, are also quickly rising in popularity and may become widely adopted in the future.

Despite being extremely thorough, one topic not covered by the aforementioned study is tool installation. In other words, if a tool needs to be installed by a teacher in a server and how hard that process is. Many educators may not have the required knowledge in computer networking or the time required to set up a server for an assessment tool. Moreover, setting up a server requires either already having the needed hardware or paying for a hosting service, which introduces a financial cost to an already complicated process [20]. Because of this, most educators would likely be inclined to only use tools provided and hosted by their school or university, or tools which are self-hosted.

In addition, assessment tools meant for use in a classroom must allow educators to create their own problem and to see their students' submissions. Some of the tools analyzed in the study, like Kattis [21], do not possess these features. This limitation makes some of these tools harder or even impossible to be used by teachers in their classes.

2.3.1 *Codeboard*

Codeboard [5] is an assessment tool used primarily in classrooms, where students can submit solutions to exercises created and shared by teachers. It supports the most common languages used in programming courses, like C, C++, Java, Haskell [22] and, more recently, Python [23]. It also supports automatic assessment of exercises. However, the files used for the assessment have to be manually written by the teacher. This means that, although the evaluation can be very thorough and versatile, since it can check for any kind of condition, it can also be a very time-consuming task. This assessment can also include static analysis, in addition to simple output comparison, but, as mentioned before, the code used for this analysis would have to be manually written.

All of this makes Codeboard a very powerful tool, but the amount of time required to set up a problem with automatic assessment can be a drawback to many teachers, which already have a busy schedule. Another hindrance is the tool's not very intuitive visual interface, which makes working with it slightly harder and reduces students' motivation [24].

In order to measure more accurately Codeboard's strengths and weaknesses, I conducted a small study with two classes of students in Universidade do Minho, who were being taught Functional Programming by me. Some exercises were created in Codeboard and shared with the students. One of those exercises was solved during class, while the others were to be solved by the students at home. This allowed us to have two different perspectives on this tool, from a teacher's and from a student's point of view.

Every student who provided feedback classified the experience of using Codeboard as positive. Many said it was a great tool to use when studying, either at home or in the classroom, because it allowed them to easily test their programs. Some also expressed their wish for tools like Codeboard to be more widely used in classes.

However, some students complained about the User Interface (UI), saying that it can be confusing when first using Codeboard. A sizable portion of students also pointed out that Codeboard could have a better error feedback system, similar to those of text editors like Visual Studio Code or IDEs, where the errors in a program are usually underlined. In Codeboard, only compilation errors are visible, and those only show up on a small window at the bottom of the screen after compiling the program. This makes errors easy to miss and hard to diagnose and fix. Finally, a few students suggested that Codeboard could be easier to use on mobile devices.

As for the teacher's perspective, starting with the positive aspects, Codeboard makes it very easy to give exercises to students during class and to evaluate their submissions, saving a lot of time and effort in the correction stage. It can also be used for homework, making it very versatile and useful in many scenarios.

The main problems of Codeboard from a teacher's point of view have already been explained above, but this experience made them more evident. Creating an exercise is a very tedious task, since every test needs to be manually defined. In addition, there are two kinds of tests, some that are visible to the students and others that are only visible to the teachers. These tests are defined in two separate files, which means that, if a test should be visible to both students and teachers, it will have to be in both files. This can lead to problems if a test needs to be changed but a teacher forgets to change both versions, in addition to creating unnecessary redundancy.

In addition, when writing the exercise's description, one can only use plain text. In other words, descriptions cannot have bold text, italic text or code blocks, among other features. The absence of code blocks in particular makes the descriptions a lot harder to read by the students, since they can't easily distinguish between code and text.

However, the biggest drawback, the one that makes Codeboard a very hard tool to use by teachers to efficiently evaluate their students is the lack of a class feature. To make this problem more clear, if an educator is teaching two or more classes and wants to consult their students' submissions per class, they cannot do that, Codeboard merges every submission in one single page. The only method to achieve this would be to duplicate every exercise for every class. However, this creates more problems than it solves, since now there are N duplicates of the same exercise, and changing one of them means having to perform N changes. In addition, with this method, a teacher cannot easily compare submissions between classes, since they are in separate pages.

It could also be extremely useful to see global statistics for an exercise, i.e., how many students passed every test, which was the least passed test, which test was passed by every student, etc. These analytics could provide a very detailed overview of the students' performance and be used by teachers to more easily assess their students.

2.3.2 *Exercism*

Although not exactly a tool that can be used by educators, since it lacks classroom management features, Exercism [25] is a very interesting web application with clever ideas. Its main goal is to be used by people who are learning programming languages. It supports a wide variety of languages and provides dozens of exercises for each, in what they call "tracks". In addition to exercises, tracks can also contain a syllabus, which slowly exposes users to new concepts, unlocking new exercises as they advance through the concepts. A lot of existing tools only focus on the exercises themselves, but Exercism tries to interlink them with the concepts needed to understand and solve them.

Another very interesting feature not seen in many automatic grading programs is Exercism's static analysis of code. In addition to using unit tests to dynamically evaluate submissions, Exercism uses ASTs to check for solutions that don't follow the question's parameters or that can be improved, in addition to displaying the compiler's warnings and errors. For example, if programs in a particular language typically have their functions and variables be written only with lowercase letters and underscores (e.g., Python [26]), a user who doesn't follow this naming guideline would receive a warning upon submitting their solution, despite this not being an error or a warning generated by the compiler or interpreter. Additionally, if an exercise requires the usage of a certain function, Exercism's static analyzer can detect if that particular function is being used or not.

This tool's main drawback, as pertaining to this state-of-the-art review, is that it can't be used in a classroom by teachers to create exercises for their students, since all exercises on the platform must be submitted and approved by contributors and are publicly available to anyone. Moreover, teachers aren't able to view nor receive feedback on their students' submissions.

2.3.3 *Tool Comparison*

In order to gain a better understanding on which features are most lacking in automatic assessment tools, a comparison was made between a few tools, some of which have already been mentioned above. Table 1 showcases some features of automatic assessment tools and which of the selected tools possess said features.

Tool	Provides static evaluation of submissions	Allows teachers to manage classes	Allows students to test their solutions before submitting	Students can find and solve problems by themselves
Codeboard [5]	Partially	Partially	Yes	Partially
Kattis [21]	No	No	No	Yes
Exercism [25]	Yes	No	Yes	Yes
Mooshak 2.0 [3]	No	Partially	Partially	No
beecrowd [27]	No	Yes	No	Yes
TestMyCode [28]	No	Yes	No	Partially

Table 1: Comparison between some automatic assessment tools.

All of these tools include support for dynamic analysis, either through unit tests, output comparison, or both. They also support a wide range of programming languages, including C, Python, and Haskell, for example.

Tools with class features allow teachers and educators to create virtual "classes", which students can join and their submissions and progress can be tracked. Tools with this feature can be used in classrooms to assign tasks, homework, or even tests.

If a tool allows students to test their solutions before submitting, it means that students are able to see how many automatic tests their solution passes, which allows them to correct their solutions and only submit once it passes as many tests as possible.

A tool that allows students to find and solve problems by themselves is a tool where exercises can be consulted in a list and searched within it. It also means that students aren't just confined to solving the exercises proposed by their teacher, and can use the tool to study at home.

Some tools, like Blackboard or Mooshak 2.0, only have partial support of some features. This means that they exist in the applications, but aren't as robust as in other applications. For example, in Mooshak, students are informed if their solution compiles successfully before submitting, but not whether it passes any tests. Another example is Codeboard, where submissions can, in theory, be evaluated through static parameters, since Codeboard can use any piece of code to evaluate submissions, but it would require teachers to manually define an algorithm to do so. Moreover, students are able to search for problems to solve by themselves, but this search is limited to going through a list of all the problems in the platform. In other words, there is no way to easily find a problem in a specific programming language or about a specific topic like in other tools, one can only search by name.

2.4 SUMMARY

The research presented in this chapter showed that current solutions for automatic grading software for programming exercises don't offer practical methods for static analysis, and those that do aren't completely suitable to be used in a classroom setting, lacking important features. Moreover, current tools don't seem to support partial grading. Despite this, some available software has many other useful features, like dynamic analysis, secure code execution, student feedback and data analytics for teachers, which can be incorporated into new solutions for automatic assessment.

3 PROPOSED APPROACH

The state-of-the-art review revealed that most current tools are lacking in static analysis. Therefore, the application developed in this master's project has a specific focus on static analysis. It also supports, at least partially, every feature showcased in Table 1 of the previous chapter, which makes it a very useful tool for teaching and learning. In short, it allows teachers to create and manage classes, create exercises that can be automatically assessed, and view detailed feedback on students' submissions. On the other hand, students can join classes, solve any publicly available exercise on the platform, suggested by their teacher or not, and view feedback on their submissions. All of these features make this tool particularly unique, compared to current solutions for automatic assessment software.

Partial grading is another interesting and useful feature that is not present in current assessment tools. However, due to its more limited use case and usefulness, it won't be the main focus of this application. Instead, the primary focus will be static grading.

3.1 SYSTEM ARCHITECTURE

3.1.1 *System Architecture Diagram*

The system follows the architecture proposed in Figure 1.

In this architecture, the main components are the front-end and the back-end.

The front-end contains the GUI of the application. This can be a web server, a mobile application or any other kind of software that reads user input and sends it to the back-end of the application, in addition to showing the back-end's output. It should contain a code editor that allows students to write and submit their solutions, an exercise creator for teachers to create new exercises, an account/class management section, where users can manage their accounts, which classes they're enrolled in, and, in the case of teachers, manage their classes, and an exercise finder, where students can find exercises to solve on their own.

The back-end is the "brain" of the application, responsible for storing, processing and analyzing the data it receives from the front-end. Any persistent data, like account information, is stored in databases after being processed by the application. There are three main

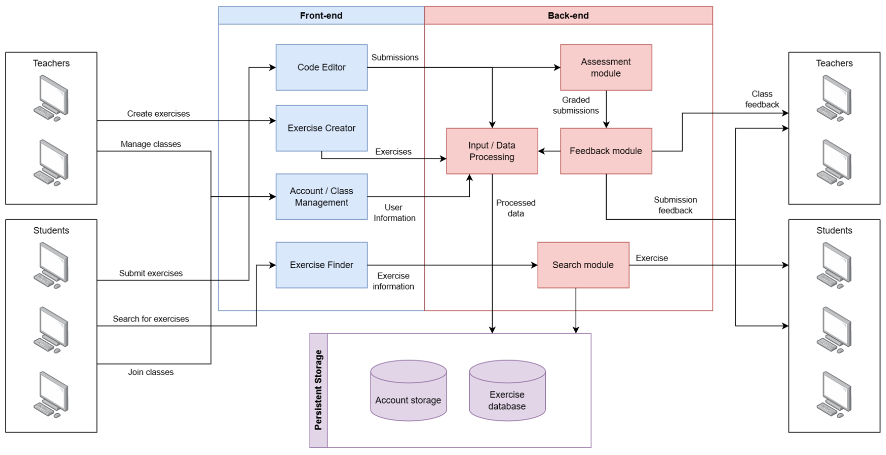


Figure 1: System architecture diagram.

modules in the back-end: the assessment module, which can be split into submodules for each programming language supported by the tool, and is tasked with grading submissions; the feedback module, which analyses the graded submissions and generates statistics, tables or graphs from the grades; and the search module, responsible for querying the exercise database when a student is looking for an exercise to solve. Some of these modules generate output that is shown to the users through the front-end.

3.1.2 *Class Diagram*

The simplified class diagram in Figure 2 illustrates the main structure of the application data, its objects, attributes, and relationships.

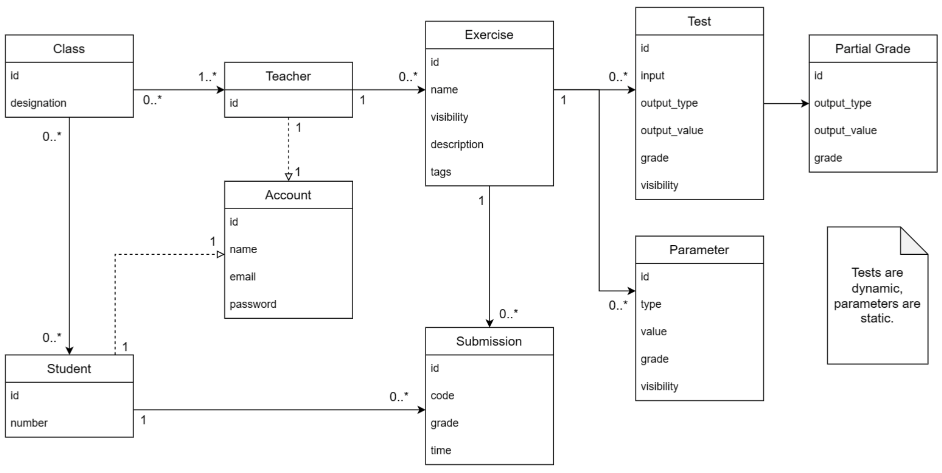


Figure 2: Class diagram.

Both teachers and students have a user account, but depending on the account type, the user may have access to different features of the application. These features are covered in the use case diagram, which is covered in the next Subsection. Teachers can have one or more classes, which students are able to join through an invitation. Teachers can also create exercises, along with tests and parameters for testing said exercises' submissions. Tests evaluate a program's output for a given input. In other words, they perform dynamic analysis. On the other hand, parameters perform static analysis, i.e., they check if the submitted code follows certain rules, without the need to compile and execute said code. This distinction between tests and parameters is explored in further detail in Section 3.2.

Each test or parameter can be set as visible or invisible to students. Visible (or public) tests and parameters allow students to test and correct their solutions before submitting them. Invisible or private tests and parameters are only visible to students after a successful submission. All tests and parameters, regardless of visibility, contribute to a submission's final grade.

Tests have an output type and an output value. The type can be "exact", which means that the solution's output will be directly compared with the test's output value, "regex", which checks if the solution's output matches a regular expression, or "items", which verifies if the solution's output is a container (e.g., list, set, dictionary, etc.) that contains every value specified in the output value, in no specific order.

In addition, tests can also have partial grades. A partial grade is a grade given to a test if it fails to match the expected output with the obtained output, but matches with the partial grade's output. In other words, partial grades allow tests to be given incomplete scores.

Exercises can be set as public or private. A public exercise can be solved by any student, while a private exercise can only be accessed though a URL. This feature allows teachers to create exercises specifically for their classes. An exercise can also optionally be tagged. These tags can refer to an exercise's difficulty or concepts needed to know in order to solve it, for example.

Additional classes or variables may be created in the final application as needed, but this class diagram should cover all the most important components.

3.1.3 *Use Case Diagram*

Finally, the use case diagram in Figure 3 shows the main features of the application and which users have access to those features.

As mentioned above, students can join classes and submit solutions to exercises, and teachers can create exercises, along with tests and parameters for said exercises.

Students can also view feedback on their submissions, i.e., their total grade and the weight of each test/parameter, and additionally search for exercises to solve on their own, in

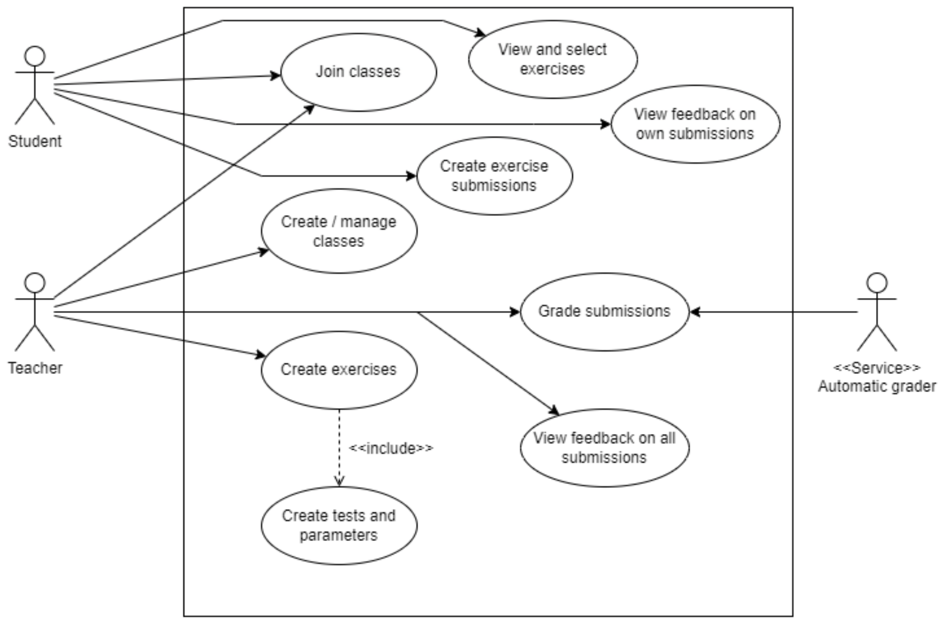

Student
Teacher
<<Service>>
Automatic grader
Join classes
View and select exercises
View feedback on own submissions
Create exercise submissions
Create / manage classes
Grade submissions
Create exercises
<<include>>
View feedback on all submissions
Create tests and parameters

Figure 3: Use case diagram.

addition to solving exercises proposed by their teacher. Teachers can create classes or join a class created by another teacher, through special invitations. In either scenario, they can view the students' submissions and make any corrections to the automatic grading.

