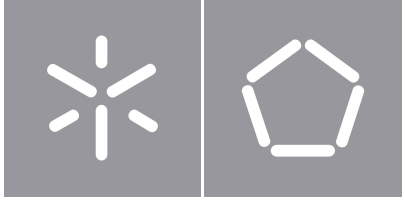


University of Minho
School of Engineering

João Luís Tibo Machado dos Santos

Stepwise Source, a Tool for Progressive Source Code Demonstration

october 2024



University of Minho
School of Engineering

João Luís Tibo Machado dos Santos

Stepwise Source, a Tool for Progressive Source Code Demonstration

Masters Dissertation
Master's in Informatics Engineering

Dissertation supervised by
Professor Pedro Rangel Henriques
Professor Alvaro Costa Neto

october 2024

Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

License granted to users of this work:



CC BY-NC

https://creativecommons.org/licenses/by-nc/4.0/

Agradecimentos

Ao fim de cinco anos na universidade, são muitas as pessoas que me marcaram, que me ajudaram a tornar-me na pessoa que sou hoje e a completar este percurso, e a quem gostaria de agradecer. Antes de mais, um muito obrigado aos meus orientadores, Professor Pedro Rangel Henriques e Professor Alvaro Costa Neto, por todo o apoio ao longo desta dissertação, pela disponibilidade, pela motivação e pela confiança que depositaram em mim, não podia ter pedido melhor. Um enorme obrigado à minha família, mãe, pai e irmãos, por ter estado sempre ao meu lado. Por me ter criado e orientado da melhor forma possível e por me permitir embarcar nesta viagem e aproveitá-la ao máximo. Aos meus amigos, pelos momentos inesquecíveis que vivenciamos juntos, pelo seu apoio e presença ativa neste percurso académico. Levarei para sempre comigo as memórias de tudo aquilo que passámos e espero continuar a construir outras tantas. Por último, mas não menos importante, um agradecimento sentido à pessoa que mais de perto acompanhou todos estes momentos, os bons e os menos positivos. A pessoa que sempre acreditou e que por vezes me fez acreditar quando eu próprio duvidava. Muito obrigado à minha namorada. Como diria Antoine de Saint-Exupery, "Aqueles que passam por nós não vão sós. Deixam um pouco de si, levam um pouco de nós". Obrigado a todos aqueles que passaram por mim.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, october 2024

João Luís Tibo Machado dos Santos

Abstract

This thesis, conducted as part of a Master's Degree in Informatics Engineering at the University of Minho in Braga, Portugal, introduces Stepwise Source, a web application to aid in bridging gaps in programming education, by tackling specific challenges faced by educators and learners. The primary goal of this project was to develop a system that enabled educators to create progressive demonstrations of source code construction in any programming language. Stepwise Source is the answer to this By presenting code incrementally, it would allow learners to move forward or backward as they follow along. Additionally, educators can set challenges where students complete specific portions of the code with real-time feedback on their answers. It is also possible to create branches in the presentation flow to show alternative code paths that achieve different outcomes or to demonstrate multiple equivalent implementations for a programming concept. By offering flexibility for both guided interaction and independent study, this tool enhances the learning experience in programming education, offering a meaningful resource to support the long and arduous journey of learning how to program.

Keywords Computer Programming Education, Source Code Demonstration, Education Technology.

Resumo

Esta tese, realizada no âmbito de um Mestrado em Engenharia Informática na Universidade do Minho, em Braga, Portugal, investiga lacunas no ensino da programação e identifica desafios específicos enfrentados por educadores e alunos. Foi realizado um estudo preliminar destinado a analisar estes desafios e as áreas nas quais as atuais ferramentas educativas para a programação são insuficientes. O objetivo primário deste projeto foi desenvolver uma ferramenta que permita aos professores criar demonstrações progressivas de construção de código-fonte. O sistema apresenta o código sequencialmente, permitindo aos alunos avançar ou recuar à medida que o acompanham. Mais ainda, os professores podem estabelecer desafios onde os alunos completem partes específicas do código e possam ramificar a demonstração para mostrar caminhos diferentes para atingir resultados diferentes ou apresentar várias implementações alternativas para um mesmo resultado. Proporcionando flexibilidade tanto para a interação assistida como para o estudo independente, esta ferramenta melhora a experiência de aprendizagem no ensino da programação, oferecendo um recurso significativo tanto para alunos como para professores no domínio do pensamento computacional e das competências em escrita de código.

Palavras-chave Ensino de Programação, Demonstração de Construção de Código-Fonte, Tecnologia Educacional.

Contents

1 Introduction **1**

1.1 Objectives . . . 2

1.2 Research Hypothesis . . . 2

1.3 Research Methodology . . . 2

1.4 Document Structure . . . 3

2 State of the Art **4**

2.1 Computer Programming . . . 4

2.1.1 Programming Languages . . . 4

2.1.2 Compilers and Interpreters . . . 6

2.2 Backus-Naur Form (BNF) . . . 6

2.2.1 Structure of BNF Rules . . . 7

2.2.2 Abstract Syntax Tree . . . 8

2.2.3 Extended Backus-Naur Form (EBNF) . . . 9

2.3 Computer Programming Education . . . 10

2.3.1 Factors that Interfere when Learning Computer Programming . . . 10

2.4 Programming Education Tools . . . 13

2.4.1 Automatic Assessment Tools . . . 13

2.4.2 Source Code Animation Tools . . . 13

2.4.3 Source Code Demonstration Tools . . . 14

2.5 Summary . . . 14

3 Stepwise Source, Architecture **16**

3.1 Parser . . . 16

3.2 Aggregator . . . 17

3.3 Answer Interpreter . . . 17

3.4 Presentation Generator . . . 17
3.5 Summary . . . 17

4 Stepwise Language 19

4.1 Overview and Use . . . 19
4.1.1 Defining a Slide . . . 19
4.1.2 Styling . . . 21
4.1.3 Macros . . . 22
4.1.4 Styling Keys and Attributes . . . 23
4.1.5 Interactive Exercises . . . 24
4.1.6 Presentation Flow . . . 25
4.2 Formal Definition . . . 26
4.3 Summary . . . 29

5 Stepwise Source, Development 30

5.1 The Parser and Interpreter . . . 30
5.1.1 Interpreter Class . . . 30
5.2 Database . . . 34
5.3 Backend . . . 35
5.3.1 Connection with the Database . . . 35
5.3.2 Connection with the Web Application . . . 37
5.4 Web Application . . . 40
5.5 Deployment . . . 44
5.5.1 Backend Dockerfile . . . 44
5.5.2 Frontend Dockerfile . . . 45
5.5.3 Docker Compose . . . 46
5.6 Summary . . . 46

6 Stepwise Source, Overview 47

6.1 Home Page . . . 47
6.2 About Page . . . 48
6.3 Presentations List Page . . . 49
6.4 Documentation Page . . . 49
6.5 Presentation Editor Page . . . 52

6.6 Presentation Navigator Page 53
6.6.1 Slide Content and Annotations 54
6.6.2 Presentation Navigation 55

7 Conclusion **56**

7.1 Assessment 56
7.2 Final Remarks 57
7.3 Future Work 57

List of Figures

1 Abstract Syntax Tree generated from an order. 9

2 Stepwise Source's architecture. 16

3 Home page. 47

4 About page. 48

5 Presentation List page. 49

6 Landing area of the Documentation page. 50

7 Documentation section of the Documentation page. 51

8 Tutorial section of the Documentation page. 51

9 Presentation Editor page. 52

10 Presentation naming panel. 53

11 Presentation creation confirmation panel. 53

12 Presentation Navigator Page. 54

13 Presentation Navigation System. 55

List of Tables

1 Comparison between BNF and EBNF. 10

2 Translation of DSL styling keys and attributes to CSS. 24

List of Listings

4.1 Syntax for a slide with an optional description. . . . 20
4.2 Example of a slide on variable declaration. . . . 20
4.3 Syntax for the declaration of new styling commands. . . . 21
4.4 Example of use for the `\newreactive` and `\newfixed` commands. . . . 22
4.5 Syntax for the declaration of macros. . . . 22
4.6 Example of a macro. . . . 23
4.7 Syntax for exercises creation. . . . 24
4.8 Example of an interactive exercise. . . . 25
4.9 Syntax for the `\order` command. . . . 25
4.10 Example of the `\order` command. . . . 26
4.11 Initial production rule. . . . 26
4.12 Slide declaration production rule. . . . 27
4.13 Description production rule. . . . 27
4.14 Production rule that corresponds to the content of a slide. . . . 27
4.15 Production rules to declare new styling commands. . . . 28
4.16 Macro production rule. . . . 28
4.17 Presentation order production rule. . . . 29
4.18 Ignoring whitespace. . . . 29
5.1 Init function and class properties. . . . 31
5.2 Function that translates the DSL's styling keys and attributes to CSS. . . . 32
5.3 Function that parses fixed style command declarations. Its reactive command counterpart is almost identical. . . . 32
5.4 Functions that parse the presentation order command. . . . 33
5.5 Slide structure initialization function. . . . 33
5.6 Slide's content processing functions. . . . 34

5.7 Order Syntax Example. 35
5.8 Basic structure of the Annotation model. 36
5.9 Definition of `exercises` field in the `slideSchema`. 38
5.10 Definition of `exerciseSchema`. 38
5.11 Exporting the `annotationSchema` model. 38
5.12 Function that lists all the existing presentations. 39
5.13 Connection between the Presentation List page and both the Editor and Navigator. . . . 41
5.14 Operations in the Editor page. 42
5.15 Component that applies slide styling into HTML. 42
5.16 Navigation graph rendering. 43
5.17 Overview of Stepwise Source's implementation. 44

Acronyms

AST Abstract Syntax Tree.

BNF Backus-Naur Form.

DSL Domain-Specific Language.

EBNF Extended Backus-Naur Form.

PDR Preliminary Dissertation Report.

SOA State of the Art.

Chapter 1

Introduction

The concept behind computer programming is to create a sequence of instructions that must be followed in order to carry out specific tasks. Although it involves writing code, i.e. text that follows a specific machine-processable syntax, writing a program begins with a thorough understanding of the problem to be solved. This process involves analysing the problem and breaking it down into increasingly smaller components, ultimately allowing the corresponding code to be written.

Ultimately, programming is the act of producing a software solution to a certain problem. This process is also about breaking down the larger problem into smaller ones, and solving them. This ability is one of the major developments that student need to achieve in order to reason, and effectively, realise programming.

It can be seen that there is a high failure rate among students when it comes to learning programming [Hundhausen and Douglas, 2000], due to failures throughout the various stages of this process. There are many flaws that contribute to this problematic situation, ranging from lack of familiarity with algorithmic thinking, inadequate interpretation of the problem by the student, difficulties in breaking down the problem and understanding the syntax of the language [Proulx, 2000], to the decline or absence of motivation [Tavares et al., 2015, Araujo, 2022, Costa Neto et al., 2021, Galvão et al., 2022].

Optimising the learning and teaching of computer programming can be achieved by integrating tools specifically designed for this purpose. These dedicated tools offer valuable assistance, tackling specific challenges inherent to this educational process.

Tools have been developed to visualise or animate the behaviour of programs. Others have been created with the aim of automatic grading. All of them are aimed primarily at improving students' understanding. While it was observed that the students found it easy to understand pre-conceived algorithms, developing them from scratch proved to be more challenging [Tavares et al., 2015].

With animation, the process of understanding is helped. It facilitates analysis and can also aid development. However, after various experiments and their analysis, it was clearly concluded that the use of animations alone does not improve learning as expected [Hundhausen et al., 2002].

On the other hand, automatic assessment provides means of encouraging students to take an active stance and improve their performance by stimulating their motivation and self-confidence. This is achieved by easing and accelerating the feedback on how the study is going.

In the light of what is mentioned above, it is pertinent to investigate alternative approaches that may be more effective or suitable for the circumstances at hand.

1.1 Objectives

The primary objective of this Master's Degree project is the design and development of an educational support tool (Stepwise Source) that can be used to create progressive demonstrations of source code construction, enriched by annotations. Moreover, this work aims to achieve the following key objectives:

- Enable the educator to define different branches to show how to change the code to produce the same result and how changes in the code may lead to distinct outcomes;
- Make it an interactive experience, allowing the learner to fill in gaps in the code;
- Turn it into a personalised user experience.

1.2 Research Hypothesis

It is possible to contribute to the teaching of computer programming through a progressive demonstration tool for the construction of source code.

1.3 Research Methodology

Throughout the academic year, this Master's work unfolded in adherence to an iterative methodology, comprising the following steps:

- In-depth exploration of existing literature and research works relevant to the project
- Construction of a state-of-the-art compilation of similar projects and tools;
- Design of the education support tool and its main features for implementation;
- Development of the education support tool;

- Evaluation of the education support tool as well as the discussion of its findings;
- Publication of the results and findings obtained from previous steps (*underway*).

1.4 Document Structure

This document is organised into the following chapters:

Introduction: The opening chapter provides an overview of the document's purpose, context, and objectives. It sets the stage for the reader by introducing the topic of investigation, the rationale and context behind the research, and the overall structure of the document;

State of the Art (Chapter 2): An extensive literature review covers key theories, methodologies, and technologies relevant to the project. It explores the current state of the field, establishing the foundation upon which the Master's project is built;

Architecture (Chapter 3): The main components that enable the project are introduced here, with explanations of each element's purpose and how they interact and collectively support the project's objectives;

Language (Chapter 4): The development of a **Domain-Specific Language (DSL)** was essential for this project. This section discusses the motivation, design process, and main components of Stepwise Language;

Development (Chapter 5): This part provides a detailed account of the components developed for this tool, including the technologies and processes employed in its construction;

Overview (Chapter 6): Focuses on the web application as the final product, examining each of its components, their functions, and the intended educational outcomes. This chapter provides a detailed look at how the application supports programming education for both students and educators;

Conclusion (Chapter 7): The concluding chapter summarizes the key findings, insights, and contributions of this project. It offers a reflective discussion on the outcomes of the research, highlights any implications, presents feedback gathered about the tool, evaluating how well it meets its objectives, and suggests potential avenues for future work. This chapter serves as a culmination of the document, leaving the reader with a comprehensive understanding of the research journey and its impact.

Chapter 2

State of the Art

This chapter explores the background theories and developments that support this Project. An overview of the state of the art in the fields of teaching programming and the tools that aid its improvement concludes the chapter establishing a foundation for the rest of the document.

2.1 Computer Programming

Described as the mental activity of thinking about instructions, a technological process for task delegation, or the formulation of specific instructions, programming essentially serves as a means for humans to employ computers to solve problems, transcending language barriers for effective communication between human and machine.

Programming is the methodical process of devising strategies to solve problems through computer execution. It involves crafting a set of instructions in a language executable by computers, enabling them to transform data into desired outcomes and address specific user needs. This collaborative effort between humans and computers requires critical thinking to provide precise instructions, resembling the creation of a recipe for a computer to follow [Coursera, 2023, Codecademy, 2024, Khan Academy, 2024a, Tavares et al., 2015, Araujo, 2022].

2.1.1 Programming Languages

A language is a system of communication that involves the use of symbols, such as words or gestures, to convey meaning. It is a complex and uniquely human ability that enables individuals to express thoughts, ideas, emotions, and information to others. Language serves various functions, including communication, expression, social interaction, and the transmission of culture. Some key components of a written language are: Alphabet, i.e. a set of symbols used to build words of a certain language; Lexis, being a set of words the language offers its users; Syntax which is a set of rules (formal or informal, written or felt intuitively)

used to determine if a certain string of words forms a valid sentence; Semantics meaning a set of rules determining if a certain phrase makes sense [Cisco Networking Academy, 2024].