3.2 GRADING

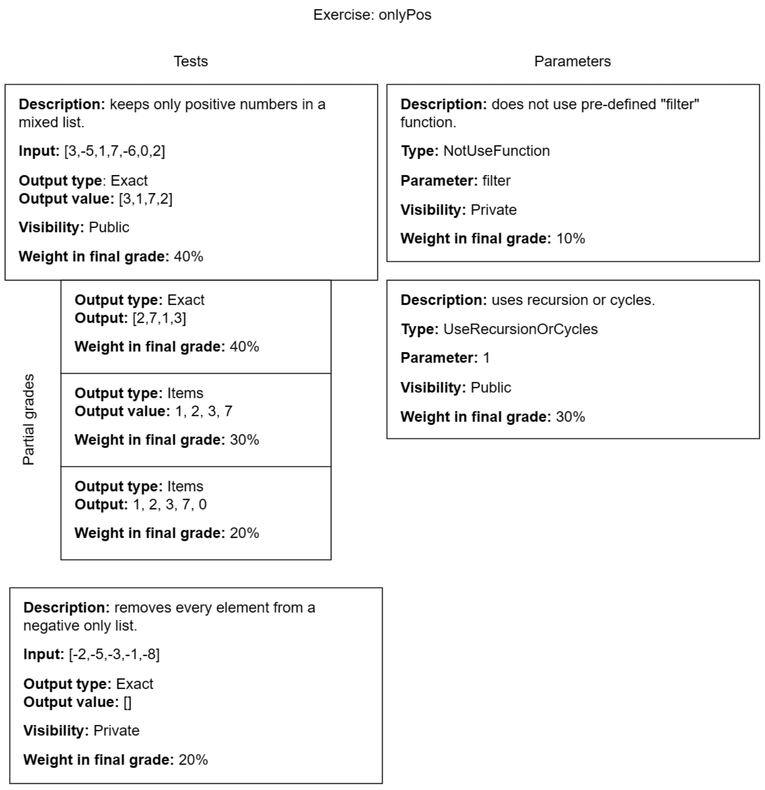
In this application, a submission's grade is calculated through a combination of tests and parameters. The distinction between these two components is that tests are a form of dynamic analysis, while parameters perform static evaluation. In other words, tests verify if the program submitted works properly and produces the expected results while parameters check if the solution obeys the exercise's criteria. These can have different weights, as long as the sum of each individual test or parameter grade adds up to 100%.

For example, if an exercise asks students to define a function that determines if a number is even or not, in most assessment tools, testing would be limited to verifying if the output of the function is correct for a range of even and odd numbers.

However, this evaluation method may not be sufficient in some cases. Returning to the previous example, most programming languages have functions or methods that, given a number, determine if it is even or not. An assessment tool that only performs dynamic analysis is incapable of knowing whether a solution uses one of these functions or not. In this application, through the aforementioned parameters, the system can statically analyze student submissions. If a parameter specifies that solutions cannot use a certain function or import a certain library or module, submissions will have a reduced grade if they fail to pass these parameters, even if they pass every dynamic test. On the other hand, if a submission fails to compile, for example, if it passes one or more parameters, it may still get a positive grade, depending on the weights of each element on the final grade.

This assessment tool converts every submission into an AST, which allows it to easily navigate through the code and validate the parameters that were defined for that particular exercise. Parameters can include: checking if a submitted piece of code uses a particular function or not; checking if a function uses recursion or cycles; checking if a program uses auxiliary functions; checking how many variables a submission uses; etc.

Figure 4 shows a possible representation for tests and parameters inside the application. The exercise in Figure 4 asks students to define a function that filters a list, keeping only positive numbers. In this case, solutions that use recursion or cycles and don't use a "filter" function are worth more points. The value "1" for the "UseRecursionOrCycles" parameter means that solutions may only have at most one recursive function or cycle. A value of "0" would remove this limit.


Exercise: onlyPos

Tests

Description: keeps only positive numbers in a mixed list.
Input: [3,-5,1,7,-6,0,2]
Output type: Exact
Output value: [3,1,7,2]
Visibility: Public
Weight in final grade: 40%

Partial grades

Output type: Exact
Output: [2,7,1,3]
Weight in final grade: 40%

Output type: Items
Output value: 1, 2, 3, 7
Weight in final grade: 30%

Output type: Items
Output: 1, 2, 3, 7, 0
Weight in final grade: 20%

Description: removes every element from a negative only list.
Input: [-2,-5,-3,-1,-8]
Output type: Exact
Output value: []
Visibility: Private
Weight in final grade: 20%

Parameters

Description: does not use pre-defined "filter" function.
Type: NotUseFunction
Parameter: filter
Visibility: Private
Weight in final grade: 10%

Description: uses recursion or cycles.
Type: UseRecursionOrCycles
Parameter: 1
Visibility: Public
Weight in final grade: 30%

Figure 4: Example of tests and parameters for exercise "onlyPos".

As mentioned in the previous section, tests and parameters can be visible to students before submitting a solution or not. If a test or parameter is public, students can use them to test their solution before submitting. For private tests and parameters, these are only revealed after a submission has been made. Either way, after submitting a solution, students can see detailed feedback on their submission, which includes every test and parameter. Teachers are free to choose if they want to have private tests or parameters in their exercises, and how many. This allows the tool to be used for various purposes.

Tests can also be partially graded. In other words, if the expected output for a test does not match the actual output and that test contains partial grades, before moving on to the next test, the tool will verify if the output matches any of the partial grade outputs, in decreasing order from the highest partial grade to the lowest.

Lastly, although not exemplified here, a test or parameter can have a "mandatory" grade. This means that the test or parameter, by itself, does not have an individual grade, but, if failed, zeroes out the total grade. In other words, a mandatory test or parameter needs to be passed for a submission to have a grade above 0%.

3.3 SYSTEM USAGE EXAMPLE

To better understand the architecture and components of the system, an illustrative scenario was created.

In this example, there are five students and two teachers. The students: Ash, Riley, Robin, Charlie, and Sam, belong to professor Parker's class. Subject coordinator Morgan, despite not teaching Parker's class, might still need to consult the students' submissions and statistics, therefore she can also join the class as a teacher. Every student will need an account in order to join a class.

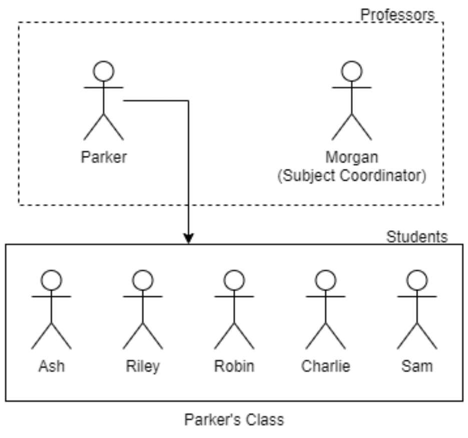
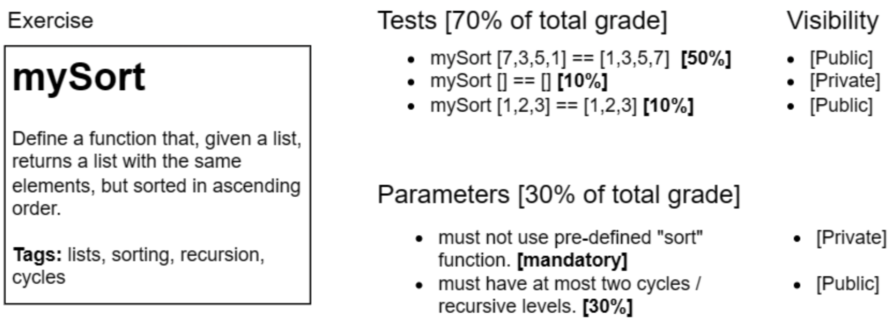


Figure 5: Example - system users.

During a class, professor Parker created an exercise and shared it with their class. This exercise asks students to define a function capable of sorting a list. Since the class is taught in Haskell, students must use that programming language to solve the problem. The exercise contains three tests and two parameters. Figure 6 illustrates this scenario. For simplicity purposes, every test has the "exact" output type, represented in the example as "==".



Exercise

mySort

Define a function that, given a list, returns a list with the same elements, but sorted in ascending order.

Tags: lists, sorting, recursion, cycles

Tests [70% of total grade]

- mySort [7,3,5,1] == [1,3,5,7] **[50%]**
- mySort [] == [] **[10%]**
- mySort [1,2,3] == [1,2,3] **[10%]**

Visibility

- [Public]
- [Private]
- [Public]

Parameters [30% of total grade]

- must not use pre-defined "sort" function. **[mandatory]**
- must have at most two cycles / recursive levels. **[30%]**

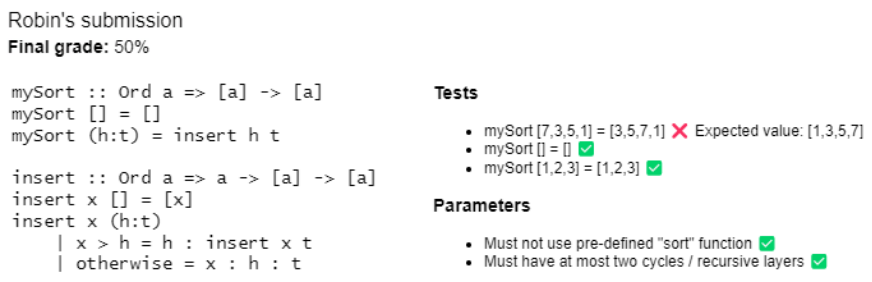
- [Private]
- [Public]

Figure 6: Example - proposed exercise.

The tests verify that the function can sort a list, that it can deal with empty lists, and that it doesn't "unsort" a list that is already sorted. The second test is private, which means that students will have to take into consideration the test case without explicitly being told about it. The other tests are public, therefore the students can execute their solution and check if it passes those tests before submitting. In total, the tests are worth 70% of the total grade, the sum of each test's percentage.

The parameters check if the solution does not use a pre-defined "sort" function (in other words, a function that already exists in the language) and that the solution does not use more than two cycles or levels of recursion. This second parameter, worth 30% of the final grade, makes sure that submissions have a worst-case complexity of $O(N^2)$. The first parameter, unlike the other tests and parameters, is mandatory. This means that every submission needs to pass this parameter in order to be graded. Otherwise, the final grade will always be 0%.

After submitting a solution, each student has access to detailed feedback about their submission. For example, if Robin submitted a solution similar to the one in Figure 7, they would have a final grade of 50% and would be able to see in detail the grade of each test and parameter, as well as, in the case of dynamic tests, the expected output. Sam's solution is only missing the recursive call in the "mySort" function, but, because of that small mistake, the final function is unable to correctly sort a list. Another grading tool might have given Sam a 0% grade, since her solution fails to perform its main task. However, this tool, with the other tests and with the parameters, is able to grade other aspects of Sam's solution.


Robin's submission
Final grade: 50%

```
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort (h:t) = insert h t

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (h:t)
    | x > h = h : insert x t
    | otherwise = x : h : t
```

Tests

- mySort [7,3,5,1] = [3,5,7,1] ✗ Expected value: [1,3,5,7]
- mySort [] = [] ✓
- mySort [1,2,3] = [1,2,3] ✓

Parameters

- Must not use pre-defined "sort" function ✓
- Must have at most two cycles / recursive layers ✓

Figure 7: Example - Robin's submission.

Students can only see their own submissions. On the other hand, both professors would be able to see every student's submission and the respective feedback, as seen, albeit simplified, in Figure 8.

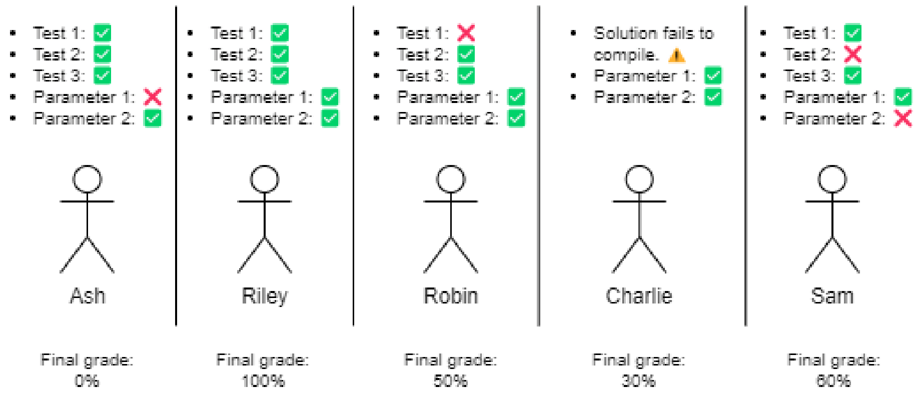
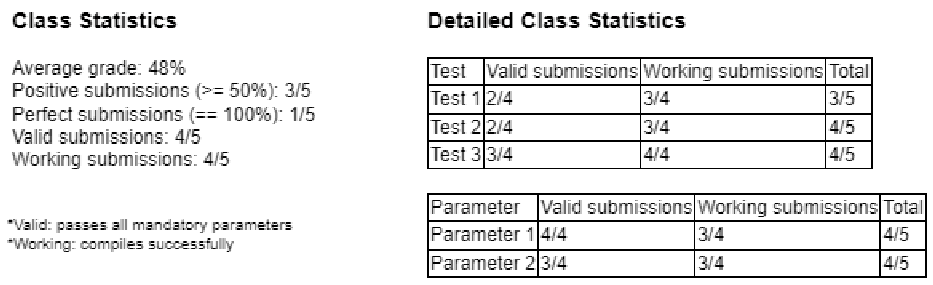


Figure 8: Example - all student submissions.

Here, one can see that Ash got a final grade of 0% because he failed to comply with the first parameter, while Riley got 100%, since she passed every test and parameter. Charlie, despite not having a working solution (i.e. it fails to compile), still wound up with a final grade of 30%, as their solution passes both parameters. Finally, Sam failed the second test which, despite being simple, is not public, so Sam might not have considered the scenario where the function receives an empty list, despite her solution being able to sort non-empty lists. Sam also failed the second parameter, which means that she used an unnecessarily complex algorithm to sort lists. All of these factors contribute to a submission's final grade.

Moreover, the professors would also be able to consult a page with statistics regarding the class' submissions, akin to the tables shown in Figure 9.



Class Statistics

Average grade: 48%
Positive submissions (>= 50%): 3/5
Perfect submissions (== 100%): 1/5
Valid submissions: 4/5
Working submissions: 4/5

*Valid: passes all mandatory parameters
*Working: compiles successfully

Detailed Class Statistics

Test	Valid submissions	Working submissions	Total
Test 1	2/4	3/4	3/5
Test 2	2/4	3/4	4/5
Test 3	3/4	4/4	4/5

Parameter	Valid submissions	Working submissions	Total
Parameter 1	4/4	3/4	4/5
Parameter 2	3/4	3/4	4/5

Figure 9: Example - class submission statistics.

The "valid submissions" and "working submissions" columns are just examples of possibly useful filters that teachers may want to apply to the global statistics, in order to more effectively assess the class' progress and average grades on each test/parameter.

4 DEVELOPMENT

This chapter covers the development process of the automatic grading tool.

Before development started, there was a need to name the application. The chosen name was ACE Grader (Automatic Code Evaluator and Grader), since it's short, easy to remember and describes the main focus of the application. From this point forward, the name ACE Grader will be used to refer to the application.

Prior to development, there was also a choice to be made regarding the type of application that ACE Grader was going to be and consequently which tool or tools would be needed for development.

For a tool like this, which needs to be easy to configure and use, the best option for the type of application seemed to be a web application, which only requires a browser to run, making it very accessible to both students and teachers. Based on this decision, I ended up choosing the Phoenix framework for the backend and the frontend of the application. Phoenix is a full-stack web framework for the Elixir programming language, which follows the MVC pattern. I chose Phoenix because, on one hand, I had prior experience with this framework, saving a lot of time that would be required to learn a new framework, and, on the other hand, it's very robust in terms of scalability and concurrency, since it runs on the Erlang VM, BEAM [29] [30]. For a tool that might be used by dozens, perhaps hundreds, of students at the same time, concurrency is something important to consider.

Lastly, for the database engine, I chose PostgreSQL because it's a very robust database engine, and it offers the best compatibility with Phoenix. However, other engines like MySQL or MongoDB could have also been used without affecting the application's performance or security.

The next two sections focus on the backend and the frontend components of ACE Grader, respectively.

4.1 BACKEND

This section highlights the development process and the main features of ACE Grader's backend.

4.1.1 *Creating and Maintaining Exercises*

One of the main features of ACE Grader is the ability to create exercises.

An exercise, in this context, is a problem that requires students to submit a solution in order to solve it. Exercises must have, at least, a title, a description, an author and a programming language that must be used in order to solve it. This version of ACE Grader only focuses on the C programming language, so this value isn't currently used. Exercises also have a "public" flag, that, if set, allows anyone to find and solve that exercise. Otherwise, only users with the exercise's URL are able to access it.

When an exercise is created, it contains a list of tests and/or parameters. The sum of the individual grade for each test/parameter should equal 100. Because of this, every exercise needs to contain at least one test or parameter. However, if someone creates an exercise and doesn't want it to be graded, they can define one test that always evaluates to a true value, for example, checking if the output contains an empty string.

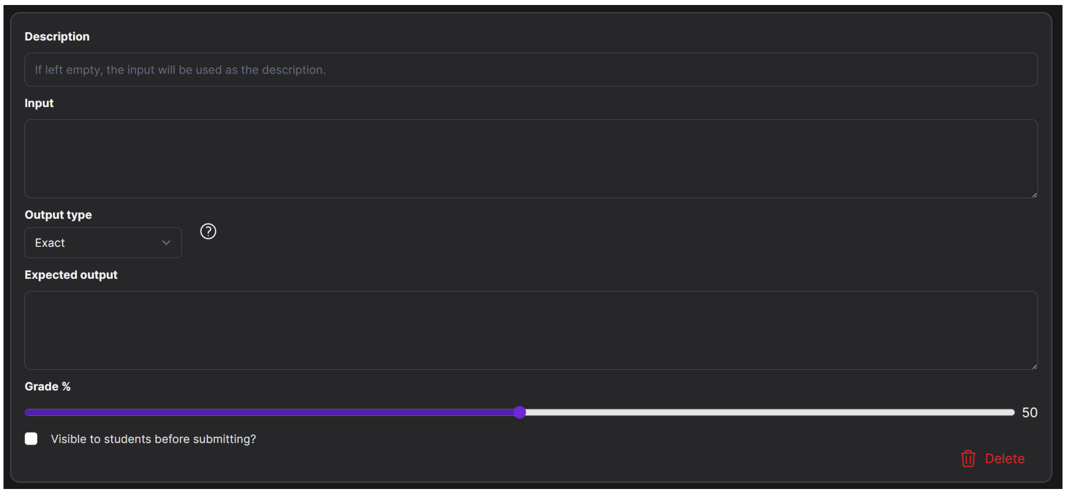
Since some tests might require additional information or setting up, exercise creators can define a "main" file when creating an exercise. This file contains the code that will be executed when testing a solution. If an exercise, for example, asks to define a function that sorts a list, this "main" file could contain a function that creates an unsorted list, sorts it using the submitted function and then prints the new list. This method also allows educators to write exercises where students must define multiple functions and, in this "main" file, combine the functions' outputs into one single string, making the testing process simpler.

Tests

As mentioned in the previous chapters, a test is a dynamic method of checking for a program's correctness. A test contains an input (what the program receives through the standard input), an expected output (what the program should print through the standard output), a grade percentage (how much that test is worth, in percentage of the total grade), and the test type. This type can be one of the following:

- **Exact** - The real output should match exactly with the expected output (excluding leading and trailing whitespace);
- **Regex** - The expected output is treated as a regular expression, and the grader checks if the regular expression returns a positive match for the real output;
- **Items** - The expected output is treated as a list of items, one per line, and the grader checks if the real output contains all of those items.

Each test can be either passed or failed, there are no intermediate states, since these would require partial grading, which the current version of ACE Grader does not support. If a test



Description

If left empty, the input will be used as the description.

Input

Output type

Exact

Expected output

Grade %

50

Visible to students before submitting?

Figure 10: Form for creating a test for an exercise in ACE Grader's frontend.

is passed, that test's grade will be added to the submission's total grade. A test can fail for a variety of reasons. These include:

- **Failed** - The real output did not match the expected output, accordingly with the test type.
- **Error** - The program crashed or exited with an error code.
- **Timeout** - The solution did not finish in a "reasonable" amount of time.

The timeout state will be explained in more detail further below, in the Grader Section.

Parameters

As previously discussed, a parameter serves as a static means of assessing a program's correctness. However, since parameters don't require executing the program, they don't contain input and output values. Instead, they have a key and an optional value.

The key identifies the parameter, and has a numerical value. This allows for us to easily divide parameters into sections and check which section a parameter belongs to without having to perform complex comparisons.

The current parameters supported by ACE Grader, identified by their keys and descriptions, are as follows:

1 - Program uses recursion

2 - Program uses loops

3 - Program uses pointers

4 - Program uses dynamic memory

5 - Program frees allocated memory

10 - Function X is used

11 - Function X is recursive

12 - Function X is iterative

13 - Function X uses pointers

14 - Function X uses dynamic memory

15 - Function X frees allocated memory

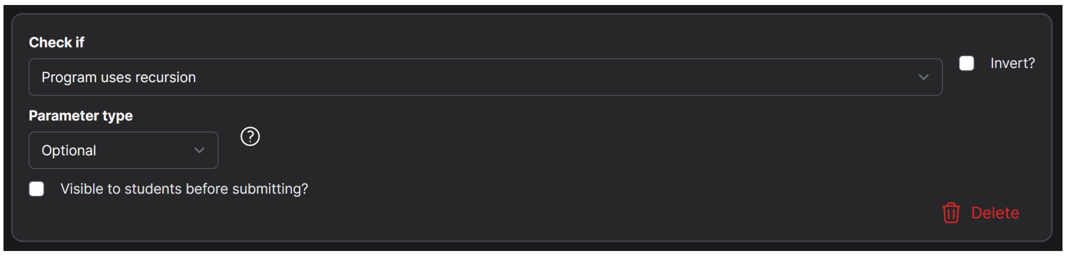
Keys smaller than 10 are related to the program itself, while keys larger than or equal to 10 are related to a specific function. Key values 6 through 9 were left unused purposefully, to facilitate the implementation of possible future parameters. For parameters that specify a function, the name of the function is stored in the aforementioned optional value.

Parameters also have types, but these are different from test types, due to the difference in analysis (dynamic versus static analysis). Thus, a parameter type can be one of the following:

- **Optional** - The parameter will not have an impact on the final grade. Failing a parameter of this type will only result in a warning.
- **Graded** - The parameter contains a grade that is added to the submission's total grade. A parameter of this type is similar to a test.
- **Mandatory** - If the parameter fails, the final grade is always 0, even if every other test or parameter passes.

Each type of parameter is useful in certain situations. For example, mandatory parameters can be used in exercises where students must use a specific concept or function in order to solve the exercise, while optional parameters can be used as warnings, for example, if an unnecessary function is used, or if a student forgets to free allocated memory.

A parameter can also be "negative". In other words, if the "negative" value for a parameter is true, the parameter's result will be flipped. For example, if a "Program uses recursion" parameter is negative, that parameter will pass if and only if the submission does **not** use recursion.


Check if

Program uses recursion

Invert?

Parameter type

Optional

Visible to students before submitting?

Figure 11: Form for creating a parameter for an exercise in ACE Grader's frontend. If the parameter key is > 10, an additional field appears for the optional value.

4.1.2 *Handling submissions*

A submission is very similar to an exercise, in terms of the data they store. They both contain tests, parameters and an author. However, since a submission is only an answer, it doesn't need to store the exercise's title and description. Instead, it stores the ID of the original exercise. Moreover, a submission stores the code written by the user who created the submission, i.e., the solution.

After a student submits a solution, ACE Grader creates a copy of each test and parameter from the original exercise and attaches them to the newly created submission. This prevents previous results from being removed or altered if the original exercise suffers any change. For example, if an exercise contains three tests, a submission is made, and then a fourth test is added to the original exercise, the submission won't contain this new test, since this could negatively impact the final grade. Thus, if an exercise is changed, students must submit a new solution if they want their submission to be graded using the new tests and/or parameters.

After being created, submissions are sent to the grader module, which compiles, executes, and grades the submission.

4.1.3 *Grader*

The grader module is responsible for compiling, executing, and analyzing every submission, before giving it a grade.

Since the compilation and execution tasks may lead to security issues, these steps were relegated to another process. A small web server was developed using Python and Flask, which has the sole responsibility of compiling and running user submissions. This allows us to run this "secondary" server in a Docker container, for example, and if a malicious user tries to exploit or harm the system, by submitting a solution with a *fork bomb* [31], for example, only this container would be affected and it can be easily restarted, leaving the rest of ACE Grader protected.

The main ACE Grader process communicates with this other server, receiving the compilation and test/parameter results.

After every test has been run and all parameters have been verified, the ACE Grader calculates the final grade for the submission and saves everything in the database, which means that this process has to be done only once per submission. A message is also sent to the front-end, so that the submission page can display the final results without the user needing to reload the page.

To prevent the grader from slowing down too much, and to avoid other issues, the secondary server has a timeout of 5 seconds for every operation. This particular value is arbitrary and can be changed if necessary. After this period of time, if the solution still hasn't finished running, or the parameters haven't been verified, the grader forces the program to close, returning a "timeout" error message to ACE Grader.

The main ACE Grader process also contains a timeout for receiving responses from the grader server. By default, this value is 8 seconds, after which a "timeout" value is given to the submission's tests and parameters. In order to prevent submissions from being graded as "timed out" due to a server error, this only occurs if the solution compilation was successful, which only happens if the grader server is online and functioning.

If, for any reason, the grader module is offline or unavailable, submissions will remain in a "pending" state. Then, whenever a student or teacher tries to access a pending submission, ACE Grader will retry the grading process once more. Originally, ACE Grader would just keep trying to grade submissions over and over again in an infinite loop, but since this method is ineffective and could lead to performance issues, it was changed.

Tests

In order for the grader to run a test, first it ensures that the submission has been successfully compiled and that an executable file has been previously generated. Then, it runs the generated program N times, once for each test, with N being the total number of tests. The input, specified in the test data, is given to the program, and after execution has finished the output is captured. This output is then returned to the main ACE Grader process, which is responsible for checking if it matches the expected value, according to the test type.

If one wanted to expand ACE Grader to support more programming languages, the only change needed here would be the inclusion of more compilers. Every other step in this process is language-agnostic.

Parameters

As mentioned in the State of the Art chapter, the most common form of static analysis relies on Abstract Syntax Trees. As such, the grader process starts the parameter evaluation by first converting the submitted solution into an AST. This process is mostly language-specific.

In the case of ACE Grader, since the only currently supported language is C, only ASTs of programs written in C can be generated. Support for other languages could be added, but it wouldn't be as simple as dynamic analysis.

For this conversion process, I decided to use the Clang compiler [32], instead of manual parsing. This saves a lot of time and ensures that the conversion is successful since Clang is widely used and has been thoroughly tested. Clang, among other features, allows us to generate an AST from source code without compiling it [33].

In order to easily make use of Clang's features in the secondary Python server, I used a Clang API available for Python [34]. With this API, generating and traversing an AST becomes substantially easier. Based on the parameters specified in Subsection 4.1.1, some auxiliary functions were defined. These functions traverse the AST and check for a specific parameter. For example, in order to verify that a function is recursive, one of the auxiliary functions receives a function name and a pointer to that function and checks if one of its instructions is or contains a function call to that same function.

4.1.4 *Student classes*

The ability to create and manage classes is a feature that can be used by teachers to keep track of their students' submissions and progress. A class must have a name and a supervisor. By default, the supervisor is the user who created the class. Only teachers can create classes, but they can also join classes created by other teachers. Students can also join more than one class.

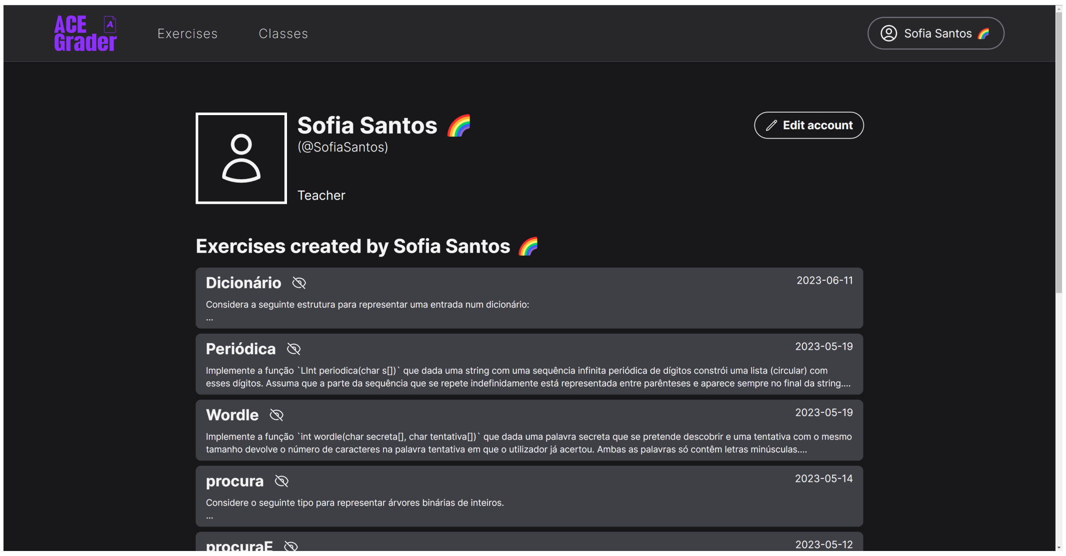
4.2 FRONTEND

This section highlights the development process and the main features of ACE Grader's frontend.

4.2.1 *User Accounts*

In order to perform most operations within ACE Grader, a user must be authenticated. Anyone can create an account in ACE Grader, and there are two kinds of accounts.

- Student accounts are able to browse exercises and submit solutions. They can also view feedback on previous submissions.
- Teacher accounts, in addition to everything that a student account can do, are also able to create new exercises and see every submission for an exercise that they have created.



Sofia Santos
(@SofiaSantos)
Teacher

Exercises created by Sofia Santos

Dicionário 2023-06-11
Considera a seguinte estrutura para representar uma entrada num dicionário:
...

Periódica 2023-05-19
Implemente a função `LInt periodica(char s[])` que dada uma string com uma sequência infinita periódica de dígitos constrói uma lista (circular) com esses dígitos. Assuma que a parte da sequência que se repete indefinidamente está representada entre parênteses e aparece sempre no final da string....

Wordle 2023-05-19
Implemente a função `int wordle(char secreta[], char tentativa[])` que dada uma palavra secreta que se pretende descobrir e uma tentativa com o mesmo tamanho devolve o número de caracteres na palavra tentativa em que o utilizador já acertou. Ambas as palavras só contêm letras minúsculas....

procura 2023-05-14
Considere o seguinte tipo para representar árvores binárias de inteiros.
...

procuraE 2023-05-12

Figure 12: A teacher's personal page in ACE Grader.