Programming languages are no different from that: they are also a system of communication, a way to express thoughts and ideas. Computers operate in binary form—strings of ones and zeroes, trues or falses, high or low voltages. Programming languages allow programmers to translate something that humans can understand to this binary form. A programming language is made up of a series of symbols that serves as a bridge that allow humans to translate thoughts into instructions computers can process [Codecademy, 2020].

Given it is a written language, programming languages have the same components as their natural counterparts:

Alphabet (Character Set): The set of characters allowed in the language, including letters, digits, symbols, and special characters. It forms the basic building blocks for writing code;

Lexis (Lexical Elements): The vocabulary or set of symbols and keywords recognized by the programming language. It encompasses identifiers (variable and function names), keywords, numerals, and operators;

Syntax: The set of rules that dictate how the symbols and keywords in the language should be combined to form valid statements and expressions. It defines the structure and organization of the code;

Semantics: Describes the meaning of the statements and expressions written in the programming language. It clarifies how the computer should interpret and execute the code, ensuring that it behaves as intended.

The most fundamental language present in a computer is the *Machine Language* or *Instruction Set*. It is composed of atomic and (usually) simple instructions coded in binary, and associated with a certain architecture. All of this makes this language unsuitable—albeit not impossible—for human use. This factor propelled the creation of other types of languages to improve programmers efficiency, efficacy and overall quality-of-life [Proença, 2018].

The next step in the evolution of programming languages is the *Assembly Language*. It is almost equivalent to Machine Language, but instead of purely binary notation, it uses mnemonics to specify the desired operations and values. Although it is better handled by humans, it is still entirely dependent on a given architecture [Proença, 2018].

Finally there are *High Level Languages* that are more powerful and closer to natural human languages. They allow construction of programs for execution in any processor that has a compiler that can translate

source code into binary [Proença, 2018]. The High Level Languages simplify programming by abstracting away hardware-specific details, making the code more readable, while facilitating portability and maintenance. Thus, for most programmers, High-Level Languages are preferable due to their increase in productivity and ease of use.

2.1.2 Compilers and Interpreters

Since the processor only "understands" commands in Machine Language, it is necessary to convert programs written in languages from higher levels of abstraction to lower levels, until eventually reaching machine language. These translators or level converters are usually referred to as assemblers, compilers and interpreters.

Assemblers are programs that read files containing source code written in assembly language and convert them into binary machine language, i.e., assemble the instructions in a format suitable for the processor.

Compilers are programs that translate source code written in high-level languages to a lower level of abstraction. Most existing compilers already include the two steps of translation, compiling and assembling, translating from a high-level language directly into binary without the need for an assembler.

With an interpreter, high-level language instructions are analysed one by one, while machine language code is generated to execute it immediately, without saving it in a file. There is no translation of one program into another, but rather the analysis of a program followed by the generation and execution of the associated machine code [Proença, 2018, Aho et al., 2024].

2.2 Backus-Naur Form (BNF)

Backus-Naur Form (BNF) is a widely recognized notation used to define the syntax of programming languages. It is a type of metalanguage that describes which sequences of symbols constitute a syntactically valid program, allowing developers to formally specify the structure of a language. The purpose of **BNF** is not to dictate the meaning of those symbols (semantics) but rather to focus on their arrangement (syntax). This distinction makes **BNF** an invaluable tool in the design and understanding of programming languages, where it serves as a bridge between human-readable language definitions and machine processing (parsing). **BNF** has become a foundational tool in the field of computer science, enabling the specification of grammars for languages ranging from natural to domain-specific ones [McCracken and Reilly, 2003, Science Direct, 2024].

When working with **BNF**, it is crucial to recognize that this notation can serve two key purposes: it can function as documentation to clarify a language's syntax, and it can be directly applied to create parsers that validate program structure. For instance, parser generators such as Lex [Lesk and Schmidt, 1975] and Yacc [Johnson, 1975], Bison [Stallman, 2024], and Lark[1], often take a BNF-like grammar as input[2] and automatically build parsers, simplifying the process of language implementation [Brigham Young University, 2024, DeNero, 2024].

The core of a **BNF** grammar lies in production rules, which consist of a symbol—the building block of the language—followed by a sequence of symbols that might substitute it. There are two types of symbols: non-terminal and terminal. Non-terminals represent abstract elements of the language that can be further broken down into terminals or other non-terminals, while terminals are the base elements, such as specific strings or regular expressions. Each rule in **BNF** expresses how a symbol, known as a non-terminal, can be replaced by a sequence of other symbols, either terminal or non-terminal [Brigham Young University, 2024, Ada Computer Science, 2024].

2.2.1 Structure of BNF Rules

BNF rules are typically written in the following form:

```
digit: "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

In this rule, the non-terminal `digit` is defined as one of the terminal characters that range from `"0"` to `"9"`. The vertical bar (`|`) in this example represents a choice between alternatives, meaning that `digit` can expand into any one of the specified characters. This is the simplest form of a rule in **BNF**, where a non-terminal symbol is replaced by a terminal symbol.

BNF's power comes from its ability to represent context-free languages, which allows it to define more complex structures than regular expressions. Regular expressions, while useful for pattern matching, cannot handle languages that require nested or recursive elements, such as balanced parentheses. In contrast, **BNF** can express such structures given that it supports recursion in its rules. This recursive nature of **BNF** is crucial for expressing hierarchical structures, such as nested expressions or conditional statements, which are common in programming languages [Brigham Young University, 2024]. For example, consider the following rule for defining an unsigned integer, recalling the above-mentioned definition of `digit`:

```
unsigned_integer: digit | unsigned_integer digit
```

[1] More info on Lark can be seen at: https://lark-parser.readthedocs.io/en/stable/#welcome-to-lark-s-documentation.

[2] For the purposes of this document, every BNF example or excerpt is written in the form of Lark, the generator that implemented this Project's parser.

Here, `unsigned_integer` is recursively defined as either a single digit or itself followed by another `digit`. This allows for the representation of integers of any length, showcasing the recursive nature of **BNF** rules. Whenever a non-terminal appears on the right-hand side of a rule, it refers to another rule (or set of rules) that defines how it can be replaced. This example demonstrates how **BNF** allows complex structures to be defined in a recursive manner, reflecting the nested nature of expressions in programming languages.

The key features of **BNF** are:

Recursion: **BNF** allows for recursive definitions, meaning that a non-terminal can refer to itself in its definition. This is essential for expressing concepts like arithmetic expressions, where terms can be composed of smaller sub-terms.

Alternation: The | symbol in **BNF** allows for alternatives, meaning that a non-terminal can be defined in multiple ways. This is useful for defining different syntactic possibilities, such as different types of statements or expressions.

Simplicity and Generality: **BNF** is designed to be simple, allowing any syntax to be expressed in terms of terminal and non-terminal symbols. It provides a general framework that can be used to describe the syntax of a wide range of languages, from programming languages to markup languages.

The primary purpose of **BNF** is to provide a formal and unambiguous description of the syntax of a language. By defining the structure of language constructs using **BNF**, language designers ensure that the syntax is precise and interpretable by both humans and machines. In the context of compilers and interpreters, **BNF** plays a crucial role in the parsing phase, where the structure of the source code is analysed and transformed into a syntax tree based on the grammar defined in **BNF**. This syntax tree is then used to generate machine code or to interpret the program [ping Chu, 2024]. **BNF** grammars, therefore, form the foundation of many modern programming language tools, bridging the gap between formal language definitions and the practical parsing of code.

2.2.2 Abstract Syntax Tree

Abstract Syntax Tree (AST) is a kind of tree aimed at representing the abstract syntactic structure of the source code, that is, the nodes that correspond to its constructs or symbols [Zhang et al., 2019]. It is an intermediate data structure used in compilation or interpretation processes. The syntax tree not only reflects the structural information but also retains the attributes of the source program [ping Zhang and sheng Liu, 2013].

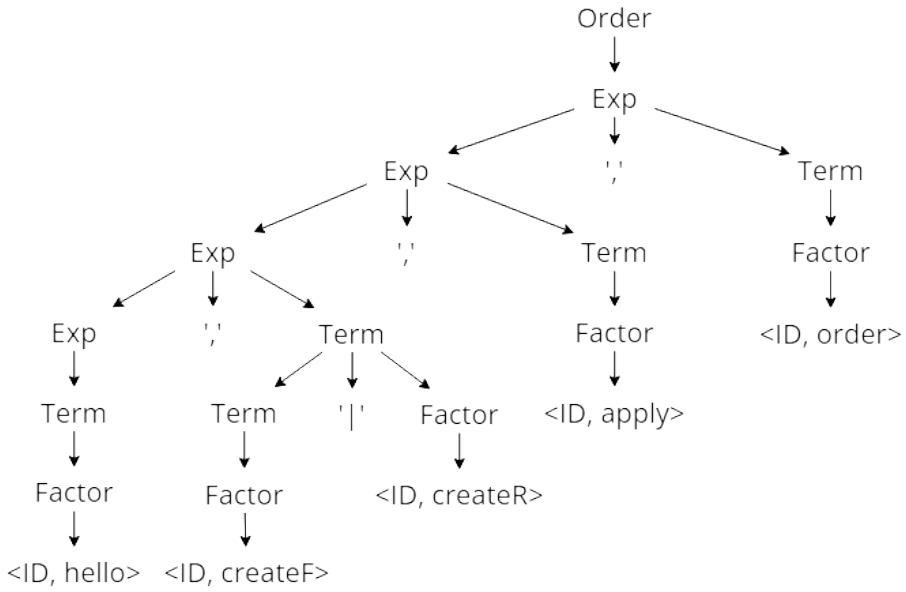


Figure 1: Abstract Syntax Tree generated from an order.

The **AST** in Figure 1 illustrates a potential structure generated by parsing the input phrase:

`[hello,createF|createR, apply,order]`

using the **BNF** definition provided in Listing 4.17.

2.2.3 Extended Backus-Naur Form (EBNF)

While **BNF** is a powerful tool, it can become verbose, particularly when dealing with optional elements or repeated patterns. This is where **Extended Backus-Naur Form (EBNF)** comes into play. **EBNF** adds several shorthand notations that allows certain grammar rules to be more concise and readable. Although **EBNF** doesn't increase the expressiveness of **BNF**, it introduces useful syntactic sugar that simplifies the definition of complex language constructs [ping Chu, 2024].

In **EBNF**, the concept of repetition can be expressed in a more compact form. Instead of recursive rules, symbols such as the asterisk (*****) for zero or more repetitions, and the plus sign (**+**) for one or more repetitions can be used. Similarly, the question mark (**?**) is used to sign that its preceding element is optional. These operators mirror their functionality from regular expressions, where they are also used to denote repetition and optionality [DeNero, 2024, Brigham Young University, 2024, ping Chu, 2024].

Take the following **EBNF** example, which defines an expression that can optionally start with a minus sign, followed by one or more digits, and optionally include a decimal point and further digits:

```
number: "-"? digit+ ("." digit+)?
digit: "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

This example illustrates the power of **EBNF** in condensing the structure of a grammar. The question

Table 1: Comparison between BNF and EBNF.

Feature	EBNF	BNF
Zero or more items	`items: item*`	`items: ϵ | item items`
One or more items	`items: item+`	`items: item | item items`
Optional item	`optitem: [item]` `optitem: item?`	`optitem: ϵ | item`

mark (?) after the minus sign indicates that it is optional, while `digit+` denotes that there must be one or more digits. In contrast, writing the same rules in pure **BNF** would require several more lines to account for the optionality and repetition, making the grammar harder to read (see Table 1 for a comparison).

By employing **EBNF**, developers can reduce the complexity of their grammar definitions, making them easier to maintain and extend. This compactness is particularly advantageous when dealing with large and intricate programming languages, where clarity and precision are paramount.

In conclusion, while **BNF** provides the essential framework for defining the syntax of programming languages, **EBNF** enhances this foundation with more readable and concise constructs. Both forms remain critical to the field of language design, serving as tools that help bridge the gap between theory and practice.

2.3 Computer Programming Education

> Learning to program is difficult and requires a lot of hard work, training, dedication and persistence [Araujo, 2022].

The problem of teaching and learning computer programming is inherently complex, usually leading students to a lack of motivation, when they encounter some of its many inherent difficulties According to researchers, the challenges arise at the inception of the learning process, where students struggle to understand and apply abstract programming concepts in order to design algorithms to solve concrete problems [Adu-Manu et al., 2013, Araujo, 2022, Gomes and Mendes, 2007, Piteira and Costa, 2013].

2.3.1 Factors that Interfere when Learning Computer Programming

Learning computer programming is influenced by a diverse set of factors, each of which plays a crucial role in shaping the overall educational experience. These factors can be grouped into distinct categories, allowing for a more systematic understanding of the challenges students face.

Attitudes and Competences

One of the main factors influencing the difficulties students face when learning programming is their attitude towards the problem and their skills to solve it, which leads to a large percentage of students struggling to solving problems [Araujo, 2022, Gomes and Mendes, 2007].

Students have a hard time understanding the problems they need to solve. This is due to interpretation difficulties and sometimes because they start solving the problem without knowledge of its context. Students also have trouble breaking down the problem and removing unnecessary details [Juárez-Ramírez et al., 2018, Lawan et al., 2019, Qian and Lehman, 2017, Tavares et al., 2015].

Relating new information to previous knowledge reveals itself as a challenge when drawing analogies with problems they have already solved. This can happen because they find it difficult to reapply knowledge constructed in previous problems or, when they do, that knowledge is misapplied [Cheah, 2020, Medeiros et al., 2019, Qian and Lehman, 2017, Araujo, 2022].

Students should also reflect on the problem and their solution. Frequent mistakes are that they either do not test the solution they have constructed, or when they do, they forget to test for the limit cases [Adu-Manu et al., 2013, Juárez-Ramírez et al., 2018, Lawan et al., 2019].

Moreover, when it comes to the overall attitude in approaching the problem, there is a lack of persistence, i.e. they give up on problems they cannot quickly find a solution to, usually in favour of looking for them on the Internet or asking a colleague rather than getting there on their own [Araujo, 2022, Gomes and Mendes, 2007, Cheah, 2020].

Finally, a lack of mathematical knowledge and logical reasoning, a reduced capacity for abstraction, and a lack of specific knowledge of programming are also factors pointed out in terms of the students' missing competences [Medeiros et al., 2019, Araujo, 2022, Bosse et al., 2019a,b, Cheah, 2020, Gomes and Mendes, 2007].

Study Methodology

In various academic disciplines, the efficiency with which students construct knowledge is significantly influenced by the study methodologies they choose.

As pointed by Tavares et al. [2015], any student can only learn to program by programming. In the context of programming education, competency acquisition requires substantial practical and intensive study. A predominant challenge in this domain lies in the insufficient dedication of students to practice, obstructing the acquisition of important programming skills [Medeiros et al., 2019, Gomes and Mendes, 2007].

Psychological Obstacles

A number of psychological factors have a negative impact on the process of learning to program computers. One such factor is low self-efficacy, characterised by reduced confidence or belief in the individual's ability to cope with challenging situations [Bandura et al., 1999, Araujo, 2022, Medeiros et al., 2019].

Low motivation is another diminishing trace that stems from the aforementioned factor, with students exhibiting a lack of persistence and a tendency to abandon problems for which immediate solutions are not readily apparent [Araujo, 2022, Cheah, 2020, Tavares et al., 2015].

Students typically begin their journey into programming at a time of complex life changes, coinciding in particular with the start of their higher education experience, a period marked by significant transitions such as the distance from their family, the place where they lived, and their friends [Gomes and Mendes, 2007, Araujo, 2022].

Nature of Programming

When it comes to the intricacies of programming, many factors can be of great impact on the students' performance. Programming languages usually have complex syntaxes, making it difficult for students to understand its logic, and consequently comprehend the language itself [Adu-Manu et al., 2013, Cheah, 2020, Araujo, 2022].

Programming also demands a highly developed capacity for dealing with abstraction, a skill that proves challenging for numerous students to master. Likewise, problem-solving skills, mathematical knowledge and logical reasoning are also indispensable and frequently difficult to obtain [Piteira and Costa, 2013, Medeiros et al., 2019, Araujo, 2022].

Teaching Methodology

Educators frequently prioritize teaching specific programming languages and their syntax, rather than fostering problem-solving skills through the application of those languages. This approach creates a dependency on the language's characteristics and commonalities, instead of establishing a strong foundation to be used with any programming language [Medeiros et al., 2019, Araujo, 2022, Cheah, 2020, Gomes and Mendes, 2007].

The principles of dynamic and abstract programming are typically taught through static resources such as slides, books, diagrams and verbal explanations, which do not promote an effective understanding of its dynamic nature [Adu-Manu et al., 2013, Bosse et al., 2019a, Gomes and Mendes, 2007].

Beyond that, teaching is not personalised to the needs of each student, and teaching strategies usually

do not accommodate all students' learning styles. Traditional teaching methods tend to enforce a uniform pace dictated by the teacher, neglecting variations in students' abilities and preferences [Bosse et al., 2019a, Araujo, 2022, Medeiros et al., 2019, Gomes and Mendes, 2007].

Finally, at times, the choice of a programming language for instructional purposes may not align optimally with its intrinsic characteristics, reducing efficiency and efficacy of the teaching-learning process [Cheah, 2020, Gomes and Mendes, 2007, Araujo, 2022].

2.4 Programming Education Tools

As previously discussed, teaching programming can be aided by various types of tools. Some of them help students to practise exercises autonomously, while others help them in the more abstract aspects, by visualising the execution of source code.

2.4.1 Automatic Assessment Tools

Automatic assessment tools have their focus on providing students with exercises and immediate feedback on their solutions, as is the case with Codeboard Meyer [2024]. This tool allows teachers to create exercises for their students, defining tests for automatic grading, and to analyse and inspect students' submissions. Another alternative is ACEGrader [dos Santos, 2023] which not only analyses the solution dynamically but also statically. Static analysis consists of checking the source code of the program itself, instead of analysing the solution. With this, it is able to check if the solution uses the specific concepts the problem requested, for example checking if the student used recursion.

In line with this theme, there are various tools that encourage student autonomy and efficiency by being more orientated towards code competitions. This type of solutions, such as uHunt [Halim, 2024], Kattis [Kattis, 2024] and Mooshak [Leal and Silva, 2003], focus on enabling learners to find problems, automatic judging of submitted programs and compiling of the result statistics. Some of them, such as Kattis, can also be used for class assessments, providing automatic code grading.

2.4.2 Source Code Animation Tools

Having another approach, program visualizers (source code animation tools) have a different goal from the previous tools. Their main objective is to help the student better visualise what is happening when the code is running. PythonTutor [Guo, 2013] is a solution that fits into this category. The student can visualise, step-by-step, what each line of code does, set the pace of the demonstration by navigating forwards and

backwards at their discretion, and view what is the current value of all variables and how they are stored.

2.4.3 Source Code Demonstration Tools

Another way animation is being used to aid learning programming is the animation of source code construction. In this field, Khan Academy [Khan Academy, 2024b] presents a solution within its exercises section. Their tool comprises a text editor, where the source code is presented, often with comments. The user is allowed to freely edit the code that is provided. It also has a separate area containing detailed instructions for the given exercise. Furthermore, it enables the execution of the source code, with a console that allows interaction as well as the display of any potential errors. The user is only then permitted to proceed if the code fulfills the desired requirements when executed.

2.5 Summary

In summary, teaching programming presents several challenges. On the educator's side, issues can arise from choosing an inappropriate programming language, focusing too much on the language itself rather than fostering computational thinking, or failing to tailor their teaching methods to meet the diverse needs of learners. From the student's perspective, difficulties may stem from a lack of foundational skills, such as weak mathematical or abstract thinking abilities, or trouble interpreting and breaking problems down into smaller, manageable tasks. Additionally, students often face challenges related to motivation and persistence, including giving up early when encountering initial obstacles or struggling to develop autonomy in their learning process.

To address some of the identified challenges, various programming education tools have been developed. Automatic assessment tools, for instance, focus on enhancing student autonomy, allowing learners to track their own progress and, often, enabling educators to monitor their development as well. Source code animation tools help students visualize how code executes, which improves their ability to grasp abstract concepts and promotes greater independence in learning. Additionally, there are tools that fall into the category of source code demonstration tools. These are designed to support students in writing code by providing partially completed templates that they must finish. This approach not only fosters autonomy and offers a more hands-on learning experience, but it may also aid students in learning the process of computational thinking by helping them understand how to break down problems into smaller parts, thus boosting motivation.

The tool developed in this project (Stepwise Source) belongs to the category of source code demon-

stration tools. Its purpose is to assist educators in better explaining the process of computational thinking and source code construction. Stepwise Source[3] is able to create presentations that include explanations, annotations and exercises, which students can later use for independent study, receiving instant feedback as they work. Additionally, it allows learners to progress at their own pace, promoting greater autonomy in the learning process.

[3] Stepwise Source is accessible at: http://stepwisesource.epl.di.uminho.pt/.

Chapter 3

Stepwise Source, Architecture

This section delves into the intricacies of Stepwise Source's architecture, elucidating its components and foundations. Figure 2 portrays the architecture of the application, showing its primary functionalities and data structures.

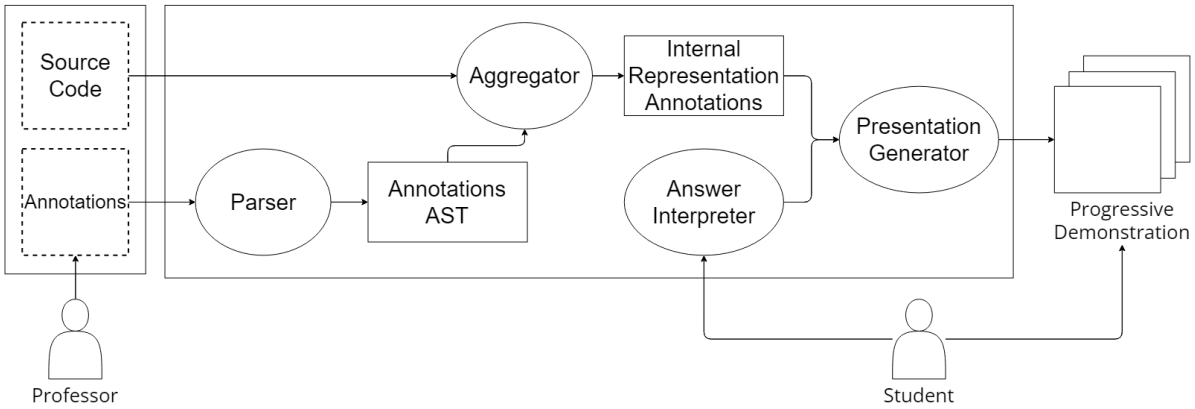


Figure 2: Stepwise Source's architecture.

Stepwise Source requires input from the educator in the form of a presentation written in Stepwise Language, a **DSL** that was designed specifically for this Project. In a presentation, the educator provides a sequence of slides that contains the *source code*[1], whose construction will be demonstrated and explained to the student, and *annotations* to this source code that will offer guidance to the student on how to carry out said construction.

3.1 Parser

In order to generate a presentation, the system processes the annotations through its *Parser*, thus generating **ASTs**. It is important to note that *the source code itself is not parsed.* Stepwise Source's Parser only deals with the annotations that are inserted into it, which inherently makes the whole system language ag-

[1] The term *source code* in this dissertation refers to the contents of a slide, not the source text of the presentation itself.

nostic. As long as the annotations are correctly written (syntactically speaking), and all special characters are escaped inside the source code, Stepwise Source will correctly generate the presentation.

In the scenario of this project, the **AST** serves as a guide for Stepwise Source, indicating which sections of the code are presented at each step, outlining the potential paths the presentation may follow.

3.2 Aggregator

Subsequently, the *Aggregator* component is responsible for generating the Internal Annotation Representation by merging the **AST** of the annotations with the source code. This source code, annotated by the educator, is enriched with specific commands that define how the presentation will unfold, detailing the number of steps, possible variations, and questions for the students. Through this process, the *Aggregator* constructs the internal representation of each potential step, ensuring that Stepwise Source can accurately follow the desired structure and interaction dynamics.

3.3 Answer Interpreter

The educator will not be the only actor providing information for the design of the presentation. In turn, the student will also provide answers to the exercises in the slides. This information will be processed by an *Answer Interpreter* so that it can provide automatic feedback on the response.

3.4 Presentation Generator

In the end, there is the *Presentation Generator*, a functionality responsible for displaying and navigating the final presentation. It relies on the results of the *Aggregator* to create a live demonstration based on the source code and the annotations added to it, as well as on the *Answer Interpreter* for the exercises' feedback.

3.5 Summary

In the domain of teaching computer programming, continuous evolution is fundamental to cultivate effective and engaging learning experiences. As elucidated in Chapter 2, there is still room for improvement in the current pedagogical scene. In response to this need, this chapter seeks to unveil a comprehensive solution designed to bridge certain gaps and improve the teaching of programming when it comes to source

code demonstrations.

The solution proposed in this document (Stepwise Source) consists in an application that aims to simulate the construction of source code in a progressive manner, similar—albeit not identical—to a live demonstration. The main difference relies on the fact that the student itself will determine its rhythm by controlling the progression or regression of steps according to his or her preferences.

Chapter 4

Stepwise Language

Chapter 3 outlined the structure of the system, highlighting the core components that drive the application. Among these, the Parser plays a pivotal role in processing input and enabling interactions within the system.

This chapter focuses on the formal foundation that underpins Stepwise Source's language processing capabilities. By applying the principles of formal grammars, it becomes possible to understand how specific rules and syntactical structures form the basis of Stepwise Language. These foundations guide the interpretation of commands and ensure that inputs are correctly processed by the system. From fundamental concepts of syntax definitions to the tailored constructs developed for this Project, this chapter will provide insight into the precise mechanisms behind the **DSL** created for and used by Stepwise Source.

4.1 Overview and Use

In contemporary educational contexts, the effective delivery of content is pivotal for facilitating student engagement and comprehension. This Project introduces a **DSL** designed to empower educators in teaching computational thinking and programming through the creation of interactive and visually appealing presentations. Central to this **DSL** are several key commands that provide educators with the tools to structure their lessons, style what is covered, integrate interactive components, and regulate the information flow. This section provides a detailed exploration of these commands, including their syntax, components, and practical applications.

4.1.1 Defining a Slide

At the core of any presentation crafted using this **DSL** lies the concept of a slide, an essential element that encapsulates both structure and content (see Listing 4.1 for its declaration syntax). The creation of a slide begins with the `\slide` command to define a new slide. Each slide is designated by a unique identifier (`ID`) that facilitates reference throughout the presentation, ensuring that content is organized

Listing 4.1: Syntax for a slide with an optional description.

```
\slide{<ID>}{<TITLE>}
\description{<DESCRIPTION>}
\begin
    <CONTENT>
\end
```

Listing 4.2: Example of a slide on variable declaration.

```
% This is a comment that spans to the end of the line.
\slide{Intro}{Introduction to Variables}
\description{Variables are used to store values that can be accessed
    ↪ later on.}
\begin
int main(int argc, char *argv[]) \{
    int x = 5;
    int y = x + 10;
\}
\end
```

and coherent.

The title of the slide, represented by the `TITLE` parameter, provides a succinct overview of the slide's focus. It sets the stage for the audience, offering a glimpse into the topic being addressed. Complementing this, an optional description can be included through the `\description` command, adding an extra layer of context and enriching the slide's message. This description elaborates on the content, helping the audience to better understand the significance of what they are about to learn.

To encapsulate the main content of the slide, the `\begin` and `\end` commands create a block where the primary material resides. This content can consist of various elements, including code snippets, explanatory annotation text, or other instructional materials relevant to the educational objectives. The ability to insert diverse types of content within this structure allows educators to tailor their presentations to suit different teaching styles and student needs.

As an running example for the rest of this section, a slide designed to introduce the concept of variable declaration in computer programming using the C language is presented in Listing 4.2. In this example, the slide (whose `ID` is `Intro`) begins with a clear title that indicates its purpose (`Introduction to Variables`), while the description succinctly explains the function of variables in programming. The block of content then demonstrates the concept with concrete examples, thereby supporting student comprehension.

Overall, the cohesive design of these elements allows for a structured flow of information, allowing

Listing 4.3: Syntax for the declaration of new styling commands.

```
1 \newreactive{<ID>}{<KEY>; <ATTRIBUTE>=<VALUE>; ...}
2 \newfixed{<ID>}{<KEY>; <ATTRIBUTE>=<VALUE>; ...}
```

educators to convey complex concepts in a clear and precise manner. By thoughtfully assembling each slide with the appropriate components, the **DSL** facilitates an effective learning environment that adapts to the needs of both teachers and students.

4.1.2 Styling

In addition to the foundational structure of slides, the visual presentation of content plays a vital role in effective communication. To enhance the clarity and appeal of presentations, the **DSL** offers two distinct commands that specify visual styles for annotations: `\newreactive` and `\newfixed`.

The `\newreactive` and `\newfixed` commands both have the essential function of defining custom styles which can be applied to the content's annotations, giving educators a versatile way of enhancing the visual appearance of their presentations. Both commands ensure that particular aspects of the presentation stand out and call attention to important information by establishing specific styles. These instructions contrast in how and when they deliver the styles to the audience, even though they both aim to improve content presentation.

The `\newfixed` command, on one hand, ensures that the applied styles are visible at all times, providing immediate and constant context to the audience. This approach is useful in scenarios where frequent reference to important information is necessary, offering a stable and predictable presentation experience.

On the other hand, the `\newreactive` command introduces a dynamic element by revealing styles only upon user interaction, such as hovering or clicking on an element. This feature reduces initial visual clutter, creating a more simplistic experience. It is particularly advantageous when a gradual reveal of information is desired, allowing educators to control the pacing of content delivery.

The syntax for defining these style commands is presented in Listing 4.3. In both declarations, `ID` serves as the identifier for each styling command definition, allowing it to be referenced throughout the presentation for consistent application. The `KEY` and `ATTRIBUTE` components consist of a set of CSS-like styling rules that can encompass various attributes such as text color, font weight, and other visual properties that contribute to the overall aesthetics of the content[1].

[1] The list of keys and attributes is presented in Subsection 4.1.4.

Listing 4.4: Example of use for the \newreactive and \newfixed commands.

```
1  \newreactive{highlight}{text-color: green; bold;}
2  \newfixed{removed}{text-color: red; strikethrough;}
3
4  \slide{Intro}{Introduction to Variables}
5  \description{Variables are used to store values that can be accessed
       ↪ later on.}
6  \begin
7  int main(int argc, char *argv[]) \{
8      \highlight{int x = 5;}{Initial value of x.}
9      \removed{int x = x + 10;}{This is wrong!}
10 \}
11 \end
```

Listing 4.5: Syntax for the declaration of macros.

```
1  \<COMMAND>{<ID>}{<PREDEFINED_COMMENT>}
```

To illustrate the application of these commands, Listing 4.4 adds them to the running example of Listing 4.2. In this example, the `highlight` style (applied on line 7 via the `\highlight` command) emphasizes key information using green and bold text, while the `removed` style (line 8, via `\removed`) can effectively signal to students which elements have been removed from prior discussions with a red colour and strike-through effect.

Such visual distinctions serve to guide students' attention to significant aspects of the material, akin to what a teacher would do while explaining a snippet of code in class.