This feature wasn't highlighted in the Backend section because Phoenix handles every step of the authentication process, which is a simple email and password authentication. Therefore, the development of this feature was solely focused on the front-end elements.

When creating an account, the user is free to choose between a student account and a teacher account. In addition, they must create a username, which is a unique identifier for their account (for example, a school or university number, or a nickname) and a display name, which helps other users to identify a particular user, if their username is not easily recognizable. For example, a student called Sam with a student ID of A12345 could use "Sam" as their display name and "A12345" as their username.

Every user contains a personal page in ACE Grader. If the user is a teacher, this page will show every exercise created by the teacher. On the other hand, a student's page will show the student's submissions to every public exercise in ACE Grader. However, for privacy reasons, a student cannot access another student's personal page, only their own.

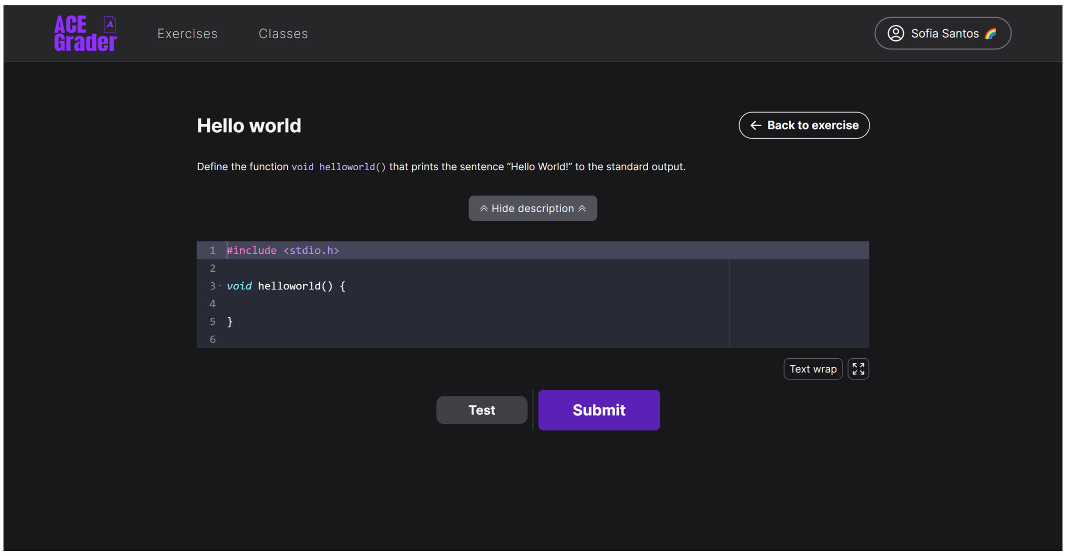
4.2.2 *Exercise Creator*

Teacher accounts have access to the exercise creator feature in ACE Grader. This allows them to create new exercises, which they can publish and make available for anyone to solve, or leave them private and use in their classes as homework or for evaluation purposes, for example.

The next chapter presents this process in more detail.

4.2.3 *Code Editor*

In order to facilitate the process of submitting a solution for an exercise, ACE Grader provides students with a code editor environment for each exercise, where they can write their solutions as if they were locally editing a file. This approach also allows ACE Grader to be used on devices without adequate coding environments, such as smartphones or tablets.



Define the function void helloworld() that prints the sentence "Hello World!" to the standard output.

```
#include <stdio.h>

void helloworld() {

}

```

Figure 13: ACE Grader's online code editor.

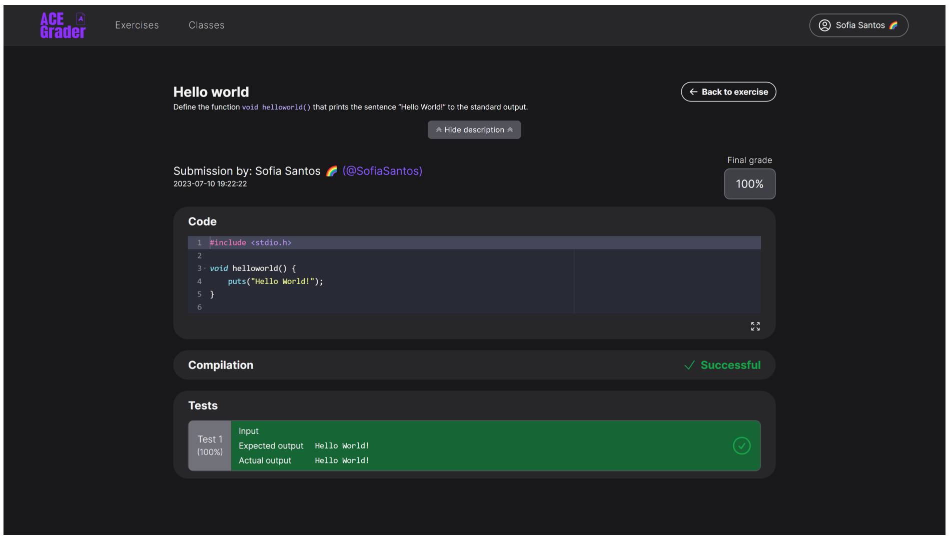
Alongside the code editor, there is a test button, which students can use to verify that their solution compiles and passes some of the exercise's tests and parameters, before officially submitting their work. Not every test or parameter might be visible on this page. When creating an exercise, as mentioned previously, teachers can choose if a test/parameter is visible for students before submission. Only those tests/parameters will appear here. However, their individual grades won't be visible. Only after submitting can students see the individual grade for each test and parameter.

The "Hide description" button makes the exercise description invisible. In exercises with long descriptions, this button allows the user to "clean" the window, making it easier to focus on the code editor itself.

There are two additional buttons in the editor. The "Text wrap" button, as the name implies, wraps the text, i.e., if the length of a line is bigger than the length of the window, the line is automatically broken, to avoid horizontal scrolling. The button with the arrows expands the editor area vertically, to avoid vertical scrolling.

4.2.4 *Submission Review*

After a student submits a solution, they are redirected to a page where they can check the total grade given by ACE Grader to their submission, as well as the individual grade for each test and parameter. This page can be seen in Figure 14.



Hello world

Define the function `void helloworld()` that prints the sentence "Hello World!" to the standard output.

Submission by: Sofia Santos 🌈 (@SofiaSantos)
2023-07-10 19:22:22

Final grade: 100%

Code

```c
#include <stdio.h>

void helloworld() {
    puts("Hello World!");
}

```

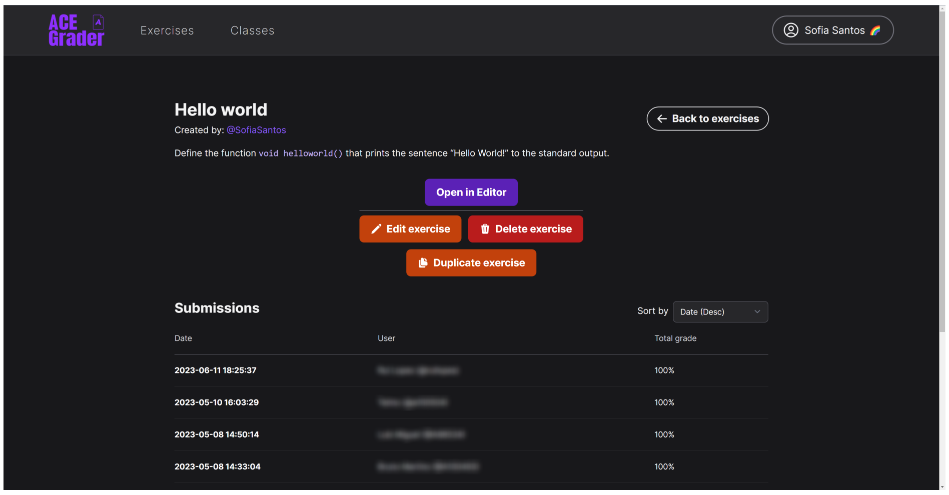
Compilation ✓ Successful

Tests

```
Test 1 (100%)
Input
Expected output   Hello World!
Actual output     Hello World!
```

Figure 14: A submission for a simple Hello World exercise.

When a teacher accesses an exercise's page, they can see every submission made for that exercise, as well as who submitted it and the total grade, in a table format. Figure 15 shows this page. By clicking on a table entry, teachers can access the individual submission page, the one mentioned in the previous paragraph.



Hello world

Created by: @SofiaSantos

Define the function `void helloworld()` that prints the sentence "Hello World!" to the standard output.

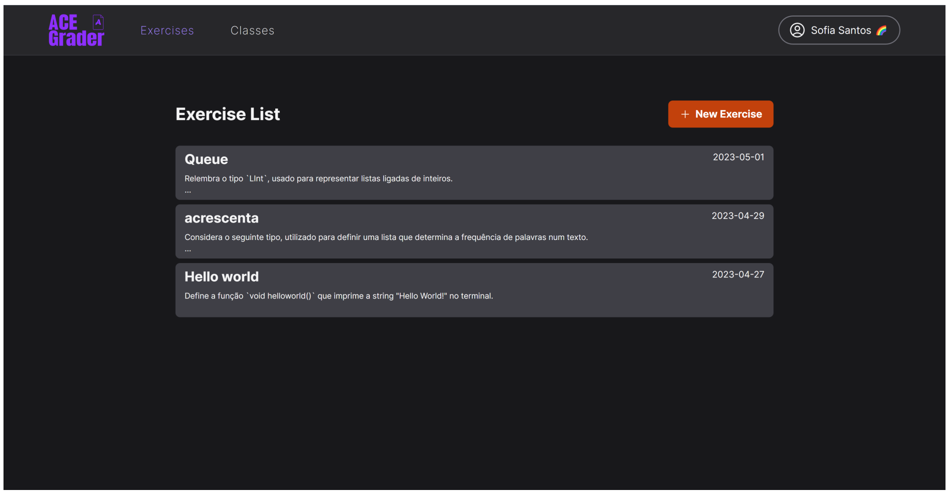
Submissions

Date	User	Total grade
2023-06-11 18:25:37	[redacted]	100%
2023-05-10 16:03:29	[redacted]	100%
2023-05-08 14:50:14	[redacted]	100%
2023-05-08 14:33:04	[redacted]	100%

Figure 15: Exercise page for a simple Hello World exercise, with some student submissions.

4.2.5 *Exercise Finder*

ACE Grader provides students with a page where they can search for and find exercises to solve on their own. Any public exercise created within ACE Grader will appear on this list, allowing students to solve it independently.



Exercise List

Queue 2023-05-01
Relembra o tipo `LInt`, usado para representar listas ligadas de inteiros.
...

acrescenta 2023-04-29
Considera o seguinte tipo, utilizado para definir uma lista que determina a frequência de palavras num texto.
...

Hello world 2023-04-27
Define a função `void helloworld()` que imprime a string "Hello World!" no terminal.

Figure 16: Example of a list of public exercises in ACE Grader. Students can't see the "New Exercise" button.

In addition to the title, this list also shows an exercise's description and creation date, and is sorted by descending creation date, i.e., the most recent exercises will appear first on the list.

4.2.6 *Class Management*

Each class created by a teacher has a page where one can see every student or teacher who joined that class. Teachers also have access to a class management page, similar to the one shown in Figure 17, where they can see every class that they are part of and create a new class. Students can also use this page to see which classes they are enrolled in. Since classes are private, users are only able to join a class if they receive the URL for that class.

4.2.7 *Accessibility*

In order to improve user experience and make the application more accessible to everyone, ACE Grader includes some accessibility settings.

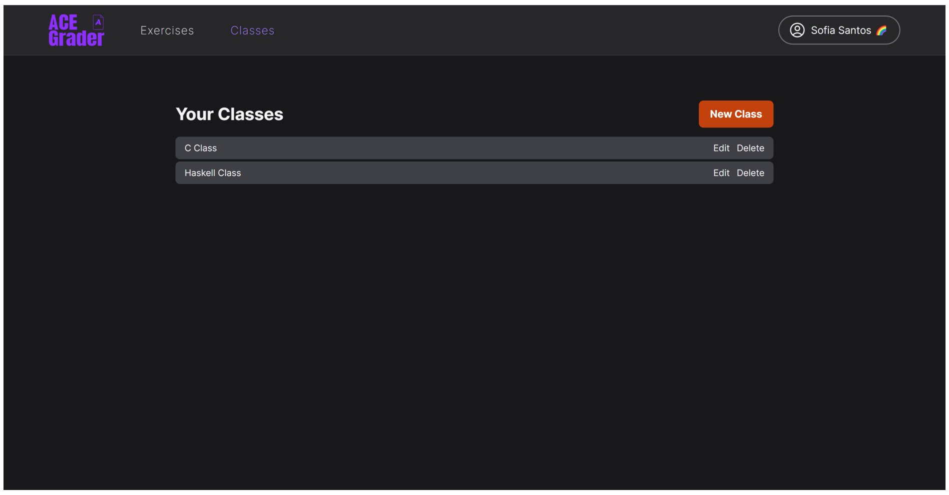


Figure 17: Class management page with two classes, "C Class" and "Haskell Class".

Language

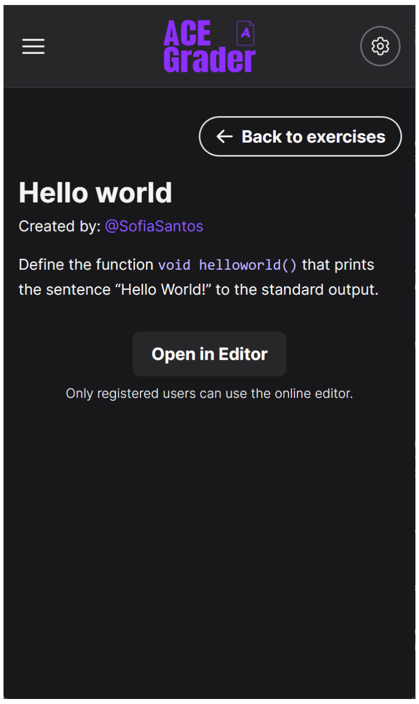
Users can choose between Portuguese or English for ACE Grader's UI. Unfortunately, exercise descriptions remain in the original language they were written in, and it's up to each individual teacher to provide translations for their exercises.

Theme

ACE Grader can detect the system's theme preference (light or dark), but users are also free to choose whichever theme they prefer. A dark theme is useful in dark environments or for users with more sensitive eyes, while a light theme offers more contrast and can be easier to read in bright environments. Every screenshot shown in this chapter uses ACE Grader's dark theme. The next chapter will include screenshots with ACE Grader's light theme.

Responsiveness

Every page in ACE Grader was designed with responsiveness in mind. This means that the application can be used on any kind of device, as long as it possesses a web browser with JavaScript support and an input method, like a keyboard. The content of every page will be scaled according to the screen size. This is especially useful for people with vertical or *ultra-wide* monitors, or people who only have access to a smartphone or a tablet. On mobile devices, writing code may not be as easy as on a computer, due to the way virtual keyboards work, but ACE Grader is still 100% usable on such devices.



Hello world

Created by: @SofiaSantos

Define the function `void helloworld()` that prints the sentence "Hello World!" to the standard output.

Only registered users can use the online editor.

Figure 18: Exercise from Figure 15, but on a small vertical device and without a user account.

4.3 DEPLOYMENT

Thanks to Docker [35], deployment of ACE Grader is extremely simple. Both the main ACE Grader process and the secondary grading module can be run as Docker containers, with minimal set-up required. Another container for the database is used, meaning that, in order to run ACE Grader with all of its features, one needs to create and run three containers in total.

However, if for any reason one does not want to use Docker, every component in ACE Grader can be executed as a normal system application, as long as all needed dependencies are installed.

ACE Grader was deployed to a server in Universidade do Minho, which allowed me to test the tool with other users, namely with students. In addition to helping to find any problems with the application, testing with students was also very useful for finding new ways to improve the application, so this process was very valuable in ACE Grader's development.

5 CASE STUDIES

This chapter showcases ACE Grader's main features and functionalities through some simple examples and demonstrations, with relevant screenshots for visual aid. However, the screenshots have been cropped in order to only show the relevant sections of each page and not the full ACE Grader window.

The account creation process is not included in this Chapter, since it doesn't deviate from how it works on most other websites. In short, when a user clicks on the "Log in" button, if they don't have an account, they can create one by pressing the "Sign up" button and filling out the form that appears with their personal information. The scenarios shown in this Chapter will skip this step and assume that all of the accounts presented have already been created.