4.1.3 Macros

To enhance consistency across presentations, the **DSL** provides robust support for predefined annotations in the form of *macros*. This feature allows educators to associate specific styling commands with a descriptive text, promoting both reusability and uniformity across multiple slides. By leveraging macros, educators can ensure that similar elements maintain the same visual appearance, which can significantly improve the coherence of the presentation.

Listing 4.5 presents the syntax for macro declarations. `COMMAND` refers to a styling command previously created with either **`\newreactive`** or **`\newfixed`**. `ID` serves as a name for the new macro that will apply to the annotation the same style as `\COMMAND`, including a `PREDEFINED_COMMENT` comment.

To illustrate the use of macros, consider the code in Listing 4.6, built on the running example of Listing 4.4. In this case, the declaration:

Listing 4.6: Example of a macro.

```
1  \newreactive{highlight}{text-color: green; bold;}
2  \newfixed{removed}{text-color: red; strikethrough;}
3
4  \removed{wrong}{This is wrong!}
5
6  \slide{Intro}{Introduction to Variables}
7  \description{Variables are used to store values that can be accessed
       ↪ later on.}
8  \begin
9  int main(int argc, char *argv[]) \{
10     int x = 5;
11     \wrong{int x = x + 10;}
12 \}
13 \end
```

`\removed{wrong}{This code was removed.}`

defines a new macro (`\wrong`) that will apply the same style as `\removed`, along with a comment (`This is wrong!`). Line 10 of Listing 4.6 shows how to use this macro. Differently from the commands created with `\newreactive` and `\newfixed`, `\wrong` does not allow for a comment as a second argument, given that it is predefined in its declaration (line 4).

Macros allow teachers to efficiently apply consistent styling and commentary across various parts of the presentation, thereby smoothing their workflow. In its core, these predefined annotations consist of shortcuts for reusing the same visual styling and comments in one command, and avoid repetition.

4.1.4 Styling Keys and Attributes

Stepwise Source **DSL** employs a CSS-like syntax for defining styles. This familiar structure assists educators who may have prior experience with web development, allowing them to focus on content creation rather than navigating complex formatting rules. Educators can apply colours through specific attributes such as `text-color`, `background-color`, and `border-color`. The colours themselves can be specified either by their common names (e.g. `red`, `green`, and `blue`) or by hexadecimal codes in the standard format `\#XXXXXX`. This flexibility enhances the customization options for presentations.

Table 2 provides a comprehensive overview of the **DSL**'s styling keys and attributes, illustrating how they correspond to standard CSS properties. By adopting a familiar syntax, the **DSL** not only enhances usability but also empowers educators to create visually appealing presentations that effectively convey their intended messages.

Table 2: Translation of DSL styling keys and attributes to CSS.

DSL	CSS Equivalent	Description
`text-color: red`	`color: red`	Changes the text color to red, drawing attention to highlighted text.
`background-color: blue`	`background-color: blue`	Sets the background color to blue, useful for emphasizing sections.
`border-color: #FA0C25`	`border-color: #FA0C25`	Defines the color of an element's border as a specific hex code, highlighting areas effectively.
`bold`	`font-weight: bold`	Highlights key terms or code for emphasis.
`strikethrough`	`text-decoration: line-through`	Visually indicates removed or outdated content.
`underline`	`text-decoration: underline`	Can be used to underscore important points or warnings.
`text-style`	`font-style`	Applies styling to text, with options for `italic` and `oblique` for emphasis or differentiation.
`text-size: 16`	`font-size: 16px`	Sets the font size of the text, allowing for adjustments in readability.
`text-align: center`	`text-align: center`	Aligns the text to the center of the element, useful for headings or important messages.

Listing 4.7: Syntax for exercises creation.

```
1 \exercise{/<REGEX>/}{<PROMPT>}
```

4.1.5 Interactive Exercises

A distinctive feature of this **DSL** is its ability to incorporate student engagement through the `\exercise` command, whose syntax is shown in Listing 4.7. This command is integral for creating interactive elements within the presentation, fostering active participation and self-assessment among students.

In this syntax, `REGEX` is a regular expression that automatically validates the input provided by the student, ensuring the accuracy of their responses. The `PROMPT` serves as the question or statement presented to the student, guiding their interaction and prompting them to reflect on the presentation.

In order to illustrate the application of interactive exercises, consider the example in Listing 4.8, which allows for immediate student feedback. In this instance, the command:

`\exercise{/[0-9]+/}{Complete the code...}`

prompts students to input a valid number, checking their response against the specified regular expression. This interactivity not only enhances student involvement but also enables them to self-assess their understanding in real-time.

Utilizing interactive exercises fosters an engaging learning environment, allowing students to test their comprehension and receive immediate feedback based on their inputs. Such features not only promote active learning but also encourage students to engage more deeply with the material, ultimately enriching

Listing 4.8: Example of an interactive exercise.

```
\slide{Intro}{Introduction to Variables}
\description{Variables are used to store values that can be accessed later on.}
\begin
int main(int argc, char *argv[]) \{
    int x = \exercise{/[0-9]+]/}{Complete the code by assigning an integer value to the variable 'x'.};
    int y = x + 10;
\}
\end
```

Listing 4.9: Syntax for the `\order` command.

```
\order{<SLIDE_LIST>}
```

the overall educational experience.

4.1.6 Presentation Flow

Controlling the flow of the presentation is essential to effectively convey information. The mandatory `\order` command enables this by allowing educators to dictate the sequence of slides, thereby creating dynamic, branching paths that are tailored to the audience's needs.

Listing 4.9 shows the syntax of the `\order` command. `SLIDE_LIST` represents the list of identifiers that dictates the order in which the slides will be presented. It is specified by a comma-separated list of slide IDs. Additionally, the *fork feature* allows educators to create branches in the presentation using the *pipe operator* (`|`), enabling conditional paths based on audience interaction or the flow of discussion.

To illustrate the functionality of the `\order` command, consider the example in Listing 4.10. It defines a presentation flow that starts with `Intro`, followed by `Variables`, where the presentation can diverge to either provide examples (`Examples`) or conduct a quiz (`Quiz`). Finally, the path merges and concludes the presentation with `Conclusion`. This flexibility allows for a tailored educational experience that adapts to the audience's needs, enhancing engagement and comprehension.

By utilizing the `\order` command, educators can effectively manage the presentation's flow, ensuring that content is delivered in a manner that is both engaging and responsive to the dynamics of the classroom environment.

Listing 4.10: Example of the \order command.

```
\slide{Intro}{Introduction to Variables}
...

\slide{Variables}{How to Declare Variables}
...

\slide{Examples}{Examples: Declaring Variables}
...

\slide{Quiz}{Quiz: Variable Declaration}
...

\slide{Conclusion}{The End}
...

\order{Intro, Variables, (Examples | Quiz), Conclusion}
```

Listing 4.11: Initial production rule.

```
start: commands
commands: command+ order
command: slide description? code_block
    | nreactive
    | nfixed
    | macro
```

4.2 Formal Definition

An **EBNF** grammar was written[2] in order to capture the syntax and structure required for Stepwise Language's specialized commands[3]. **EBNF**'s flexibility and precise notation make it ideal for defining the sequence, hierarchy and optionality inherent in this language's features. This section provides a detailed analysis of each production rule within the **EBNF**, starting with the overall structure of the language and proceeding through the components of each command, including slides, style definitions, macro declarations and presentation order.

The **EBNF** definition for Stepwise Language begins with the required `start` production rule, as seen in Listing 4.11. This initial rule is defined as a list of commands, including slide definitions (line 4), style (lines 4 and 5) and macro (line 6) declarations. After all the commands are set, the presentation order is defined.

[2] The source code for the whole project including the language can be found here: https://github.com/Yakari144/StepwiseSource

[3] More specifically, the grammar was written in Lark's dialect for **EBNF** in order to both *design and implement* the parser.

Listing 4.12: Slide declaration production rule.

```
1 slide: "\\" SLIDE "{" ID "}" "{" TEXT "}"
2 SLIDE: "diapositivo" | "slide"
3 ID: ("a".."z"|"A".."Z")("a".."z"|"A".."Z"|"_"|"0".."9")*
4 TEXT: /((\\\})|([^}]))+/
```

Listing 4.13: Description production rule.

```
1 description: "\\" DESCRIPTION "{" TEXT "}"
```

The slide command has three main components, represented by non-terminal symbols: `slide`, `description` (optional), and `code_block`. The slide non-terminal rule, shown in Listing 4.12, maps to the slide command with an identifier and a title. The slide keyword is defined in both Portuguese and English[4] (line 2), while the slide identifier follows typical identifier syntax: a letter followed by alphanumeric characters (line 3). The title uses the `TEXT` terminal symbol (line 4), which matches either escaped curly brackets (\}) or any character with the exception of a closing curly bracket.

The `description` command, shown in Listing 4.12, uses the same `TEXT` terminal for specifying text content. The `code_block` command, meanwhile, encapsulates `content` within `BEGIN` and `END` keywords, as shown in Listing 4.12.

Inside a slide's content, it is possible to include both source code (in any programming language language) and Stepwise Language commands. The EBNF rule for `content`, shown in Listing 4.14, includes three commands. The first is the `EXERCISE` command, which accepts a regular expression (`REGEX`, defined on line 6) and a text value (line 3) as arguments. The `NOT_END` commands (lines 4 and 5) so declared to ensure they do not accidentally signal the end of the slide, as would happen if their identifiers were `end`. Instead, by using the terminal `NOT_END`, these command definitions operate within the slide's content without triggering its conclusion.

[4] All keywords are defined in both languages. Identifier, by their own nature, can be written in any language that uses the English alphabet.

Listing 4.14: Production rule that corresponds to the content of a slide.

```
1 code_block: "\\" BEGIN content "\\" END
2 content: (entry | code)+
3 entry: "\\" EXERCISE "{" REGEX "}" "{" TEXT "}"
4     | "\\" NOT_END "{" content "}" "{" TEXT "}"
5     | "\\" NOT_END "{" content "}"
6 REGEX: /\/(\\\/|[^\/])+\//
7 code: CODE
8 CODE: /(\\\\|\\\}|\\\{|[^\\\}\{])+/
```

Listing 4.15: Production rules to declare new styling commands.

```
nreactive: "\\" NEWREACTIVE "{" ID "}" "{" mycss (";" mycss)* "}"
nfixed: "\\" NEWFIXED "{" ID "}" "{" mycss (";" mycss)* "}"
mycss: TXT_COLOR ":" COLOR
    | BG_COLOR ":" COLOR
    | BORDER_COLOR ":" COLOR
    | BOLD
    | ITALIC
    | UNDERLINE
    | STRIKETHROUGH
    | TXT_SIZE ":" NUM
    | TXT_ALIGN ":" ALIGN
```

Listing 4.16: Macro production rule.

```
macro: "\\" MACRO "{" ID "}" "{" TEXT "}"
MACRO: /(?!"newreactive"|"newfixed"|"order"|"novoreativo"|"novofixo"|
   ↪ "ordem")[a-z_A-Z][a-zA-Z_0-9]*/
```

Source code, defined by the `CODE` terminal, supports any kind of character and escaped symbols that are preceded with backslash (\). It is designed to allow for the use of any programming language inside Stepwise Source, even those that require characters that are reserved for Stepwise Language, such as open and close curly brackets, and backslash itself. The listings in Section 4.1 that contain a running example of slides with a source code written in C show that, in order to use curly brackets to open and close blocks of code, it is necessary to escape them with backslash (\{ and \}). This strategy, very common in other languages, even allows the use of Stepwise Language source code inside itself, as the content of a slide[5].

The `nreactive` and `nfixed` rules, detailed in Listing 4.15, share a similar syntax and only differ by their keyword. They define a unique identifier followed by a list of CSS-like styling rules, each separated by a semicolon.

The `macro` rule (Listing 4.16) is defined similarly to the other command rules, but without a specific keyword. The `MACRO` terminal captures identifiers that do not conflict with the other command keywords using a negative lookahead to avoid them. The remaining syntax aligns closely with that of the other commands.

The final command rule defined in the Stepwise Language's EBNF, `order`, is shown in Listing 4.17. This command uses a hierarchy of productions to handle command ordering and prioritization. The recursive structure self-manages precedence: parentheses and identifiers are evaluated first, followed by the

[5] In a similar fashion to programming languages *bootstrapping*.

Listing 4.17: Presentation order production rule.

```
order: "\\" ORDER "{" exp "}"
exp: term
    | exp ("," term)
term: factor
    | term ("|" factor)
factor: ID
    | "(" exp ")"
```

Listing 4.18: Ignoring whitespace.

```
%import common.WS
%ignore WS
```

pipe operator (`|`) and then commas, as the lowest priority.

In addition to commands, the EBNF also ignores whitespace, which is implemented as shown in Listing 4.18.

4.3 Summary

The Stepwise Source **DSL** presented in this chapter establishes a comprehensive framework for creating interactive, styled, and logically organized presentations. By integrating commands for defining slides, applying styles, incorporating interactive exercises, and managing the presentation flow, this **DSL** empowers educators with effective tools to enhance student's engagement and understanding. This innovative approach not only facilitates effective teaching but also transforms the learning experience into a more dynamic and participatory process, from the student's point-of-view.

Chapter 5

Stepwise Source, Development

A series of carefully thought-out technological decisions shaped the application as the transition from the conceptualization stages to the actual implementation occurred. This chapter outlines the rationale behind these choices, emphasizing how each component—from the parser to the frontend—integrates to form a seamless Web App.

5.1 The Parser and Interpreter

The selection of the programming languages and tools for building the parser was crucial in setting a strong foundation for the system. Python [Python, 2024] was chosen due to its flexibility, ease of use, and the wide range of libraries available. Among these libraries, Lark was selected for its ability to easily and rapidly construct a parser suited for Stepwise Language.

5.1.1 Interpreter Class

The implementation of Lark's Interpreter module in this project enables a customized parsing process tailored to the needs of the **DSL**. Beyond basic command interpretation, the Interpreter class traverses the **AST**, handling each node to build the core data structures required by the parser, while ensuring efficient communication with the database.

The Interpreter class includes key properties for managing data that is essential to Stepwise Source's functionality, including commands and slides declarations, the current slide's identifier and content, and the order structure, as shown in Listing 5.1

Beyond its internal properties, the Interpreter class also implements methods for each of the non-terminal symbols in the grammar. Beginning with the annotation style declarations, in order to convert the CSS-like syntax of the Stepwise Language into CSS (as explained in Subsection 4.1.4), the function in

Listing 5.1: Init function and class properties.

```
# Initializes the parser internal structures.
def __init__(self):
    # All style commands that have been declared using \newfixed,
    # \newreactive or macros.
    self.cmds = {}
    # All slides in the presentation. The slide structure is
    # implemented via a dictionary whose keys are: "code" for the
    # slide's translation to HTML, "exercises" for the exercises,
    # "annotations" for all annotations to the slide's content, and
    # "description" as the optional description of the slide.
    self.slides = {}
    # Current slide that is being parsed.
    self.current_slide = ""
    self.current_content = []
    # Presentation order.
    self.order_array = []
    self.order_ids = []
```

Listing 5.2[1] translates each property into CSS-compatible strings. This function iteratively processes each token, adding the appropriate CSS formatting.

The Interpreter also includes functions to create objects that get stored in the database. The function in Listing 5.3 gathers CSS data and an identifier to add a fixed command (declared in Stepwise Language via **\newfixed**) to the command object. This function operates similarly to the one handling reactive commands, as they process almost identical non-terminal symbols.

The presentation order command (stored in `self.order_ids` and `self.order_array`) is handled by the functions `order(...)` and `exp(...)` (see Listing 5.4). The former is responsible for storing the order of the presentation, while the latter parses the slide order expression declared inside the **\order** command parameter. Since the operator precedences were structurally defined in the **EBNF**[2], as mentioned in Section 4.2, the interpreter only has to collect and store the data. It checks each token, determining whether it represents a stopping point or a recursion case, structuring the data accordingly.

Every time a slide declaration is parsed, a new slide object is created and set as the current one (see Listing 5.5). The Interpreter then iterates through the slide's contents, populating the slide object with properties. The annotated source code inside the slide's content is translated into HTML in order to style it accordingly, with each exercise command converted to an input text box (see Listing 5.6).

With all input data parsed, collected and structured, the last step is to generate a unique identifier for the presentation—and check if it has not been already assigned—to finally store it within the appropriate

[1] Notice that this function is directly related to its grammar counterpart in Listing 4.15.

[2] These precedence rules are almost identical to usual arithmetic precedence definitions in regular programming languages.

Listing 5.2: Function that translates the DSL's styling keys and attributes to CSS.

```
def mycss(self, tree):
    f = 0, t = ""
    for e in tree.children:
        # CSS-like keys that are defined by a single token.
        if e.type == "BOLD":
            t += "font-weight : bold"
        elif e.type == "ITALIC":
            t += "font-style : italic"
        #
        # ...
        #
        # CSS-like attributes that are not defined by a single token.
        else:
            # Get the first token (the attribute)...
            if not f:
                t += e.value
                f = 1
            # ...and add the second one (the value).
            else:
                t += " : " + e.value
    return t
```

Listing 5.3: Function that parses fixed style command declarations. Its reactive command counterpart is almost identical.

```
def nfixed(self, tree):
    key = ""
    css_properties = []
    for e in tree.children:
        # CSS-like attributes and keys.
        if type(e) == Tree:
            if e.data == "mycss":
                css_properties.append(self.visit(e))
        # The command ID.
        else:
            if e.type == "ID":
                key = e.value
    self.cmds[key] = {"category": "command", "type": "fixed", "style"
    ↪ : css_properties}
```

Listing 5.4: Functions that parse the presentation order command.

```
def order(self, tree):
    # The list of IDs for the presentation order ("order_ids") is
    # filled within the "factor" method of the Interpreter class.
    r = [], self.order_ids = []
    for t in tree.children:
        if type(t) == Tree:
            r.extend(self.visit(t))
    self.order_array = r

def exp(self, tree):
    t = None, is_seq = False, is_fork = False
    for e in tree.children:
        if type(e) == Tree:
            if e.data == "term":
                term = self.visit(e)
                t = term[0], is_fork = term[1]
            elif e.data == "exp":
                is_seq = True
                exp = self.visit(e)
    if is_fork:
        t = [t]
    if is_seq:
        exp.extend(t)
        return exp
    else:
        return t
```

Listing 5.5: Slide structure initialization function.

```
def slide(self, tree):
    #
    # ...
    #
    self.current_slide = slide_ID
    # The slide's content must be translated to HTML. The following
    # line starts the process by opening the <span> tag that will
    # encapsulate it.
    self.slides[self.current_slide]["code"] = "<span class='code' id
    ↪ ='" + slide_ID + "'>"
    self.slides[self.current_slide]["annotations"] = []
    self.slides[self.current_slide]["description"] = ""
```

Listing 5.6: Slide's content processing functions.

```
def exercise(self):
    #
    # ...
    #
    # Add the exercise to the slide's structure.
    self.slides[self.current_slide]["exercises"]["#" + str(i)] = {"
    ↪ regex": reg, "text": text}
    # Translate the exercise to a HTML input text box.
    r = "<input type='text' class='exercise' regex='" + reg + "'
    ↪ title='#" + str(i) + "'></input>"
    return r

def not_end(self):
    #
    # ...
    #
    # Any command used to annotate the slide's content is translated
    ↪ into a <span> HTML tag.
    self.slides[self.current_slide]["annotations"].append(name)
    return "<span command='" + name + "' class='" + self.cmds[name]['
    ↪ command'] + "'>" + cont + "</span>"
```

tables in the database.

5.2 Database

Given the importance of managing and retrieving complex data, choosing a suitable database was essential. *MongoDB* [MongoDB, 2024], a document-based database, was selected for its flexibility in handling the structured data generated by the parser. Its capacity to store JSON-like documents made it highly compatible with the system's data structures and Python's internal libraries and functionalities, enabling seamless integration and optimal storage for content and configurations.

To effectively organize the parsed data structures within MongoDB, the database system is divided into three collections: *presentations*, *annotations*, and *orders*. Each collection plays a specific role in managing the structured information resulting from Stepwise Language's parsing.

The *presentations* table stores presentations that contain an identifier, a name, the original text input used to generate the presentation, and a list of processed slides. Each slide within a presentation includes its identifier, title, description, generated HTML code, a list of annotation identifiers, and a list of exercises, where each exercise entry specifies its identifier, assessment regular expression, and question.

In the *annotations* table, each entry represents the annotations linked to a specific presentation. This

Listing 5.7: Order Syntax Example.

```
[Beginning, [[Tips], [Exercise]], Syntax, Ending]
```

table includes the presentation identifier it refers to, alongside a list of annotations. Each annotation entry has an identifier and a category attribute, classifying it as either a `"command"` or `"macro"`. Entries in the `"command"` category also include a type and a list of styling attributes, while those in the `"macro"` category contain the command that created it and the associated text.

The final table, *orders*, holds the structural order of each presentation. Each order entry includes the presentation identifier and an order string that defines the slide arrangement. The syntax, shown in Listing 5.7, organizes slides by identifier. Nested lists in the order string represent forks, with each path in a fork following the same syntax as the main order. This organization allows each path to be processed as an independent presentation structure.

5.3 Backend

The backend serves as the intermediary layer, managing communication between the frontend, the parser, and the database. Built with *JavaScript* [Eich, 1995] and *Express.js* [Express, 2024], the backend efficiently handles requests and processes data, whether it involves retrieving stored content from MongoDB or sending commands to be processed by the Python-based parser. This architecture enables the frontend to dynamically interact with the system, providing the necessary content to users without delays or inconsistencies.

5.3.1 Connection with the Database

As mentioned in Section 5.2, Stepwise Source's database is powered by MongoDB. To enable interaction between the Express.js backend and MongoDB, the *Mongoose* [Mongoose, 2024] package was utilized. Mongoose serves as an Object Data Modelling (ODM) library, simplifying the process of accessing and manipulating MongoDB data through schema-based models.

The interaction with the data stored in MongoDB requires each collection to correspond to a model that defines its structure. For instance, the model for the Annotations collection includes its schema, specifying the data fields, their requirement and validation. In this case, each slide's annotations are modeled as shown in Listing 5.8.

Listing 5.8: Basic structure of the Annotation model.

```
var annotationSchema = new mongoose.Schema({
    idDemo: String,
    annotations: [
        idAnnotation: {
            type: String,
            required: true
        },
        category: {
            type: String,
            required: true
        },
        command: {
            type: String,
            validate: {
                validator: function(v) {
                    return this.category == 'macro'
                },
                message: 'This is a macro, it must have a command.'
            },
        },
        text: { /* Similar to "command" above */ },
        style: {
            type: [String],
            validate: {
                validator: function(v) {
                    return this.category == 'command'
                },
                message: 'This is a command, it must have a style.'
            },
        },
        type: { /* Similar to "style" above */ }
    ]
})
```

Common Attributes

Each annotation can have two distinct formats, but there are common attributes to all of them. For example, both `idAnnotation` and `category` are defined as required strings. Listing 5.8 shows the definition of `idAnnotation`, which follows a similar pattern for the `category` field.

Conditional Validation for Distinct Attributes

Certain attributes vary based on the type of their annotations. To handle this, a validation function was implemented to ensure that specific attributes only exist where appropriate. For instance, entries with the `macro` category require `command` and `text` attributes, while those with the `command` category require `style` and `type` attributes. Listing 5.8 provides an example of the `style` field definition, which includes a custom validation to check its command category.

Nested Schemas

In addition to simple data types like strings and numbers, certain attributes may require more complex types, such as arrays or nested objects. Mongoose allows for the definition of these attributes through nested schemas. For instance, the `exercises` field in the *Slide* model is an array of `exerciseSchema` objects. This modular approach keeps the codebase organized and readable. Listing 5.9 shows how the `exercises` field is defined within the `slideSchema`.

The corresponding `exerciseSchema` is then defined separately, as demonstrated in Listing 5.10, ensuring that each exercise follows a consistent structure containing its own unique attributes, such as `idExercise`, `regex`, and `text`.

Exporting the Schema

Once the schema for the `annotations` collection is fully defined, it can be exported for use across the application using `mongoose.model`, as shown in Listing 5.11.

By using Mongoose in this manner, the project ensures clear, modular, and maintainable schema definitions, with seamless interaction between the backend and the database.

5.3.2 Connection with the Web Application

In addition to its role in database interaction, the backend also manages communication with the web application. This is achieved by exposing predefined API routes that the frontend can use to send or request

Listing 5.9: Definition of `exercises` field in the `slideSchema`.

```
var slideSchema = new mongoose.Schema({
    idSlide: {
        type: String,
        required: true
    },
    code: {
        type: String,
        required: true
    },
    text: {
        type: String,
        required: true
    },
    description: {
        type: String,
        required: true
    },
    annotations: {
        type: [String],
        required: true
    },
    exercises: {
        type: [exerciseSchema],
        required: false
    }
})
```

Listing 5.10: Definition of `exerciseSchema`.

```
var exerciseSchema = new mongoose.Schema({
    idExercise: {
        type: String,
        required: true
    },
    regex: {
        type: String,
        required: true
    },
    text: {
        type: String,
        required: true
    }
}, { versionKey: false })
```

Listing 5.11: Exporting the `annotationSchema` model.

```
module.exports = mongoose.model('annotation', annotationSchema);
```

Listing 5.12: Function that lists all the existing presentations.

```
module.exports.list = () => {
    return Demo.find({}, {idDemo: 1, demoName: 1})
        .then(docs => {
                return docs
        })
        .catch(erro => {
                return erro
        })
}
```

specific data. These routes enable the client to make HTTP requests to the server, which processes them and returns the appropriate responses.

When the client-side application needs to retrieve information from the database, it initiates a `GET` request to the backend. The server then interacts with the database, fetches the required records, processes them as needed, and responds to the frontend with the appropriate data.

Several routes are available for various tasks, such as retrieving all presentations stored in the database or obtaining details for a specific one. For instance, when the frontend requests a list of all presentations, it sends a `GET` request to the `/api/presentations` route in the backend. This request triggers the execution of the `list` function that can be seen in Listing 5.12. The function is defined in the backend and utilizes the Mongoose's `find(...)` method to query the database for all presentations. However, instead of retrieving all fields for each presentation, the query is restricted to only return two specific fields: `idDemo` and `demoName`. This ensures that only the necessary information is sent back to the frontend, optimizing performance and reducing payload size. Once the query completes, the list of presentations is displayed to the user. If an error occurs during the database query, the function catches the error and sends it back as the response, allowing the frontend to handle the issue accordingly.

To fetch a specific presentation, the frontend includes the presentation's `ID` as a parameter in the URL (e.g., `/api/:idDemo`). The server then queries the database for the corresponding presentation. In cases where only partial data is required, such as when the frontend requests an exercise's input text to enable user interaction, the server ensures that only the relevant information is returned, keeping the response efficient.

Apart from `GET` requests, the backend also handles `POST` requests, which allow the frontend to send new data to be stored in the database. For example, when a user creates a new presentation, the frontend submits a `POST` request containing essential details such as the presentation's name and source text. The backend forwards this information to the parser, which validates the syntax and generates the

necessary structured data for the presentation. Once the parser is done, the new presentation is saved in the database, and its generated `ID` is returned to the frontend. Should any errors occur, the server sends an error message back to the client, allowing the user to address the issue.

A similar process applies when a user edits an existing presentation. The key difference is that the frontend includes the presentation's `ID` in the request, enabling the backend to update the correct record. Beyond that, the workflow is the same as when a new presentation is created.

Finally, the backend supports a third type of `POST` request when users submit feedback regarding Stepwise Source in the About page. This feedback is relayed to the interested individuals via email, by leveraging the *Mailgun API* [Mailgun, 2024] service.

5.4 Web Application

The frontend was developed with *Create React App* [Create React App, 2024] due to its simplicity and efficiency. React's component-based architecture made it easier to manage dynamic content and consistency throughout the application. This framework allows the Web App to render and update the learning materials in real time, providing an interactive experience for users.

The web application is divided into six pages: Home, About, Documentation, Presentation List, Presentation Editor, and Presentation Navigator.

The *Home* page serves as the primary point of contact between users and Stepwise Source. The home page is designed to leave a strong first impression. Since this is the landing page, the goal was to create a clean, intuitive interface that clearly communicates the App's purpose and functionality.

The *About* page provides users with a comprehensive overview of Stepwise Source, covering its purpose, the team behind its development, and options for user feedback. Implemented as a React component, this feedback form communicates with the backend upon submission, triggering a notification to the development team, as the Subsection 5.3.2 previously described. This page was designed to communicate the core objectives of the project and offer an accessible channel for users to connect with the development team.

The *Documentation* page assists users with understanding the structure and usage of the **DSL** and the Web App. This page is divided into three primary sections: an introductory landing area, the main documentation content, and a practical example. The practical example is the tutorial's source code being dynamically retrieved from the backend rather than hardcoded in the frontend, ensuring it updates automatically whenever the tutorial presentation is modified.

Listing 5.13: Connection between the Presentation List page and both the Editor and Navigator.

```
// "Go To" button.
const navigate = useNavigate();
navigate('/demo/${idDemo}');

// "Edit" button.
const navigate = useNavigate();
navigate('/edit/${idDemo}');
```

As the hub of management for the App, the *Presentation List* page allows users to access and manage all presentations created within the platform. This page displays a table with two columns: the first lists the names of each presentation, guiding users to identify relevant content, while the second provides action buttons for each presentation. The frontend sends a `GET` request to the backend to retrieve the necessary data—a list of all available presentations and their respective names. This request process is detailed in the backend section (see Subsection 5.3.2). Each presentation is then displayed as a row within the presentations table. The redirection, both for the "Go To" and the "Edit" buttons, is implemented using the React's `navigate(...)` function, as illustrated in Listing 5.13.

The *Presentation Editor* page provides users with an editor component and three basic operations to write or edit presentations. The operations were implemented through handler functions as shown in Listing 5.14. This is the page that educators would use to create or edit their presentations.

Finally, the *Presentation Navigator* page renders the presentations, and allows students to navigate the source code demonstration using links and a *navigation graph*. The styling from the **DSL** is translated into HTML and CSS in order to show comments and highlights in the correct visual format (as explained in Section 5.1). Styled code portions are wrapped in `span` tags with specific classes, generated by the parser. Listing 5.15 shows the code that applies the styling for each slide's content. Here, style data retrieved from the backend is dynamically assigned to specific classes that are used in HTML elements within the slide content. These classes are linked to each element to apply the appropriate styles, making it possible to visually distinguish styled text within each slide. The navigation graph is rendered (see Listing 5.16) directly based on the order of presentation saved in the database (previously shown in Listing 5.7).

Given that the overall look and structure of the Web App has been developed directly through React and its mechanisms, the general implementation follows its foundations, as presented, in resumed form, in Listing 5.17.