5.1 CREATING AN EXERCISE

Since one of ACE Grader's main features is the ability to create exercises, this process has been made simple, yet robust and with many options.

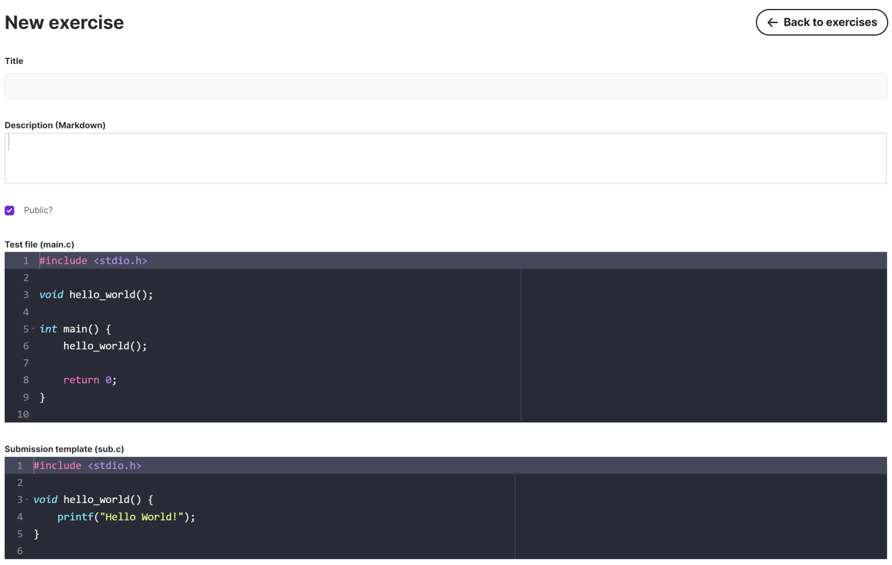
In order to showcase this feature, I will present a concrete example of a programming exercise in C and go through the process of adding that exercise to ACE Grader.

The programming exercise used in this example consists of the following problem definition:

> Define the function `void upcase(char *s)` that converts every lowercase character in a string to **uppercase**. This function must be recursive and should not use the `toupper()` function from the `<ctype.h>` header file.

This exercise is a good example for this demonstration since it requires multiple tests and parameters to be correctly graded.

When adding an exercise to ACE Grader, the first step is to log into a teacher account and go to the "Exercises" page. This page can be easily accessed from ACE Grader's header. From there, one just needs to press the "New Exercise" button. This will lead to the exercise creation form. This form is divided into three parts. The first one, shown in Figure 19, is related to the exercise itself.



Title

Description (Markdown)

[✓] Public?

Test file (main.c)

```c
#include <stdio.h>

void hello_world();

int main() {
    hello_world();

    return 0;
}

```

Submission template (sub.c)

```c
#include <stdio.h>

void hello_world() {
    printf("Hello World!");
}

```

Figure 19: First part of the exercise creation form.

The first field is the exercise title. This field makes an exercise easy to identify, but it can have any value.

The second field is the exercise description. Here, one should place the problem definition, as presented above. Since this field accepts Markdown input, the description can include bold and italic text, code blocks, quotes, and any other element supported by Markdown [36].

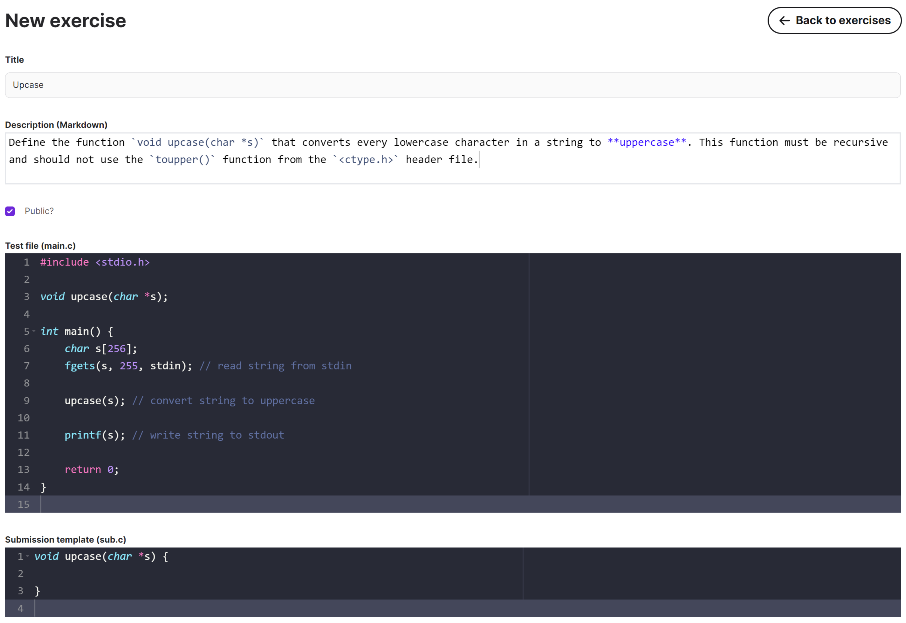
The third field defines whether the exercise is public or private. As mentioned previously, a public exercise can be accessed from ACE Grader's exercises page and is visible to every user, while a private exercise can only be accessed by users who have the URL for the exercise. In this scenario, the exercise can remain public.

The fourth field is the test file, which is used to evaluate submissions. In other words, when a student submits a solution, this file will be compiled and executed, along with the solution file. The program defined in this file should call the function or functions from the solution file in order to execute them. This method, as opposed to directly executing the solution file, allows for more granular testing.

In this case, since the function in question must modify a string, the test file can receive a string from the standard input, call the function with that string and print the new string to the standard output.

Finally, the fifth field is the submission template. This field can be left empty, but, as the name implies, it can be used to define a template for student submissions.

The filled exercise creation form can be seen in Figure 20.



New exercise

Back to exercises

Title

Upcase

Description (Markdown)

Define the function `void upcase(char *s)` that converts every lowercase character in a string to **uppercase**. This function must be recursive and should not use the `toupper()` function from the `<ctype.h>` header file.

☑ Public?

Test file (main.c)

```c
#include <stdio.h>

void upcase(char *s);

int main() {
    char s[256];
    fgets(s, 255, stdin); // read string from stdin

    upcase(s); // convert string to uppercase

    printf(s); // write string to stdout

    return 0;
}

```

Submission template (sub.c)

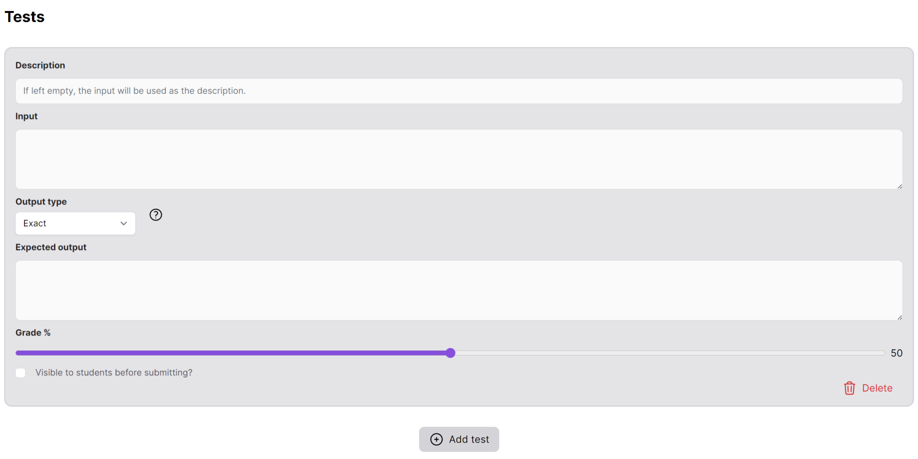
```c
void upcase(char *s) {

}

```

Figure 20: First part of the exercise creation form, filled according to the given example.

The second part of the exercise creation form can be seen in Figure 21 and pertains to the exercise's tests.



Tests

Description

If left empty, the input will be used as the description.

Input

Output type

Exact

Expected output

Grade %

50

☐ Visible to students before submitting?

Delete

Add test

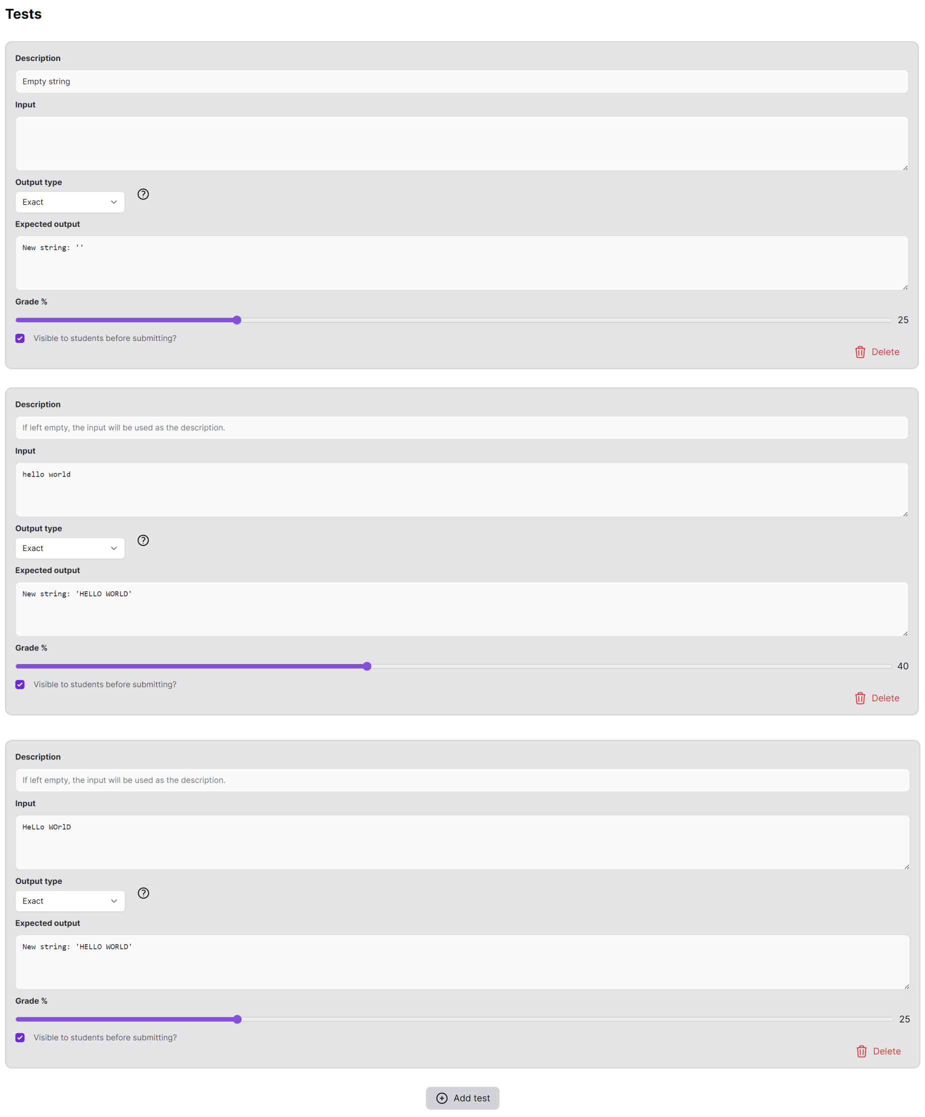
Figure 21: Second part of the exercise creation form.

The structure of a test in ACE Grader has already been described previously, except for the first field, the "description". This field can be used to describe the purpose of a test, so that students, when viewing feedback for their submission, are able to understand what exactly their solution can or cannot do. If this field is left empty, the test input will be used as the description.

ACE Grader allows teachers to create as many tests as they desire, as long as the sum of every test is equal to 100. The "Add test" and the "Delete" buttons can be used to add and remove tests, respectively.

For this example, we can define 3 tests, one for an empty string, one for a string with just lowercase letters, and one for a string with both lowercase and uppercase letters. More tests could be defined, but, for practical purposes, these tests already cover most scenarios.

After filling out the second part of the form, the page looks like the pictures from Figure 22.


Tests

Description
Empty string
Input

Output type
Exact
Expected output
```
New string: ''
```
Grade %
25
☑ Visible to students before submitting?

Description
If left empty, the input will be used as the description.
Input
```
hello world
```
Output type
Exact
Expected output
```
New string: 'HELLO WORLD'
```
Grade %
40
☑ Visible to students before submitting?

Description
If left empty, the input will be used as the description.
Input
```
HeLLo WOrlD
```
Output type
Exact
Expected output
```
New string: 'HELLO WORLD'
```
Grade %
25
☑ Visible to students before submitting?

Figure 22: Second part of the exercise creation form, filled according to the given example.

These tests were left visible to students. This was an arbitrary decision. As mentioned before, public tests can be seen by students before submission. This allows students to check if their solution passes those tests before they submit.

The total grade only adds up to 90, instead of 100. This is not a mistake, the remaining 10% will be used for the parameters, which is the last part of the exercise creation form. Figure 23 shows the empty version of this final section.

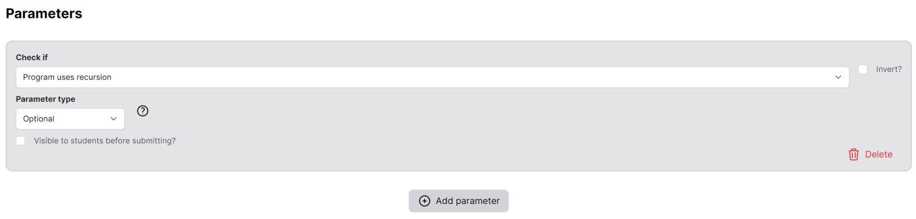
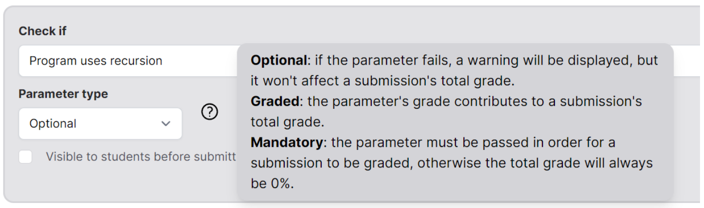


Figure 23: Third part of the exercise creation form.

These fields have already been described in detail in the previous Chapter. However, if a teacher is unsure of what each parameter type means, they can click on the question mark, which will reveal a pop-up describing each type of parameter in detail, as can be seen in Figure 24. The question mark in the Tests section of the form has the same behavior, revealing a pop-up with a description for each output type.

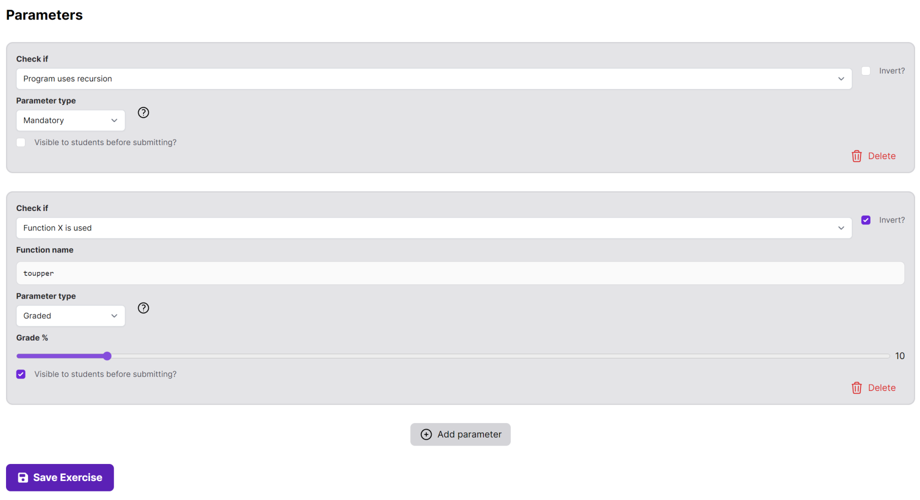


Optional: if the parameter fails, a warning will be displayed, but it won't affect a submission's total grade.
Graded: the parameter's grade contributes to a submission's total grade.
Mandatory: the parameter must be passed in order for a submission to be graded, otherwise the total grade will always be 0%.

Figure 24: Description of what each parameter type means.

Going back to the example, we can define two parameters, one stating that the program must use recursion (this should be a mandatory parameter) and one stating that the function `toupper()` should not be used (since the word "should" is used, instead of "must", this parameter can be graded instead of mandatory, and have a grade of 10%). For this demonstration, the mandatory parameter can be left as "private" so that students must pay attention to the problem definition instead of relying on ACE Grader's automatic feedback before submission.

Figure 25 shows the final part of the form after being filled.

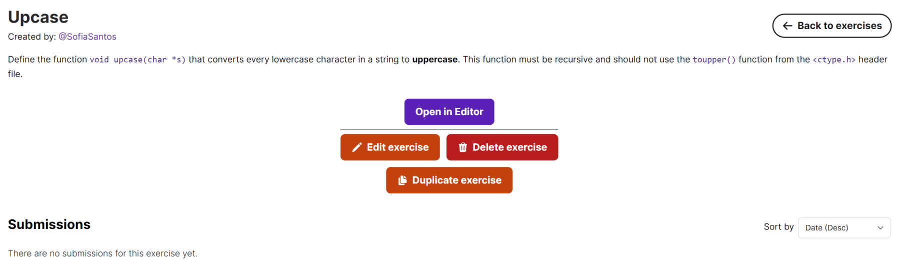

Parameters

Check if: Program uses recursion — Invert? ☐
Parameter type: Mandatory
☐ Visible to students before submitting?

Check if: Function X is used — Invert? ✓
Function name: `toupper`
Parameter type: Graded
Grade %: 10
✓ Visible to students before submitting?

Figure 25: Third part of the exercise creation form, filled according to the given example.

After the entire form is filled, one can press the "Save exercise" button, which will add the exercise to ACE Grader and redirect the user to the newly created exercise's page, as shown in Figure 26.


Upcase

Created by: @SofiaSantos

Define the function `void upcase(char *s)` that converts every lowercase character in a string to **uppercase**. This function must be recursive and should not use the `toupper()` function from the `<ctype.h>` header file.

Submissions

Sort by: Date (Desc)

There are no submissions for this exercise yet.

Figure 26: Exercise page for the example exercise.

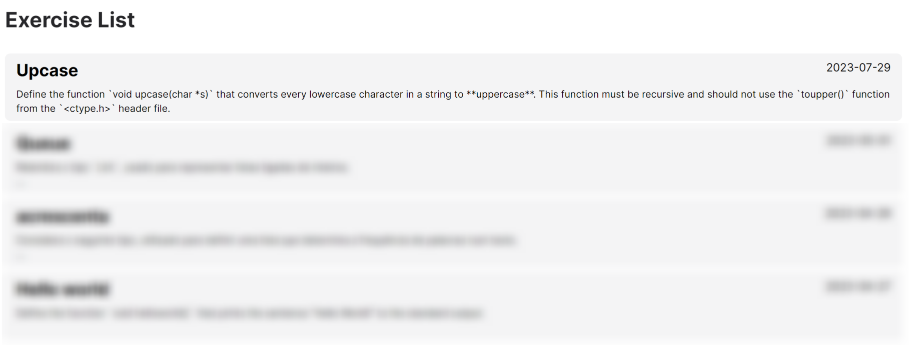
Through the "Edit exercise" button, the exercise author can edit the exercise and its tests/parameters. It's also possible to add and/or remove tests and parameters for existing exercises. The "Delete exercise" button, as the name implies, can be used only by the exercise author to delete the exercise and every submission associated with it.

Lastly, the "Duplicate exercise" button allows other teachers to create a copy of the exercise, which they are then able to freely edit. This button is particularly useful for classes with many shifts, when other shifts are taught by other teachers, they can simply duplicate an exercise and give it to their students, making changes to it if needed.

5.2 SUBMITTING A SOLUTION

Another very important feature of ACE Grader is the ability to submit solutions to exercises. This section will showcase this process from the point of view of a student.

In this scenario, a student called Jane Doe wants to solve the exercise that was created in the previous section. To achieve this, she starts by going to the exercise list page and selecting the "Upcase" exercise.



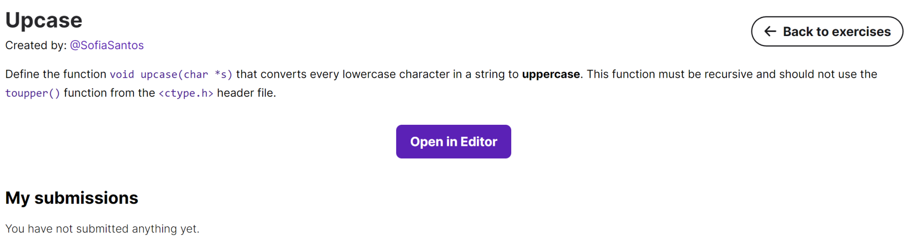
Exercise List

Upcase 2023-07-29

Define the function `void upcase(char *s)` that converts every lowercase character in a string to **uppercase**. This function must be recursive and should not use the `toupper()` function from the `<ctype.h>` header file.

Figure 27: Selecting the "Upcase" exercise from the exercise list.

Figure 28 shows the page for the "Upcase" exercise from Jane's point of view. Because she is a student, the only possible action for Jane is to open the exercise in the online editor, which she can use to create a submission. Since she hasn't made any submissions yet, the submissions section is empty.



Upcase

Created by: @SofiaSantos

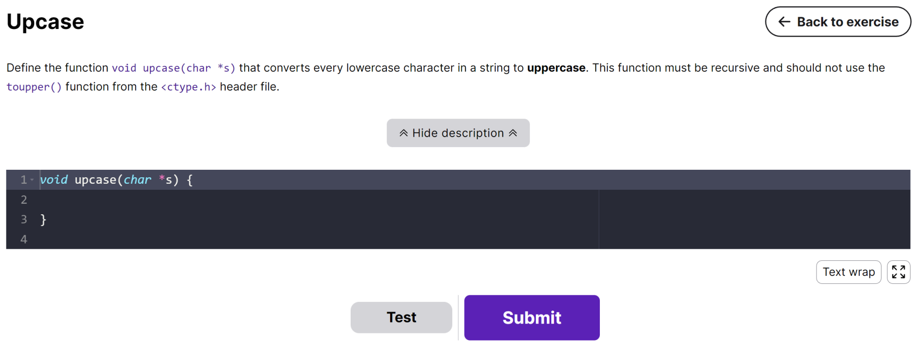
Define the function `void upcase(char *s)` that converts every lowercase character in a string to **uppercase**. This function must be recursive and should not use the `toupper()` function from the `<ctype.h>` header file.

My submissions

You have not submitted anything yet.

Figure 28: Page for the "Upcase" exercise.

After clicking on the "Open in Editor" button, Jane is redirected to the online editor, as can be seen in Figure 29, with the template that was defined in the previous section.



Upcase

Back to exercise

Define the function `void upcase(char *s)` that converts every lowercase character in a string to **uppercase**. This function must be recursive and should not use the `toupper()` function from the `<ctype.h>` header file.

Hide description

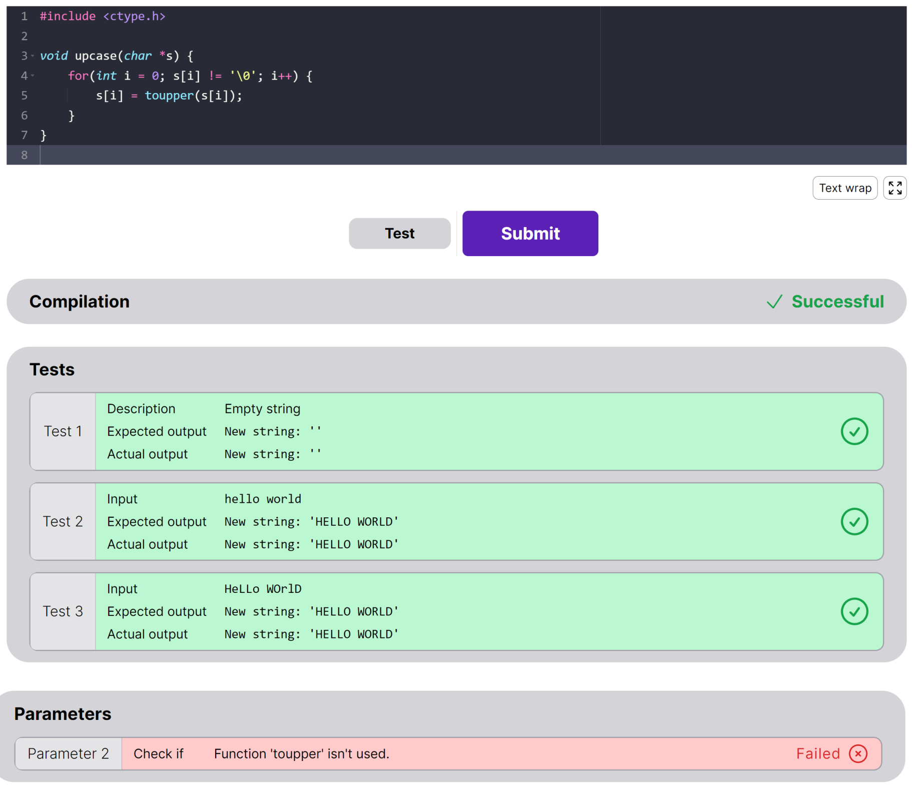
```c
void upcase(char *s) {

}

```

Figure 29: Code editor for the "Upcase" exercise.

Now, Jane can try to solve this exercise as she sees fit. First, she writes the following solution and presses the "Test" button:



```c
#include <ctype.h>

void upcase(char *s) {
    for(int i = 0; s[i] != '\0'; i++) {
        s[i] = toupper(s[i]);
    }
}

```

Compilation ✓ Successful

Tests

```
Test 1   Description       Empty string
         Expected output   New string: ''
         Actual output     New string: ''               ✓

Test 2   Input             hello world
         Expected output   New string: 'HELLO WORLD'
         Actual output     New string: 'HELLO WORLD'    ✓

Test 3   Input             HeLLo WOrlD
         Expected output   New string: 'HELLO WORLD'
         Actual output     New string: 'HELLO WORLD'    ✓
```

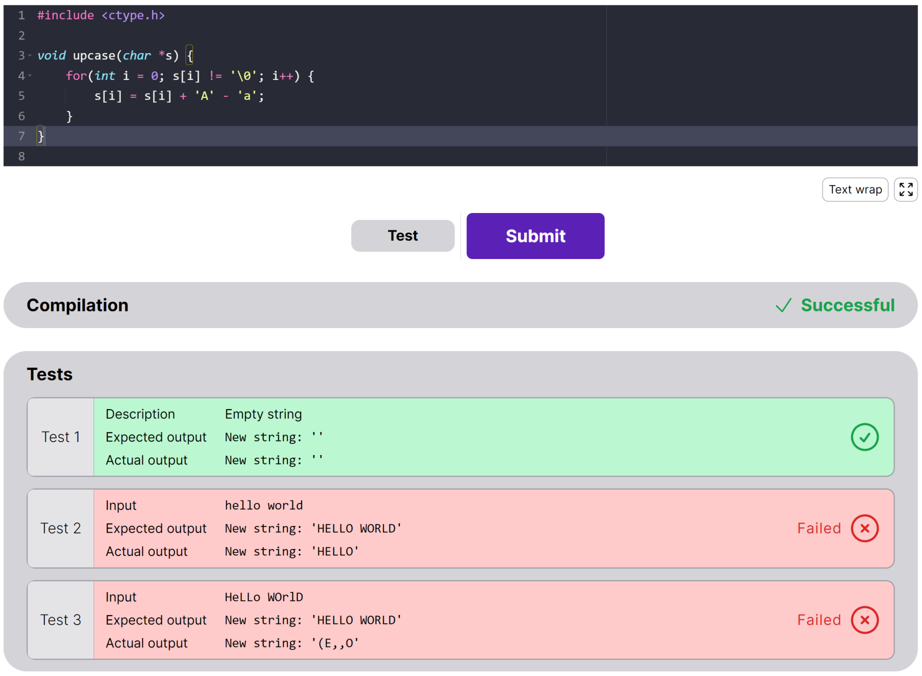
Parameters

```
Parameter 2   Check if   Function 'toupper' isn't used.   Failed ⊗
```

Figure 30: Jane's first solution.

Despite passing every test, this solution does not pass the parameter that requires the solution to not use the `toupper()` function. Now, Jane has two options. She can submit her current incomplete solution or try to improve this solution in order to pass the parameter.

In this scenario, Jane doesn't submit this solution. Instead, she refactors it and comes up with this second version:



```c
#include <ctype.h>

void upcase(char *s) {
    for(int i = 0; s[i] != '\0'; i++) {
        s[i] = s[i] + 'A' - 'a';
    }
}
```

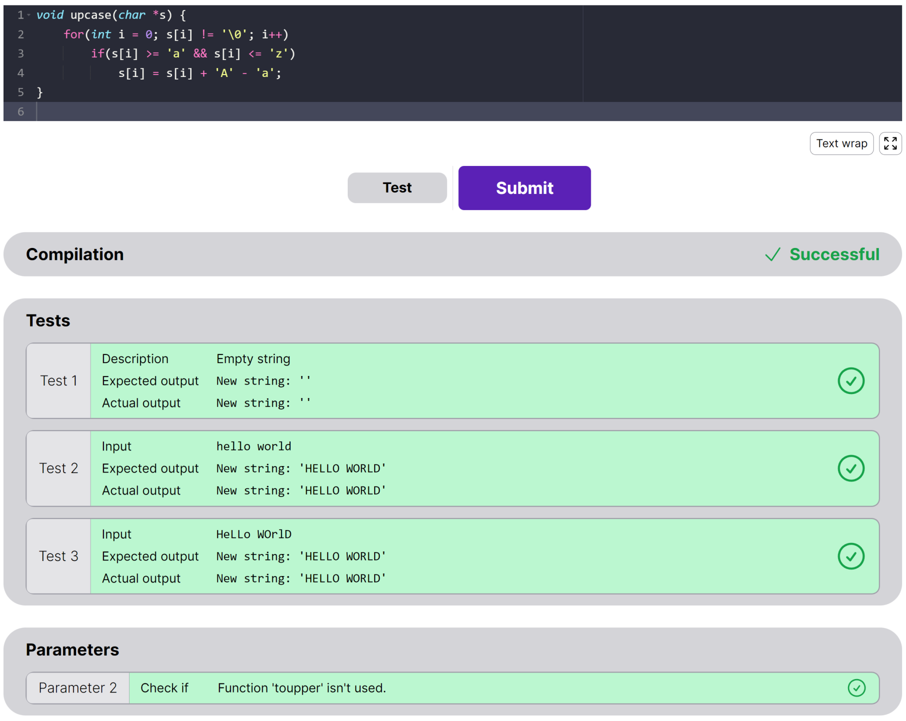
Compilation ✓ Successful

Tests

Test 1	Description	Empty string	✓
	Expected output	`New string: ''`	
	Actual output	`New string: ''`	
Test 2	Input	`hello world`	Failed ✗
	Expected output	`New string: 'HELLO WORLD'`	
	Actual output	`New string: 'HELLO'`	
Test 3	Input	`HeLLo WOrlD`	Failed ✗
	Expected output	`New string: 'HELLO WORLD'`	
	Actual output	`New string: '(E,,O'`	

Figure 31: Jane's second solution.