Listing 5.14: Operations in the Editor page.

```
// Upload button handlers.
const fileInputRef = useRef(null);
const handleUploadFile = () => {
    fileInputRef.current.click();
};
const handleFileChange = (event) => {
  const selectedFile = event.target.files[0];
  if (selectedFile) {
    const reader = new FileReader();
    reader.onload = (e) => {setText(e.target.result);};
    reader.onerror = (e) =>{console.error("Error reading file:", e)
    ↪ ;};
    reader.readAsText(selectedFile);
  }
};

// Download button handlers.
const handleDownload = () => {
    const blob = new Blob([text], {type: 'text/plain'});
    const url = URL.createObjectURL(blob);
    downloadLinkRef.current.href = url;
    downloadLinkRef.current.download = 'presentation.txt';
    downloadLinkRef.current.click();
    //
    // ...
    //
};
```

Listing 5.15: Component that applies slide styling into HTML.

```
const Styledhmtl = styled.div`
    ${Object.keys(style).map((key) => `
      & .${key} {
        ${Object.keys(style[key]).map((property) => `
          ${property}: ${style[key][property]};
        `).join(' ')}
      }
    `).join(' ')}
    `
```

Listing 5.16: Navigation graph rendering.

```
function handleFork(array, x, y, properties) {
    let nodesRendered = [];
    for (let j = 0; j < array.length; j++) {
        //
        // ...
        //
        // Handle each of the paths recursively.
        const new_nodes= calculateNodes(array[j], new_x, new_y, zoom)
    ↪ ;
        nodesRendered = nodesRendered.concat(new_nodes);
    }
    return nodesRendered;
  }

const calculateNodes = (nodes, x, y, zoom) => {
    let nodesRendered = [];
    for(let i = 0; i < nodes.length; i++) {
        if (Array.isArray(nodes[i])) {
            // Handle forks.
            const nds = handleFork(nodes[i], x, y, properties);
            nodesRendered = nodesRendered.concat(nds);
            //
            // ...
            //
        } else {
            // Handle nodes.
            const node = nodes[i];
            nodesRendered.push({id: node, x: x, y: y});
            x += properties.varX;
        }
    }
    return nodesRendered;
};
```

Listing 5.17: Overview of Stepwise Source's implementation.

```
function App() {
  return (
    <div>
      <BrowserRouter>
        <Routes>
          <Route path="/demo/:id" element={<Demo />} />
          <Route path="/create" element={<NewDemo />} />
          <Route path="/edit/:idDemo" element={<NewDemo />} />
          <Route path="/error" element={<ErrorPage />} />
          <Route path="/documentation" element={<Documentation />} />
          <Route path="/tutorial" element={<Demo id="TutorialID"/>}/>
          <Route path="/presentations" element={<Presentations />} />
          <Route path="/about" element={<About />} />
          <Route path="*" element={<Home />} />
        </Routes>
      </BrowserRouter>
    </div>
  );
}
```

5.5 Deployment

Effective deployment is crucial for ensuring that the Stepwise Source application operates smoothly on a dedicated server. To achieve this, Docker [Docker, 2024] was employed to create isolated containers for each component of the system. This approach ensures that the database, backend, and frontend run independently and without conflicts, enhancing the reliability of the application.

This section discusses the deployment process in detail, covering the specific Dockerfiles used for both the backend and frontend components, as well as the orchestration of these containers using Docker Compose. This setup not only streamlines the deployment process but also simplifies future updates, ensuring that the application remains scalable and adaptable to evolving requirements.

5.5.1 Backend Dockerfile

Built upon the latest Ubuntu image, this Dockerfile establishes a comprehensive set of steps to ensure that all necessary packages and configurations are in place for seamless execution.

The process begins with updating the package list and installing essential tools, including *curl* [curl, 2024]. This utility is required to install the specific version of *Node.js* [Node.js, 2024] and *Python*, as they are the primary programming languages utilized in the backend development. While curl is not directly

invoked by the backend application itself, it is instrumental in setting up an environment to run the backend services.

Following this, the application directory is created, and the files detailing the dependencies for Python and Node.js are copied into it. They are then installed within the container to ensure that the backend has all the necessary libraries and frameworks to operate correctly.

After the dependencies are in place, the remainder of the backend application is copied to the working directory. At this stage, it is important to note that the tutorial presentation is added to the database at the moment the backend container is deployed. Therefore, a script designed to check whether the database is operational before the backend starts is included and copied into the working directory.

With all components in place, a port is exposed for incoming connections. The final step involves starting the backend application, which includes executing a waiting script to ensure the database is ready, creating the tutorial presentation, and launching the Express.js application using the command `npm start`.

5.5.2 Frontend Dockerfile

The Frontend Dockerfile is organized into two main stages, enabling efficient building and serving of the React application.

The process begins with the official *Node.js* (version 16) image, which serves as the build environment for the React app. A working directory is established, and the files that define the application's dependencies are copied over, similar to the approach taken in the backend Dockerfile. Subsequently, the necessary Node.js dependencies are installed.

Once the dependencies are in place, the remainder of the project files are copied into the container, followed by the execution of the command `npm run build` to compile the React application.

In the second stage, the lightweight *nginx:alpine* image is employed to serve the application. The compiled files from the first stage are transferred into the default directory of *nginx* [nginx, 2024], while a custom nginx configuration file is also copied into the container to configure the server settings.

To allow external access to the application, a specific port is exposed. Finally, the Dockerfile concludes by running nginx as a background service, ensuring that the container remains active as long as nginx is operational.

5.5.3 Docker Compose

The Docker Compose file orchestrates the deployment of the application's services: *MongoDB*, *backend*, and *frontend*. It facilitates communication between the components while ensuring they remain isolated within their own containers.

MongoDB is defined first, which utilizes the latest version of its image. A specific container name is assigned to this service, enhancing identification. A volume is created, to ensure data persistence, and an external port is exposed, mapping it to the default MongoDB port inside the container. This configuration allows for seamless communication with the database.

Following this, the backend service is defined. Its container name is specified, along with the directory from which it will be built. Given its dependency on the MongoDB service, this relationship is clearly articulated within the configuration. An external port is again exposed, mapped to the internal port used by the backend application, facilitating access to its functionality.

Lastly, the frontend service is configured, with its container name and build context defined. Similar to the previous services, an external port is exposed and connected to the internal port which is utilized by NGINX this time, enabling the serving of the built React application.

In addition to the service definitions, the Docker Compose file concludes with declarations for named volumes and networks. The Mongo data volume ensures persistent storage for the database, while a network is defined using the bridge driver. This setup allows for efficient inter-service communication, enhancing the overall performance and functionality of the application.

5.6 Summary

This chapter detailed the key technological decisions that brought the proposed approach to life, demonstrating how each component works in conjunction with the others. From the parsing of commands to the frontend's user interface, every decision was aimed at creating a tool that not only functions effectively but can also be expanded upon in future iterations.

Chapter 6

Stepwise Source, Overview

This section provides an overview of each main page in the web app, detailing their functionalities and layout to illustrate how they support the user experience.

6.1 Home Page

When users visit the website, they are greeted by this landing page, which features a title, header, and a couple of calls to action.

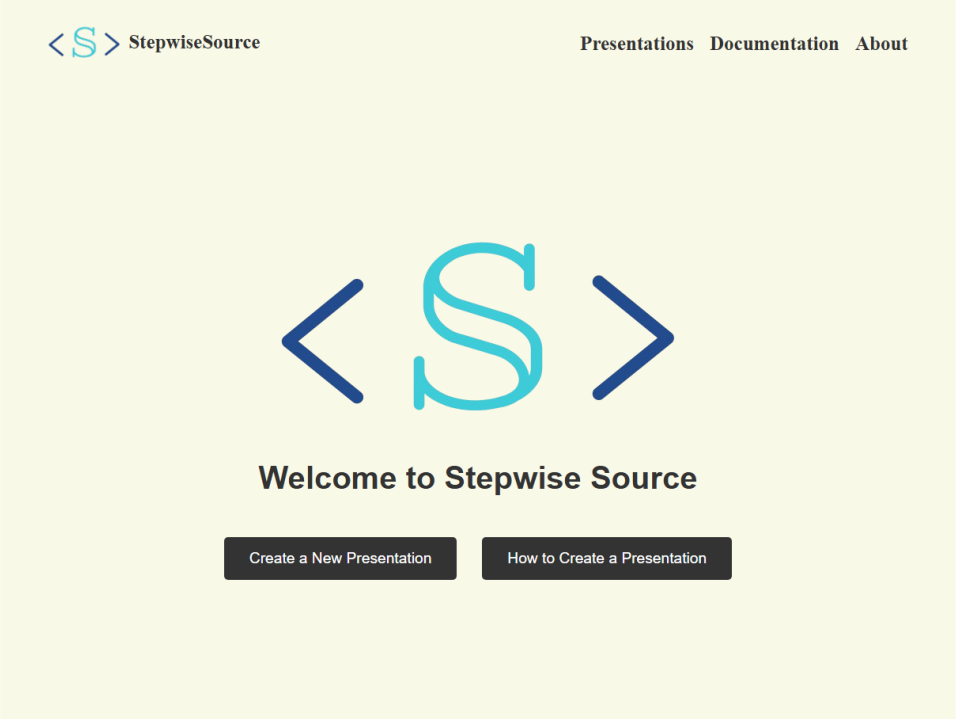


Figure 3: Home page.

At the centre of the page (see Figure 3), which naturally draws the visitor's attention, a welcoming message is prominently displayed alongside buttons that link to the core features of the platform: creating

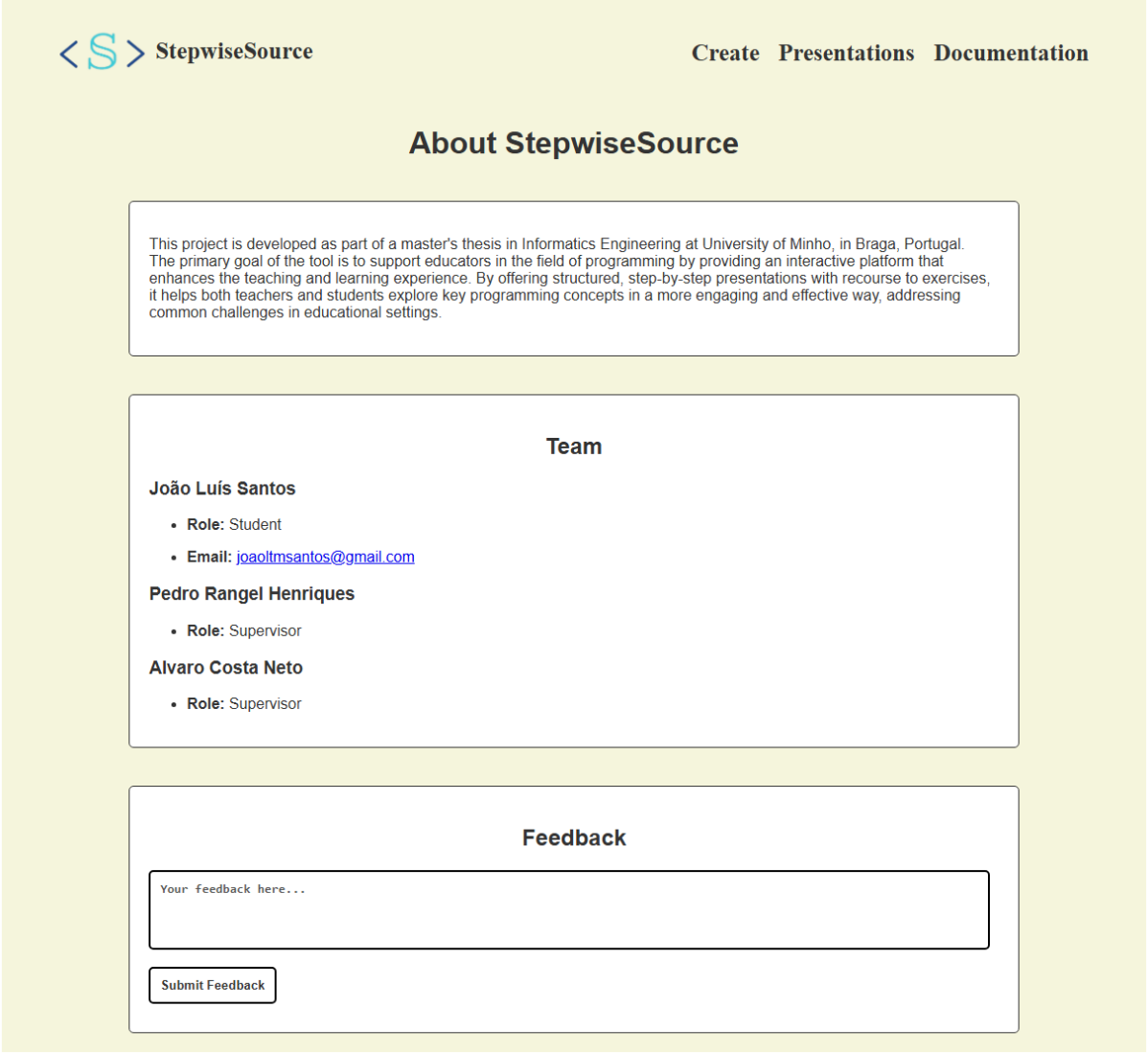


Figure 4: About page.

a presentation and watching one–in this case a tutorial. This design choice ensures that users are greeted warmly and guided toward exploring the main functionalities of Stepwise Source right from the start.

Additionally, the home page includes a header that provides navigation links to all other sections of the website, allowing users to easily access various features and flow freely through the app. This header remains consistent across all pages of the site, dynamically adjusting the available links based on the current page the user is viewing.

6.2 About Page

To clarify the context and purpose of the project, the team of developers and allow for user feedback about the application, this page was created.

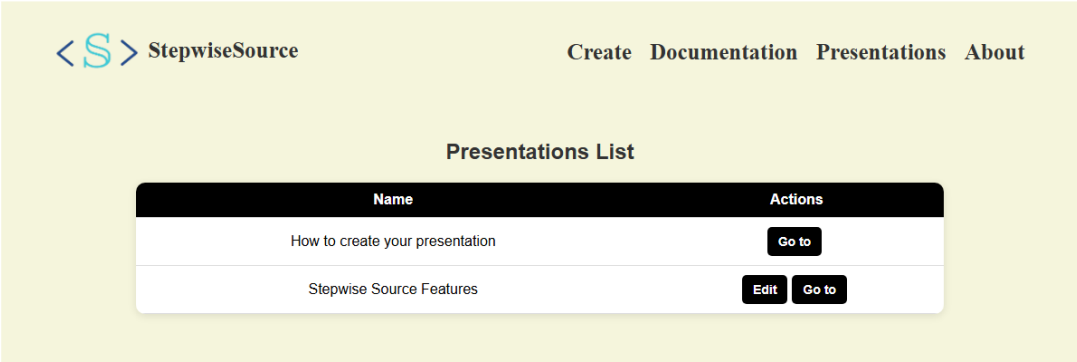


Figure 5: Presentation List page.

At this page, users are introduced to the platform, followed by a section explaining that Stepwise Source is a project developed as part of this master's thesis. This section offers a concise description of the tool's primary goals, helping users understand its intended impact and functionality (see Figure 4).

Below this, a section displays the names and roles for each team member involved in Stepwise Source's development. An email address is provided for direct communication with one of the developers, fostering a connection between users and the creators.

The final section invites users to submit feedback. A text box allows users to input any feedback they wish to share. Upon successful submission, users are greeted with a thank-you message, which is displayed for five seconds before the text box is restored, ready for additional feedback if needed.

6.3 Presentations List Page

The presentations page was designed to showcase all presentations created on the platform, providing an alternative way to access them beyond simply sharing URLs.

As shown in Figure 5, the first row is reserved for the tutorial presentation. Since this tutorial is not intended user modification, only a "Go To" button is available, preventing any edits. This "Go To" button serves as a redirect, guiding users to the respective presentation page.

For all other presentations, besides "Go To", an additional "Edit" button is available, allowing users to modify the presentation content.

6.4 Documentation Page

The documentation page serves as a comprehensive resource for users to understand and navigate the language's syntax and functionality. Here, users can find clear explanations, examples, and guidance on

effectively using the language, making it an essential reference for troubleshooting and mastering the tool.