This solution fails because it tries to convert every character to uppercase, not just lowercase letters. Therefore, Jane comes up with a third version:



```c
void upcase(char *s) {
    for(int i = 0; s[i] != '\0'; i++)
        if(s[i] >= 'a' && s[i] <= 'z')
            s[i] = s[i] + 'A' - 'a';
}
```

Compilation ✓ Successful

Tests

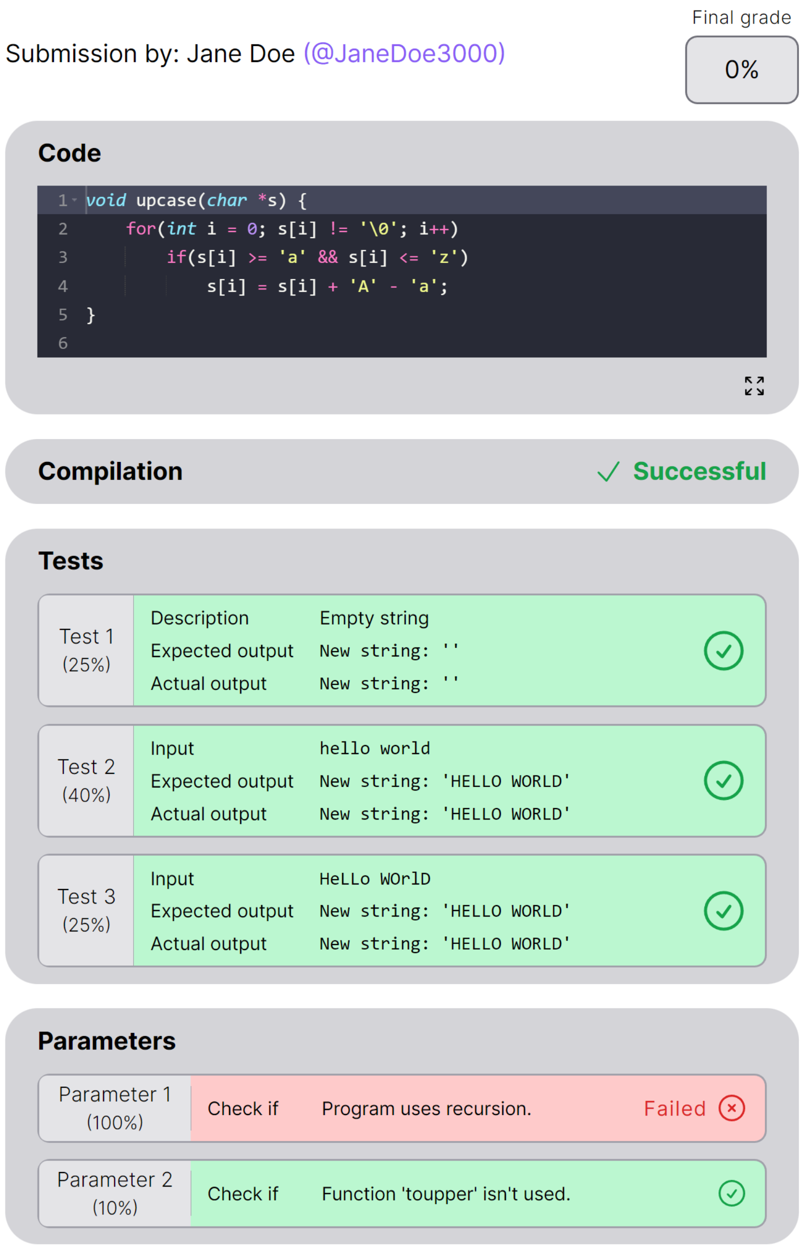
Test 1	Description	Empty string	✓
	Expected output	`New string: ''`	
	Actual output	`New string: ''`	
Test 2	Input	`hello world`	✓
	Expected output	`New string: 'HELLO WORLD'`	
	Actual output	`New string: 'HELLO WORLD'`	
Test 3	Input	`HeLLo WOrlD`	✓
	Expected output	`New string: 'HELLO WORLD'`	
	Actual output	`New string: 'HELLO WORLD'`	

Parameters

Parameter 2	Check if	Function 'toupper' isn't used.	✓

Figure 32: Jane's third solution.

This version seems to pass every test and parameter, so Jane submits it, which redirects her to the submission feedback page. Now, she can see that she didn't pass the "invisible" parameter, since her solution is not recursive, and therefore her final grade is 0%.



Submission by: Jane Doe (@JaneDoe3000)

Final grade: 0%

Code

```c
void upcase(char *s) {
    for(int i = 0; s[i] != '\0'; i++)
        if(s[i] >= 'a' && s[i] <= 'z')
            s[i] = s[i] + 'A' - 'a';
}
```

Compilation ✓ Successful

Tests

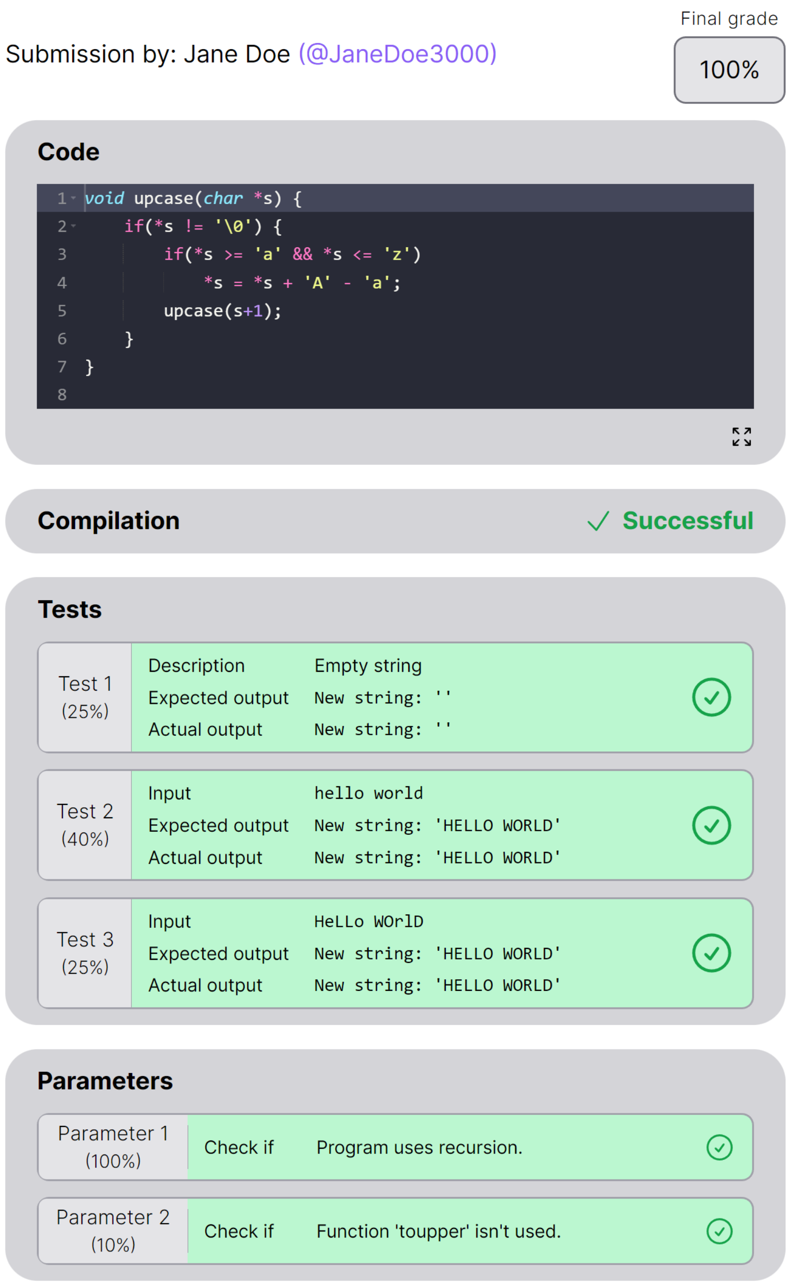
Test 1 (25%)	Description	Empty string	✓
	Expected output	`New string: ''`	
	Actual output	`New string: ''`	
Test 2 (40%)	Input	`hello world`	✓
	Expected output	`New string: 'HELLO WORLD'`	
	Actual output	`New string: 'HELLO WORLD'`	
Test 3 (25%)	Input	`HeLLo WOrlD`	✓
	Expected output	`New string: 'HELLO WORLD'`	
	Actual output	`New string: 'HELLO WORLD'`	

Parameters

Parameter 1 (100%)	Check if	Program uses recursion.	Failed ⊗
Parameter 2 (10%)	Check if	Function 'toupper' isn't used.	✓

Figure 33: Jane's submission feedback.

Now, Jane can try to submit another solution that passes this parameter. After coming up with a recursive solution, Jane submits this final solution, shown in Figure 34, which gives her a final grade of 100%.


Submission by: Jane Doe (@JaneDoe3000)

Final grade

100%

Code

```c
void upcase(char *s) {
    if(*s != '\0') {
        if(*s >= 'a' && *s <= 'z')
            *s = *s + 'A' - 'a';
        upcase(s+1);
    }
}

```

Compilation ✓ Successful

Tests

Test 1 (25%)	Description	Empty string	✓
	Expected output	`New string: ''`	
	Actual output	`New string: ''`	
Test 2 (40%)	Input	`hello world`	✓
	Expected output	`New string: 'HELLO WORLD'`	
	Actual output	`New string: 'HELLO WORLD'`	
Test 3 (25%)	Input	`HeLLo WOrlD`	✓
	Expected output	`New string: 'HELLO WORLD'`	
	Actual output	`New string: 'HELLO WORLD'`	

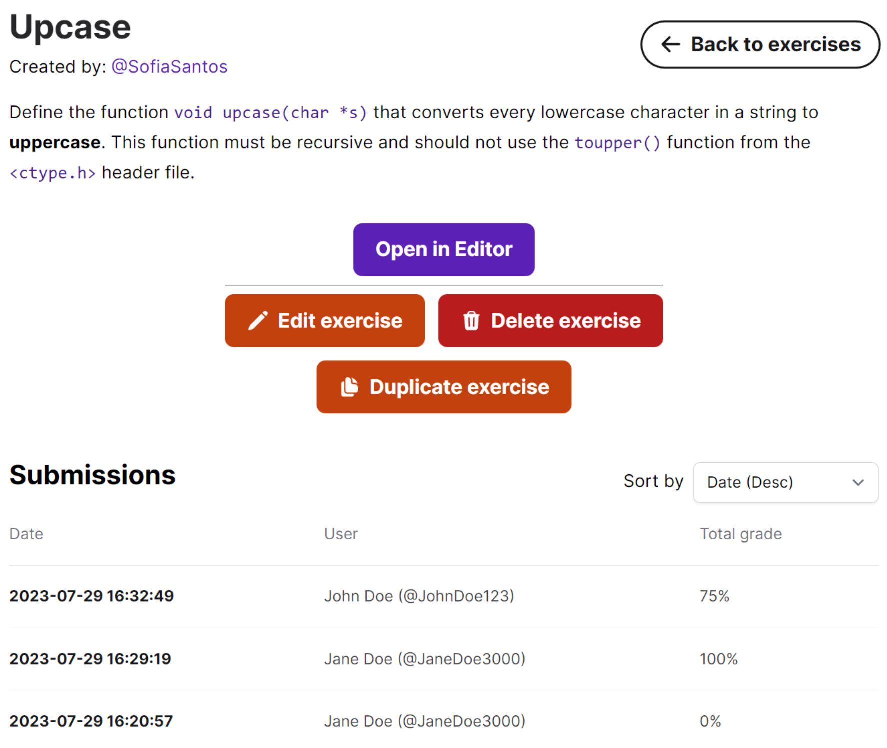
Parameters

Parameter 1 (100%)	Check if	Program uses recursion.	✓
Parameter 2 (10%)	Check if	Function 'toupper' isn't used.	✓

Figure 34: Jane's final submission.

5.3 REVIEWING SUBMISSIONS

Another important feature of ACE Grader is the ability to, as a teacher, review student submissions. If we return to the "Upcase" exercise using the account that created the exercise, we can now see some submissions. The submissions by Jane Doe were shown in the previous section, while a new submission was created by a user called John Doe, with a final grade of 75%, as can be seen in Figure 35.



Upcase

Created by: @SofiaSantos

Define the function `void upcase(char *s)` that converts every lowercase character in a string to **uppercase**. This function must be recursive and should not use the `toupper()` function from the `<ctype.h>` header file.

Submissions

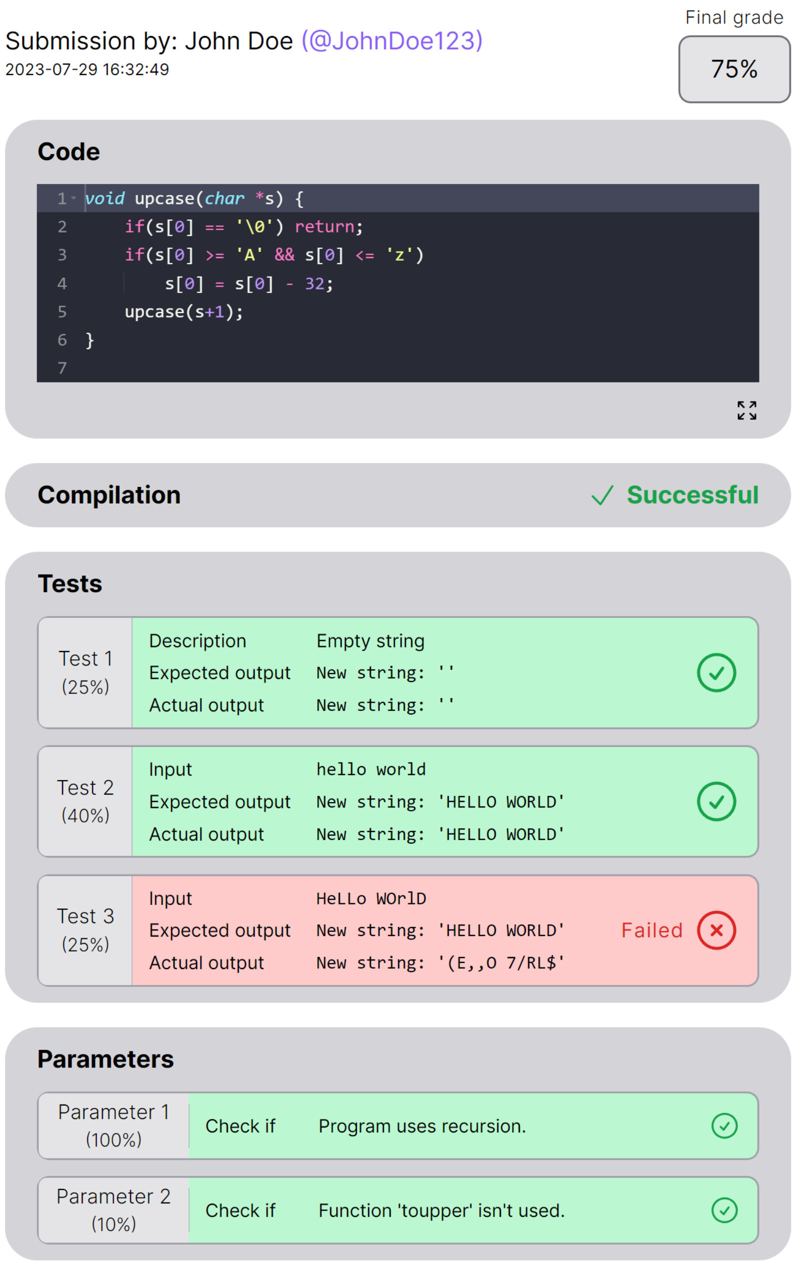
Sort by Date (Desc)

Date	User	Total grade
2023-07-29 16:32:49	John Doe (@JohnDoe123)	75%
2023-07-29 16:29:19	Jane Doe (@JaneDoe3000)	100%
2023-07-29 16:20:57	Jane Doe (@JaneDoe3000)	0%

Figure 35: Submissions for the "Upcase" exercise.

In this scenario, only two students have made submissions, making it easy to navigate through this page. However, in big classes with many students, the submissions section might contain dozens or hundreds of submissions. In situations like this, the "Sort by" option allows one to easily sort the submissions by username, date, or final grade, in ascending or descending order, so that teachers can go through their students' submissions in alphabetical order, for example, or check which students obtained the best/worst grades.

By opening John's submission, it's possible to see his solution and the individual grade for each test and parameter. Figure 36 shows the page for John's submission.



Submission by: John Doe (@JohnDoe123)
2023-07-29 16:32:49

Final grade
75%

Code

```c
void upcase(char *s) {
    if(s[0] == '\0') return;
    if(s[0] >= 'A' && s[0] <= 'z')
        s[0] = s[0] - 32;
    upcase(s+1);
}

```

Compilation ✓ Successful

Tests

Test 1 (25%)	Description	Empty string	✓
	Expected output	`New string: ''`	
	Actual output	`New string: ''`	
Test 2 (40%)	Input	`hello world`	✓
	Expected output	`New string: 'HELLO WORLD'`	
	Actual output	`New string: 'HELLO WORLD'`	
Test 3 (25%)	Input	`HeLLo WOrlD`	Failed ✕
	Expected output	`New string: 'HELLO WORLD'`	
	Actual output	`New string: '(E,,O 7/RL$'`	

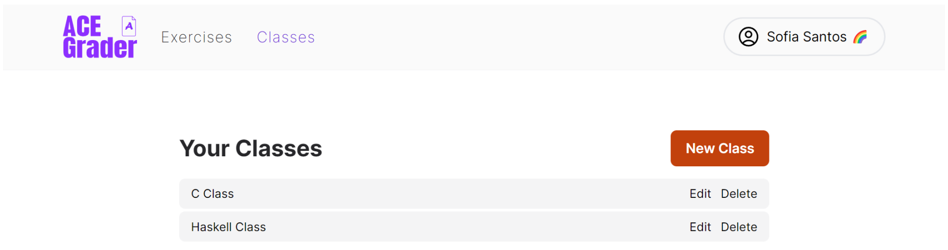
Parameters

Parameter 1 (100%)	Check if	Program uses recursion.	✓
Parameter 2 (10%)	Check if	Function 'toupper' isn't used.	✓

Figure 36: John Doe's submission for the "Upcase" exercise.

5.4 MANAGING CLASSES AND STUDENTS

As mentioned in the previous section, sometimes teachers are responsible for many classes. In situations like these, it can be useful to group students that belong to the same class together. For that purpose, ACE Grader contains a class management feature. To create a class, teachers must go to the "Classes" page, which can be accessed through ACE Grader's header. From there, they can see a list of their existing classes, as well as a button that allows them to create a new class.



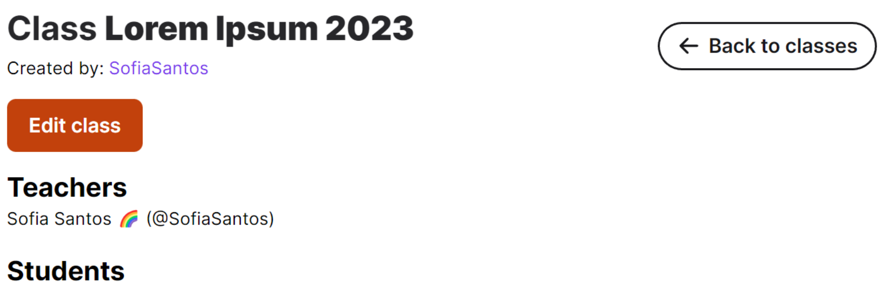
Your Classes

C Class

Haskell Class

Figure 37: Class management page.

After pressing the "New Class" button, one is prompted to define a name for a new class and is subsequently redirected to the newly created class's page. In this scenario, a class called "Lorem Ipsum 2023" was created. Figure 38 shows the page that the class creator sees after creating this class.



Class Lorem Ipsum 2023

Created by: SofiaSantos

Teachers

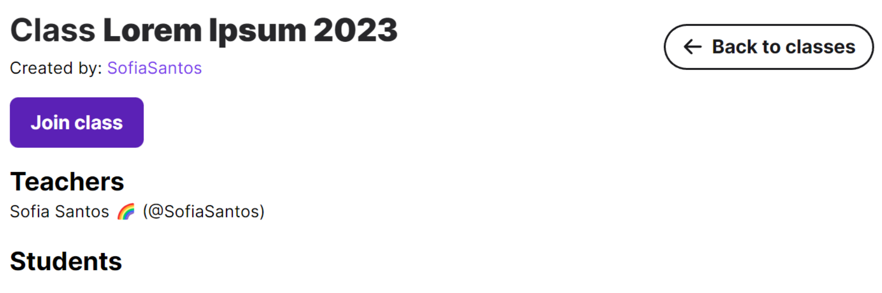
Sofia Santos 🌈 (@SofiaSantos)

Students

Figure 38: Page for a newly created "Lorem Ipsum 2023" class.

Every class is private, which prevents unwanted users from joining them. Therefore, for students to join a class, they must be sent the URL for that class by a teacher or by another student in that class. Since information about a class is stored in ACE Grader's database using UUIDs [37], it's virtually impossible for a malicious user to "guess" the URL for a specific class and join it.

In a scenario where Jane Doe wishes to join the "Lorem Ipsum 2023" class, if she is sent the URL for the class, after opening it, she would see the window shown in Figure 39.



Class Lorem Ipsum 2023

Created by: SofiaSantos

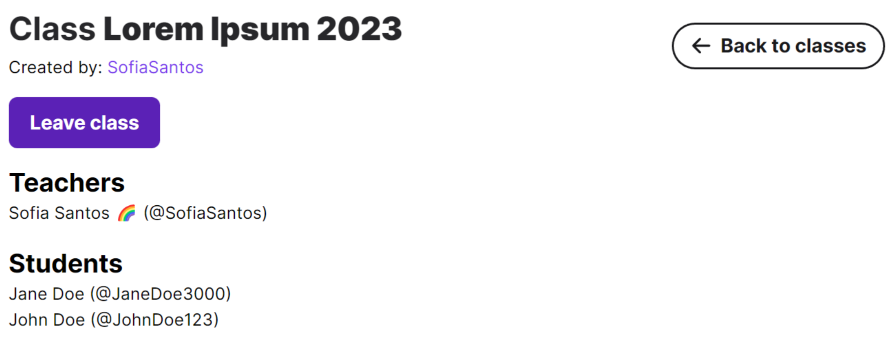
Teachers

Sofia Santos 🌈 (@SofiaSantos)

Students

Figure 39: Jane's point of view of the "Lorem Ipsum 2023" class page before joining.

The "Join class" button allows her to join this class. If Jane now sends the URL to John, and he also joins the class, he would see a window like the one shown in Figure 40. The "Join class" button was replaced with a "Leave class" button since John has already joined.



Class Lorem Ipsum 2023

Created by: SofiaSantos

Teachers

Sofia Santos 🌈 (@SofiaSantos)

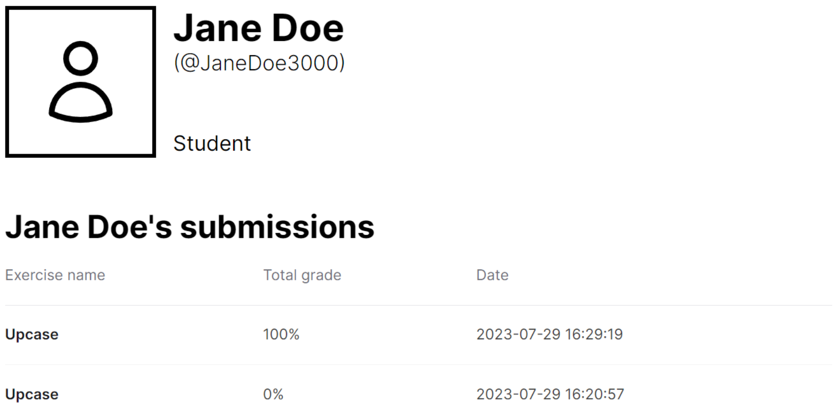
Students

Jane Doe (@JaneDoe3000)

John Doe (@JohnDoe123)

Figure 40: John's point of view of the "Lorem Ipsum 2023" class page after joining.

If another teacher were to join this class, they would appear in the "Teachers" section and would be able to click on a student's name to open that student's page. In this scenario, Jane's personal page can be seen in Figure 41. Only Jane herself or a teacher can see Jane's submissions when entering this page, for privacy purposes.


Jane Doe
(@JaneDoe3000)

Student

Jane Doe's submissions

Exercise name	Total grade	Date
Upcase	100%	2023-07-29 16:29:19
Upcase	0%	2023-07-29 16:20:57

Figure 41: Jane Doe's personal page.

Unfortunately, since the class management feature wasn't one of the main focuses of this project, there aren't any other uses for classes, aside from letting a teacher see a list of every student enrolled in a class. In the future, this feature could be expanded, allowing teachers to, for example, group submissions in an exercise by class.

6 CONCLUSION

In Chapter 1, the topic of this Master's project, automatic assessment of programming exercises, was introduced, and its objectives and research method were laid out.

Then, Chapter 2 summarized the research made on the topic. A general state of current tools was presented, along with a more in-depth description of assessment methods.

Chapter 3 described in more detail the proposal of the automatic assessment tool, ACE Grader, and went over its features. Through diagrams, the architecture of the application was presented, and a practical example provided an easy-to-understand overview of the tool's main features.

Chapter 4 covered the development process of ACE Grader, focusing on its two primary components, backend and frontend. It also covered the process of how ACE Grader grades a solution, with dynamic tests and static parameters.

Chapter 5 showcased some of ACE Grader's main features, with practical examples and screenshots of the application.

6.1 RESULTS AND DISCUSSION

Unfortunately, since ACE Grader's development process ended during the summer, there were no opportunities to test the final version of the application in a real classroom setting. Despite this, several experiments were conducted with earlier versions of ACE Grader throughout the second semester of the 2023/24 school year in Imperative Programming classes at Universidade do Minho, taught by me. In general, students gave positive feedback about ACE Grader, in particular about its user interface, ease of use, and accessibility features. Specific features, like being able to test solutions before submitting them, were regarded as positive. Some of the students, who had also used Codeboard in the past, admitted that, in general, they preferred ACE Grader.

From a teacher's perspective, there wasn't an opportunity to conduct in-depth testing, for the same reason as stated above, but since I was teaching some classes while developing ACE Grader, I was able to test ACE Grader in real scenarios as a teacher. This also allowed me to fine-tune ACE Grader in order to meet teachers' needs for an automatic grading application. Not every teacher and not every class is the same, but most educators have a common set of objectives, for which I believe ACE Grader is suitable.

6.2 CONTRIBUTIONS

ACE Grader is publicly accessible at `https://acegrader.epl.di.uminho.pt` and its source code can be consulted at `https://github.com/RisingFisan/ACE_Grader`. Since ACE Grader is an open-source project, anyone is free to contribute to the project, or modify the source code and create their own version.

As mentioned previously, ACE Grader can be executed as a multi-container Docker application, by using the Compose tool within Docker or by launching each container individually.

6.3 FINAL THOUGHTS AND FUTURE WORK

Going back to the original research hypothesis, this Master's Thesis presented a tool that uses both dynamic and static analysis to automatically grade programming exercises. In order to affirm that ACE Grader is preferable to other grading tools, one would need to conduct experiments and carefully study the results, something that goes beyond the reach of this project. However, the State-of-the-Art research showed that there are no equivalent tools to ACE Grader currently available for classroom usage, which means that, even without a survey, it is safe to assert that ACE Grader fills a hole in the area of automatic assessment software, making it useful in certain situations, for example, when one needs a combination of static and dynamic analysis in order to automatically grade a programming exercise.

Due to a shorter than expected development time, one of ACE Grader's planned features, partial grading, could not be implemented. In addition, some possible ideas for new features arose during development, which weren't further explored. The idea to expand the class management feature has already been discussed in Chapter 5, but, to recapitulate, it could be interesting to be able to group submissions by class, or to make an exercise only visible to a specific class or classes. Another idea that came up revolved around manual teacher feedback. In other words, while ACE Grader provides automatic feedback, sometimes there may be a need for a teacher to provide further feedback which cannot be easily automated. While this can be done directly between the teacher and the student, a future version of ACE Grader could have an option for a teacher to fill a text box with feedback for a particular submission, or for any submission that passes/fails a specific test or parameter, for example.

BIBLIOGRAPHY

[1] International olympiad in informatics. URL https://ioinformatics.org/.

[2] Swerc 2021-2022. URL https://swerc.eu/2021/about/.

[3] José Carlos Paiva. Mooshak 2, Sep 2016. URL https://mooshak2.dcc.fc.up.pt/.

[4] Cms :: Main. URL http://cms-dev.github.io/.

[5] Codeboard - the IDE for the classroom. URL https://codeboard.io/.

[6] Anjana Gosain and Ganga Sharma. A survey of dynamic program analysis techniques and tools. *Advances in Intelligent Systems and Computing*, page 113–122, 2015. doi: 10.1007/978-3-319-11933-5_13.

[7] Michael Striewe and Michael Goedicke. *A Review of Static Analysis Approaches for Programming Exercises*, page 100–113. Springer International Publishing, Cham, 2014. URL http://dx.doi.org/10.1007/978-3-319-08657-6_10.

[8] Kirsti M Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005. doi: 10.1080/08993400500150747.

[9] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. Automated assessment in computer science education: A state-of-the-art review. *ACM Trans. Comput. Educ.*, 22(3), jun 2022. doi: 10.1145/3513140. URL https://doi.org/10.1145/3513140.

[10] Vreda Pieterse. Automated assessment of programming assignments. *CSERC*, 13:4–5, 2013.

[11] SmartBear Software. What is unit testing? URL https://smartbear.com/learn/automated-testing/what-is-unit-testing.

[12] Hunit. URL https://hackage.haskell.org/package/HUnit.

[13] Dinis Cruz. Ast (abstract syntax tree), Oct 2018. URL https://medium.com/@dinis.cruz/ast-abstract-syntax-tree-538aa146c53b.

[14] The modern static analysis platform. URL https://deepsource.io/glossary/ast/.

[15] Better unused variable detection in psalm 4. URL https://psalm.dev/articles/better-unused-variable-detection.

[16] C. E. Beevers, D. G. Wild, G. R. McGuine, D.J. Fiddes, and M.A. Youngson. Issues of partial credit in mathematical assessment by computer. *ALT-J*, 7(1):26–32, 1999. doi: 10.1080/0968776990070105. URL https://doi.org/10.1080/0968776990070105.

[17] Philip E. Robinson and Johnson Carroll. An online learning platform for teaching, learning, and assessment of programming. In *2017 IEEE Global Engineering Education Conference (EDUCON)*, pages 547–556, 2017. doi: 10.1109/EDUCON.2017.7942900.

[18] Thomas Richard Rossi. Semi automated partial credit grading of programming assignments. 2016. URL https://scholars.unh.edu/thesis/1079.

[19] 0xffccdd. Virtual machines vs containers - 0xffccdd. *Medium*, Nov 2022. URL https://medium.com/@cloud_tips/virtual-machines-vs-containers-ef1b03bf2e6f.

[20] Jon Westfall. *Set up and manage your virtual private server: Making system administration accessible to professionals*. APress, Berlin, Germany, 1 edition, 2021. ISBN 9781484269657.

[21] Kattis. URL https://open.kattis.com/.

[22] Michael de Raadt, Richard Watson, and Mark Toleman. Language trends in introductory programming courses. *USQ ePrints*, Jan 1970. URL http://eprints.usq.edu.au/id/eprint/5195.

[23] Philip Guo. Python is now the most popular introductory teaching language at top u.s. universities, Jul 2014. URL https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext.

[24] S. Stoney and M. Wild. Motivation and interface design: Maximising learning opportunities. *Journal of Computer Assisted Learning*, 14(1):40–50, 1998. doi: 10.1046/j.1365-2729.1998.1410040.x.

[25] Exercism. URL https://exercism.org/.

[26] Python enhancement proposals. URL https://peps.python.org/pep-0008/#prescriptive-naming-conventions.

[27] beecrowd. URL https://www.beecrowd.com.br.

[28] Test my code exercise server. URL https://github.com/testmycode/tmc-server.

[29] Concurrent programming - erlang. URL http://erlang.org/documentation/doc-5.6/doc/getting_started/conc_prog.html.

[30] A brief introduction to beam - erlang/otp. URL https://www.erlang.org/blog/a-brief-beam-primer/.

[31] Contributors to Wikimedia projects. Fork bomb, Apr 2023. URL https://en.wikipedia.org/wiki/Fork_bomb.

[32] Clang C Language Family Frontend for LLVM — clang.llvm.org. https://clang.llvm.org/, . [Accessed 08-Jun-2023].

[33] Introduction to the Clang AST. https://clang.llvm.org/docs/IntroductionToTheClangAST.html, . [Accessed 08-Jun-2023].

[34] clang — pypi.org. https://pypi.org/project/clang/. [Accessed 08-Jun-2023].

[35] Docker: Accelerated, containerized application development. URL https://www.docker.com/.

[36] Markdown guide. URL https://www.markdownguide.org/.

[37] Complete guide to universal unique identifiers (uuid). URL https://www.uuidtools.com/what-is-uuid.