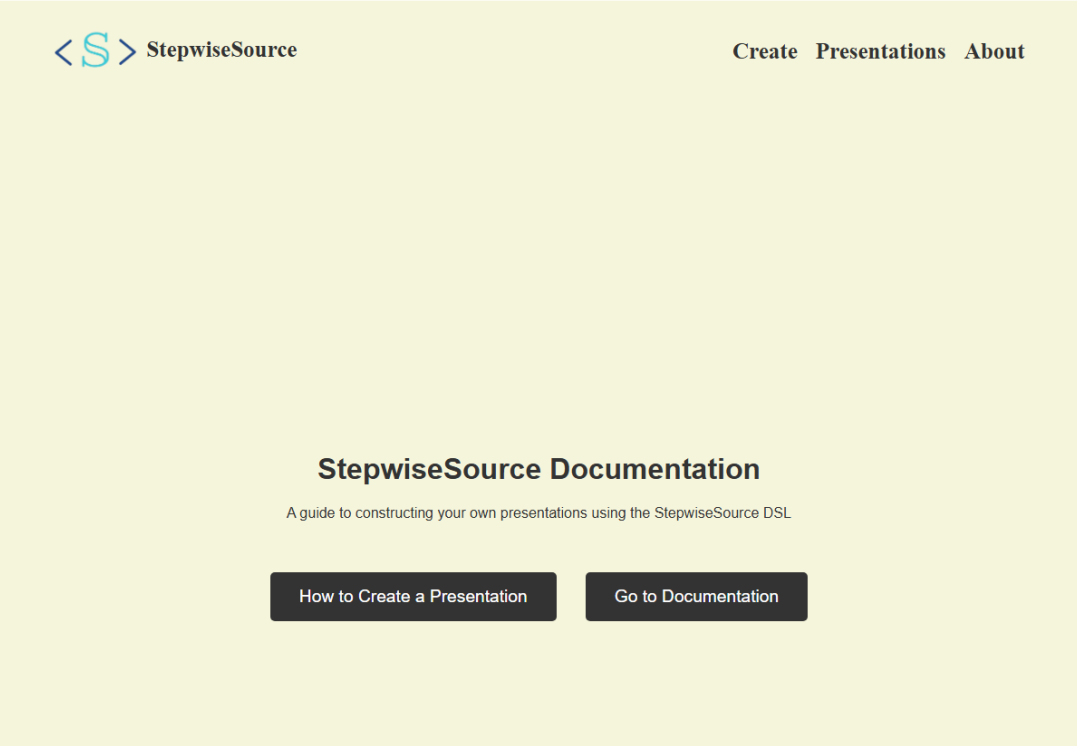
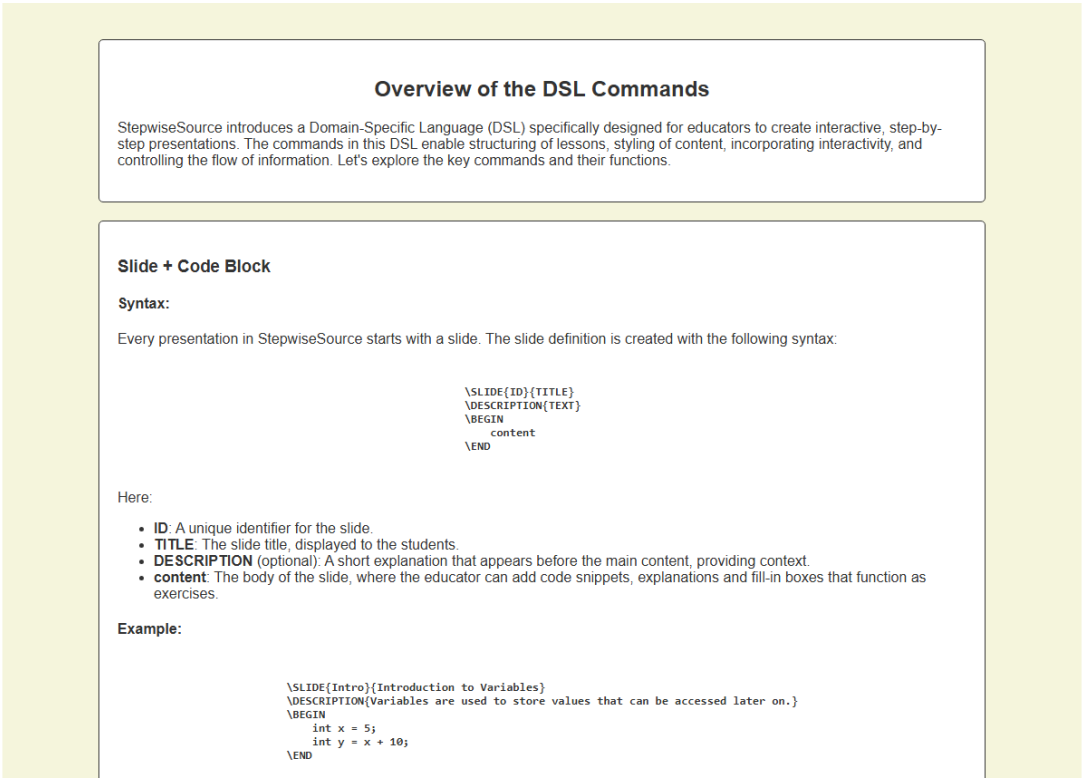


Figure 6: Landing area of the Documentation page.

On the landing area, Figure 6, a page identifier is displayed along with two calls to action: "How to Create a Presentation" and "Go to Documentation", arranged similarly to the layout of the home page (Section 6.1). The "How to Create a Presentation" button redirects users to the tutorial presentation, offering a guided introduction to the Stepwise Language's core features. The "Go to Documentation" button initiates a smooth scroll down to the documentation section below.

The documentation section (Figure 7) provides an organized overview of Stepwise Language's main features, aiding users in locating specific elements quickly. Each feature is introduced with a brief statement of its purpose, maintaining a narrative flow between sections. A syntax example is then presented for each feature, with the various components explained in detail. Additionally, a running example is built throughout all the features, illustrating how each of them can be applied in a practical context.

The page concludes with the complete source text of the tutorial presentation, allowing users to view a finished presentation with several features. Alongside this, a button provides direct access to the tutorial presentation itself, enabling users to observe the transition from source text to final presentation and realise the Stepwise Language's functionalities (see Figure 8).



Overview of the DSL Commands

StepwiseSource introduces a Domain-Specific Language (DSL) specifically designed for educators to create interactive, step-by-step presentations. The commands in this DSL enable structuring of lessons, styling of content, incorporating interactivity, and controlling the flow of information. Let's explore the key commands and their functions.

Slide + Code Block

Syntax:

Every presentation in StepwiseSource starts with a slide. The slide definition is created with the following syntax:

```
\SLIDE{ID}{TITLE}
\DESCRIPTION{TEXT}
\BEGIN
    content
\END
```

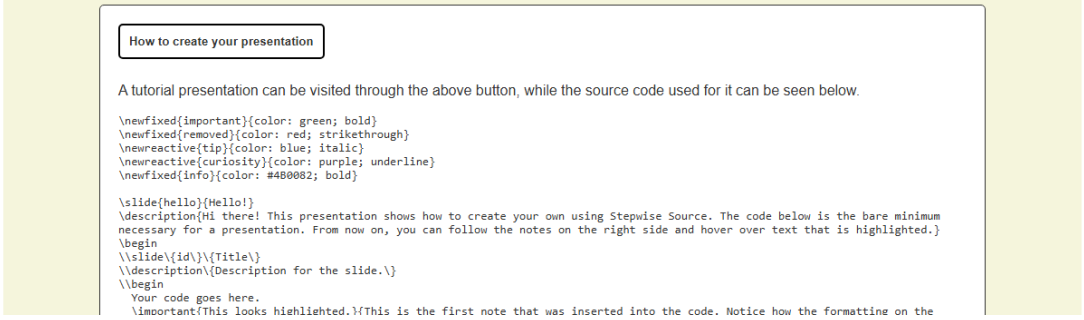
Here:

- **ID**: A unique identifier for the slide.
- **TITLE**: The slide title, displayed to the students.
- **DESCRIPTION** (optional): A short explanation that appears before the main content, providing context.
- **content**: The body of the slide, where the educator can add code snippets, explanations and fill-in boxes that function as exercises.

Example:

```
\SLIDE{Intro}{Introduction to Variables}
\DESCRIPTION{Variables are used to store values that can be accessed later on.}
\BEGIN
    int x = 5;
    int y = x + 10;
\END
```

Figure 7: Documentation section of the Documentation page.



How to create your presentation

A tutorial presentation can be visited through the above button, while the source code used for it can be seen below.

```
\newfixed{important}{color: green; bold}
\newfixed{removed}{color: red; strikethrough}
\newreactive{tip}{color: blue; italic}
\newreactive{curiosity}{color: purple; underline}
\newfixed{info}{color: #4B0082; bold}

\slide{hello}{Hello!}
\description{Hi there! This presentation shows how to create your own using Stepwise Source. The code below is the bare minimum
necessary for a presentation. From now on, you can follow the notes on the right side and hover over text that is highlighted.}
\begin
\\slide\{id\}\{Title\}
\\description\{Description for the slide.\}
\\begin
  Your code goes here.
  \important{This looks highlighted.}{This is the first note that was inserted into the code. Notice how the formatting on the
```

Figure 8: Tutorial section of the Documentation page.

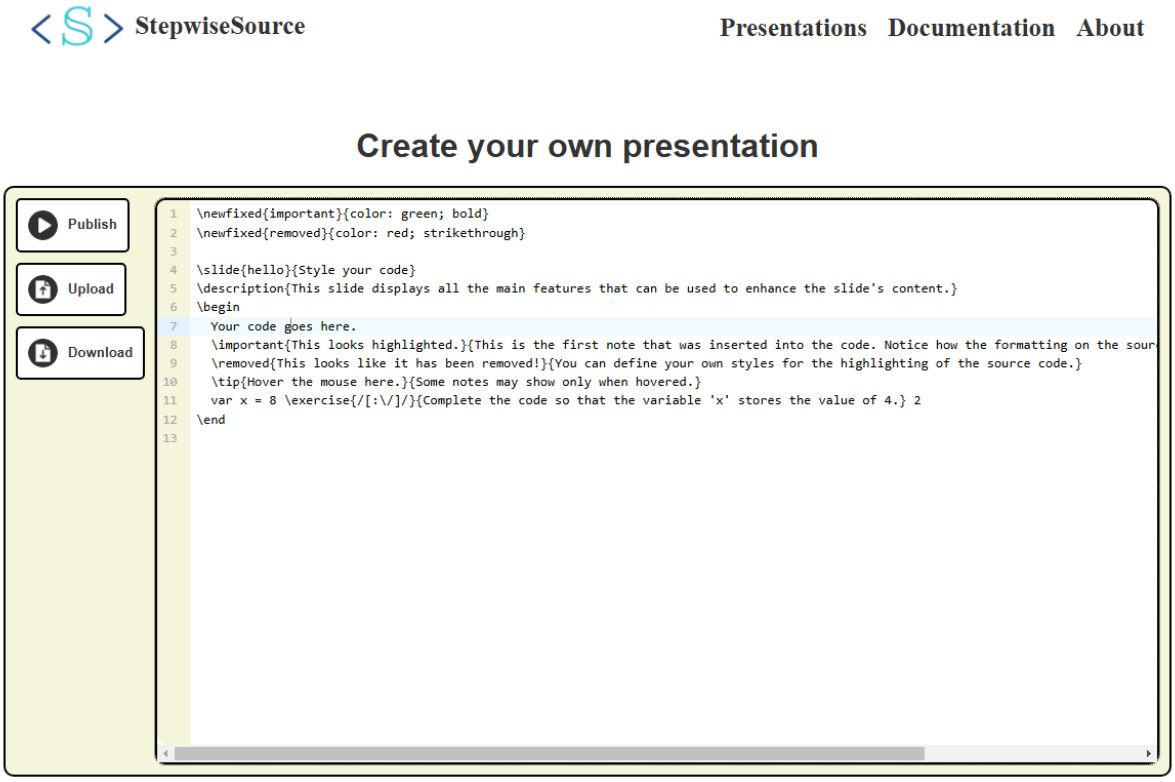


Figure 9: Presentation Editor page.

6.5 Presentation Editor Page

Once again, the page opens with a title that clearly communicates its purpose (see Figure 9). The central feature of this page is an integrated text editor, where users can write and edit the source text for creating new presentations within Stepwise Source.

A large text box serves as the editor, allowing users to directly input the code needed to build their presentations. If this page is accessed to edit an existing presentation, the editor will already be populated with its source code.

An “Upload” button opens a file explorer, enabling the user to select and load a file with pre-existing code. The file’s content appears instantly in the editor, providing flexibility for those who prefer working outside the website’s editor to import code from external sources, continue editing it within the site, or make adjustments to existing presentations.

To support users who may wish to save their progress locally, a “Download” button is also available. This function instantly downloads the current content of the text editor as a file named “presentation.txt”. This feature is useful for users who want to save a work in progress to return to it later, uploading it back into the editor as needed.



Figure 10: Presentation naming panel.

Once the presentation code is finalized, users can click the "Publish" button to publish their presentation. This action opens a panel for naming the presentation, making it easier to find later. The panel prompts for a name and displays any input or naming errors, guiding the user to make necessary corrections before publishing, as seen in Figure 10. If the page was accessed to edit an existing presentation, clicking "Publish" will update it instead of creating a new one.

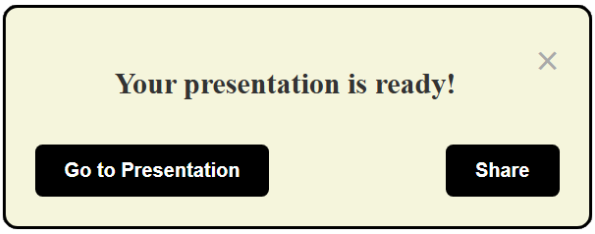


Figure 11: Presentation creation confirmation panel.

Upon successful creation, a success message appears along with two options: "Go to Presentation" and "Share" (see Figure 11). The first button redirects to the visual representation of the newly created presentation. Meanwhile, the "Share" button copies the URL of that page to the clipboard and displays it in the panel, simplifying sharing and collaboration.

6.6 Presentation Navigator Page

The central feature of Stepwise Source is the Presentation Page, where each slide in a presentation is displayed along with its source code, descriptions, annotations, styling, and unique navigation options.

At the top of the page (see Figure 12), multiple key sections provide an overview of the current slide. These sections, listed from top to bottom, include:

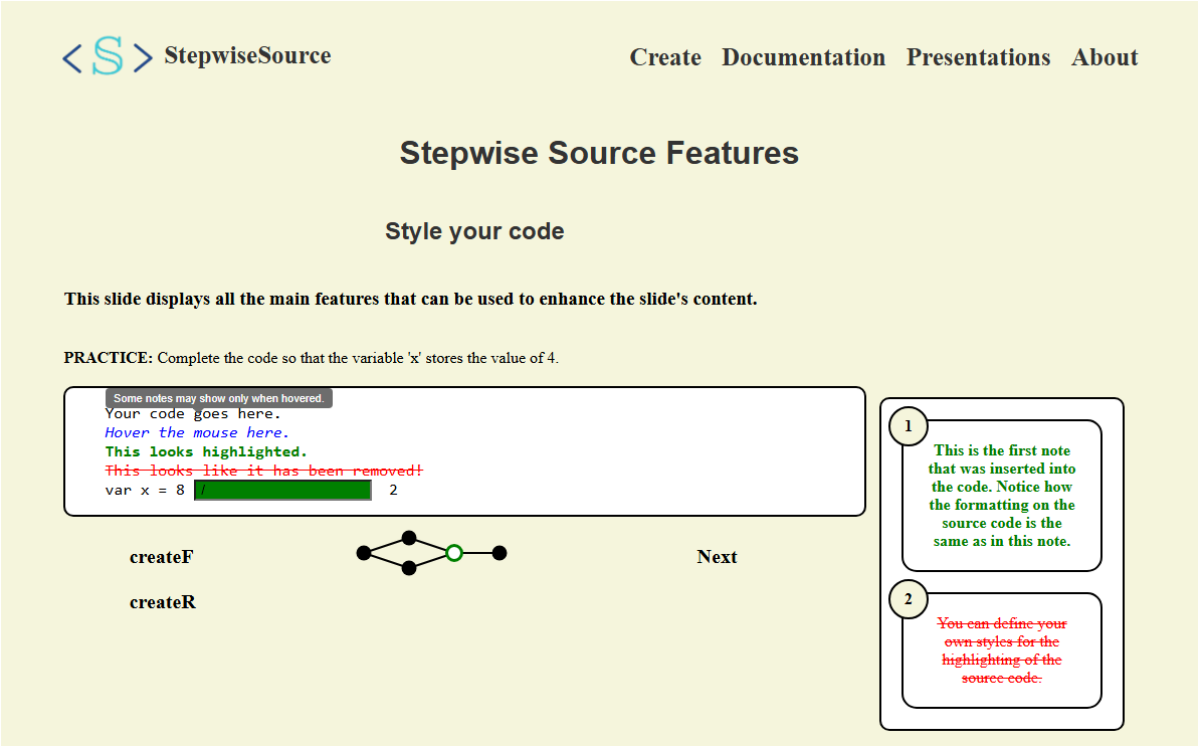


Figure 12: Presentation Navigator Page.

Presentation Title a fixed title describing the entire presentation, which remains the same across slides.

Slide Title specific to each slide, this title updates as the user navigates through the presentation.

Slide Description (optional) a detailed explanation of the slide's main goal, adding context to the source code and its components.

Exercises Instructions (optional) clear guidance on tasks or activities students need to complete to solve exercises presented on the slide.

6.6.1 Slide Content and Annotations

Beneath the slide description is the Slide Content Box, which displays the source code associated with the slide. This code may have special styling, highlighting key parts or showing annotations set by the teacher.

For instance, each styling feature enriches the user experience: lines highlighted in blue display an informational message on hover (e.g., "Some nodes may show only when hovered"). Other styles, such as green or red lines, showcase annotations that are persistently visible on the right-hand Annotations Box without requiring a hover.

The Annotations Box, located on the right, displays a list of annotations corresponding to specific lines of code. Each annotation is styled as its matching code line, ensuring consistent visual formatting between

the source code and the annotation. For example, an annotation describing a bold, green-highlighted line of code will also appear in bold green text in the Annotations Box, creating a clear visual link.

When exercises are included on a slide, an input text box appears within the slide content at the designated exercise location. This text box allows students to enter their answers and provides immediate feedback: the background colour shifts to red or green depending on the input's correctness, wrong and right respectively.

6.6.2 Presentation Navigation

The navigation section, located below the slide content, enables users to move through the presentation slides. Three main navigation tools are provided:

Previous and Next Slide Buttons "Previous" on the left and "Next" on the right allow simple linear navigation, moving backwards and forwards respectively. In the case of multiple choices (forks) in the presentation flow, more than one option will be available, as shown on the left side of Figure 13;

Navigation Graph positioned in the centre, this interactive graph shows the presentation's entire slide flow, with each circle representing a slide. While the current slide appears in white with a green border, all other slides are represented in black circles. Users can click any node to jump directly to that slide, allowing flexible navigation.

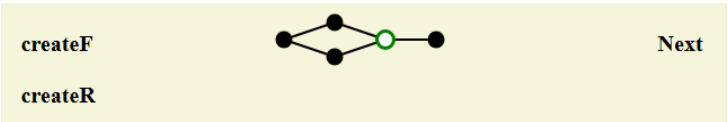


Figure 13: Presentation Navigation System.

Each slide is represented by a node in the graph, and direct paths between slides are indicated by connecting lines. When a fork happens, the graph arranges these branches vertically, with staggered heights to visually separate each branch.

Chapter 7

Conclusion

Teaching computer programming has long been a complex issue, presenting numerous challenges for both students and educators. To support this process, a tool for progressive demonstration of source code (Stepwise Source) was designed, implemented and assessed.

7.1 Assessment

Stepwise Source was designed to support both educators and students in the educational journey of computer programming. To evaluate its effectiveness, a lecture was conducted with a Master's Degree class in Informatics Education at Minho University, Braga. The session, with the participation of three professors and nine students, covered the need for educational tools that enhance programming instruction, highlighted the specific challenges Stepwise Source aims to address, and included a demonstration of the tool.

Students and teachers explored both perspectives by using the application directly while commenting on its use and usefulness. In addition to pointing out areas where the current version needed improvement, the feedback from this session, more importantly, affirmed Stepwise Source's value as a relevant contribution for teaching and learning computer programming. Both learners and educators recognized its usefulness, suggesting it would be a meaningful addition to their educational toolkit. This positive reception highlights Stepwise Source's relevance in supporting computer programming education.

This preliminary assessment session allowed for a confirmation on the positive aspects of Stepwise Source's usability and user experience. Nonetheless, further testing that includes formally structured questionnaires and more participants will be conducted in the near future.

7.2 Final Remarks

To achieve the objectives of this thesis, an initial study was carried out to identify the challenges faced in computer programming education and to evaluate existing tools designed to aid in solving these matters. This exploration provided valuable information on the problems that deserved attention. Subsequently, an architecture for the system was designed, offering a clear road map for its implementation. This plan highlighted the need to define a **DSL** (Stepwise Language) that would allow users to communicate with the application, along with the development of a parser and interpreter to process it and transform source text in visual presentations. With the communication framework established, the next step involved implementing a user-friendly interface, ensuring an intuitive experience for users. This phase also included the development of the presentation mechanism, which stands as the tool's core feature. The completion of the initial version of the tool enabled engagement with a class of students and professors, yielding positive feedback regarding its utility and highlighting areas for improvement. Overall, the project has successfully met its objectives, providing a progressive demonstration tool that benefits both students and educators, confirming the validity of its Research Hypothesis (Section 1.2). Stepwise Source incorporates functionalities designed to facilitate a deeper understanding of source code construction, including both general and specific annotations. It also offers an interactive component that enables practical study with instant feedback, while the slide navigation system allows learners to set the pace of the demonstration according to their individual needs.

7.3 Future Work

While a fully functional version of the tool has been developed, several enhancement ideas emerged during the development process and the assessment session that could not be implemented due to time shortage. One potential improvement is to clarify the relationship between the styled annotation in the source code and its comment, possibly by incorporating note identification numbers. Additionally, offering varying levels of detail in the annotations could further personalize the learning experience, allowing students to engage at their preferred pace. Another suggested feature is the integration of a user management system, enabling educators to create dedicated classrooms where they could organize presentations by specific themes. This would them to share a curated list of related presentations through a single link, enhancing accessibility and coherence for their students. The last component of future improvement would be the possibility of embedding presentations into other web pages, including Learning Management Systems

(LMSs) such as Moodle.

Finally, a more robust and structure testing procedure would result in a better overview of Stepwise Source's usability and user experience, but also its value as a computer programming educational tool.

Bibliography

Ada Computer Science. Ada computer science. https://adacomputerscience.org/concepts/trans_bnf, 2024.

Kofi Adu-Manu, John Arthur, and Prince Amoako. Causes of failure of students in computer programming courses: The teacher learner perspective. *International Journal of Computer Applications*, 77:27–32, 09 2013. doi: 10.5120/13448-1311.

AV Aho, R Sethi, and JD Ullman. Compiler principles, techniques and tools, bell telephone laboratories, inc. 1987. *Google Scholar Google Scholar Digital Library Digital Library*, 2024.

Cristiana Esteves Araujo. *Training Computational Thinking: exploring approaches supported by Neuroscience*. PhD thesis, University of Minho, Mar 2022. PDInf - prethesis.

Albert Bandura, W. H. Freeman, and Richard Lightsey. Self-efficacy: The exercise of control. *Journal of Cognitive Psychotherapy*, 13(2):158–166, 1999. doi: 10.1891/0889-8391.13.2.158. URL https://connect.springerpub.com/content/sgrjcp/13/2/158.

Yorah Bosse, David Redmiles, and Marco A. Gerosa. Pedagogical content for professors of introductory programming courses. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 429–435, New York, NY, USA, 2019a. Association for Computing Machinery. ISBN 9781450368957. doi: 10.1145/3304221.3319776. URL https://doi.org/10.1145/3304221.3319776.

Yorah Bosse, David F Redmiles, and Marco Gerosa. Connections and influences among topics of learning how to program. In *2019 IEEE Frontiers in Education Conference (FIE)*, pages 1–8, 2019b. doi: 10.1109/FIE43999.2019.9028393.

Brigham Young University. Brigham young university. https://cs111.byu.edu/lectures/33-Backus-Naur_Form/, 2024.

Chin Soon Cheah. Factors contributing to the difficulties in teaching and learning of computer programming: A literature review. *Contemp. Educ. Technol.*, 12(2):ep272, May 2020.

Cisco Networking Academy. Python essentials 1 course. https://skillsforall.com/course/python-essentials-1, 2024.

Codecademy. What Is a Programming Language? https://www.codecademy.com/resources/blog/programming-languages/, 2020. Accessed: 2024-01-15.

Codecademy. What is programming? https://codecademy.com/article/what-is-programming, 2024. Accessed: 2024-01-15.

Alvaro Costa Neto, Cristiana Araújo, Maria João Varanda Pereira, and Pedro Rangel Henriques. Programmers' Affinity to Languages. In Pedro Rangel Henriques, Filipe Portela, Ricardo Queirós, and Alberto Simões, editors, *Second International Computer Programming Education Conference (ICPEC 2021)*, volume 91 of *Open Access Series in Informatics (OASIcs)*, pages 3:1–3:7, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-194-8. doi: 10.4230/OASIcs.ICPEC.2021.3. URL https://drops.dagstuhl.de/opus/volltexte/2021/14219.

Coursera. What Is Programming? https://www.coursera.org/articles/what-is-programming, 2023. Accessed: 2024-01-15.

Create React App. Create react app. https://create-react-app.dev/, 2024.

curl. curl. https://curl.se/, 2024.

John DeNero. Cs 61a. https://cs61a.org/study-guide/bnf/, 2024.

Docker. Docker. https://www.docker.com/, 2024.

Sofia Guilherme Rodrigues dos Santos. Automatic grading of programming exercises. University of Minho, August 2023.

Brendan Eich. The javascript programming language. *Netscape Communications*, 1995.

Express. Express. https://expressjs.com/, 2024.

Gustavo Galvão, Alvaro Costa Neto, Cristiana Araújo, and Pedro Rangel Henriques. The Visual Programming Environment ROBI for Educational Robotics. In João Cordeiro, Maria João Pereira, Nuno F.

Rodrigues, and Sebastião Pais, editors, *11th Symposium on Languages, Applications and Technologies (SLATE 2022)*, volume 104 of *Open Access Series in Informatics (OASIcs)*, pages 14:1–14:15, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-245-7. doi: 10.4230/OASIcs.SLATE.2022.14. URL https://drops.dagstuhl.de/opus/volltexte/2022/16760.

Anabela Gomes and Antonio Mendes. Learning to program - difficulties and solutions. In *International Conference on Engineering Education – ICEE 2007*, pages 283–287, 01 2007.

Philip J. Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 579–584, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318686. doi: 10.1145/2445196.2445368. URL https://doi.org/10.1145/2445196.2445368.

Felix Halim. uhunt. https://uhunt.onlinejudge.org/, 2024.

C. Hundhausen and S. Douglas. Using visualizations to learn algorithms: should students construct their own, or view an expert's? In *Proceeding 2000 IEEE International Symposium on Visual Languages*, pages 21–28, 2000. doi: 10.1109/VL.2000.874346.

Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, 2002. ISSN 1045-926X. doi: https://doi.org/10.1006/jvlc.2002.0237. URL https://www.sciencedirect.com/science/article/pii/S1045926X02902375.

Stephen C. Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

Reyes Juárez-Ramírez, Christian X. Navarro, Verónica Tapia-Ibarra, Ricardo Macías-Olvera, and César Guerra-García. What is programming? putting all together - a set of skills required. In *2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 11–20, 2018. doi: 10.1109/CONISOFT.2018.8645956.

Kattis. Kattis. https://www.kattis.com/, 2024.

Khan Academy. What Is Programming? https://pt.khanacademy.org/computing/computer-programming/programming/intro-to-programming/v/programming-intro, 2024a. Accessed: 2024-01-15.

Khan Academy. Khan academy. https://www.khanacademy.org/, 2024b.

Abdulmalik Ahmad Lawan, Abdo Sulaiman Abdi, Anas Ahmad Abuhassan, and Mohammed Salim Khalid. What is difficult in learning programming language based on problem-solving skills? In *2019 International Conference on Advanced Science and Engineering (ICOASE)*, pages 18–22, 2019. doi: 10.1109/ICOASE.2019.8723740.

José Paulo Leal and Fernando Silva. Mooshak: a web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, 2003. doi: https://doi.org/10.1002/spe.522. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.522.

Michael E. Lesk and Eric Schmidt. *Lex: A lexical analyzer generator*, volume 39. Bell Laboratories Murray Hill, NJ, 1975.

Mailgun. Mailgun. https://www.mailgun.com/, 2024.

Daniel D. McCracken and Edwin D. Reilly. *Backus-Naur form (BNF)*, page 129–131. John Wiley and Sons Ltd., GBR, 2003. ISBN 0470864125.

Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, 62(2):77–90, 2019. doi: 10.1109/TE.2018.2864133.

Bertrand Meyer. Codeboard. https://codeboard.io/, 2024.

MongoDB. Mongodb. https://www.mongodb.com/, 2024.

Mongoose. Mongoose. https://mongoosejs.com/, 2024.

nginx. nginx. https://nginx.org/en/, 2024.

Node.js. Node.js. https://mongoosejs.com/, 2024.

I ping Chu. Csc447 concepts of programming languages, de paul university. https://condor.depaul.edu/ichu/csc447/notes/wk3/BNF.pdf, 2024.

Li ping Zhang and Dong sheng Liu. Ast-based multi-language plagiarism detection method. In *2013 IEEE 4th International Conference on Software Engineering and Service Science*, pages 738–742, 2013. doi: 10.1109/ICSESS.2013.6615411.

Martinha Piteira and Carlos Costa. Learning computer programming: study of difficulties in learning programming. In *Proceedings of the 2013 International Conference on Information Systems and Design of Communication*, ISDOC '13, page 75–80, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322997. doi: 10.1145/2503859.2503871. URL https://doi.org/10.1145/2503859.2503871.

Alberto José Proença. Introdução aos sistemas de computação. University of Minho, 02 2018.

Viera K. Proulx. Programming patterns and design patterns in the introductory computer science course. *SIGCSE Bull.*, 32(1):80–84, mar 2000. ISSN 0097-8418. doi: 10.1145/331795.331819. URL https://doi.org/10.1145/331795.331819.

Python. Python. https://www.python.org/, 2024.

Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Educ.*, 18(1), oct 2017. doi: 10.1145/3077618. URL https://doi.org/10.1145/3077618.

Science Direct. Science direct. https://www.sciencedirect.com/topics/computer-science/backus-naur-form, 2024.

Richard Stallman. Gnu bison. https://www.gnu.org/software/bison/, 2024.

Paula Correia Tavares, Elsa Ferreira Gomes, and Pedro Rangel Henriques. Animation and automatic evaluation in supporting the teaching of programming. In *2015 10th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–4, 2015. doi: 10.1109/CISTI.2015.7170548.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794, 2019. doi: 10.1109/ICSE.2019.00086.