VCAT: An automatic assessment model for visual programming languages

Lucas Mendonça de Souza

Thesis presented to the
Institute of Mathematics and Statistics
of the University of São Paulo
in partial fulfillment
of the requirements
for the degree of
Master of Science

Program: Computer Science
Advisor: Prof^a. Dr^a. Anarosa Alves Franco Brandão

During the development if this work, the author received financial support from CNPq

São Paulo
August, 2023

VCAT: An automatic assessment model for visual programming languages

Lucas Mendonça de Souza

This is the original version of the
thesis prepared by candidate Lucas
Mendonça de Souza, as submitted
to the Examining Committee.

À minha mãe, Dinalva M. dos Anjos, sem
a qual eu não teria chegado até aqui.
À todos os docentes que passaram por minha caminhada, que me deram as bases necessárias para
essas conquista: Profa. Anarosa e Prof Leonidas,
Prof. Eduardo Cambruzzi e Profa. Eliete Franco
que são e sempre serão minhas referências.
Aos amigos que seguraram minha mão nos momentos de
inseguranças e incertezas dando suporte para seguir em
frente: Vanessa Mendes Brito, Andrey Ribeiro de Almeida,
Igor Moreira Félix, e tantos outros não menos importantes.

Resumo

Lucas Mendonça de Souza. **VCAT: um modelo de avaliação automatica para linguages visuais**. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

Aprender a como programar tem se tornado um aspecto crucial da sociedade moderna. A presença de tecnologias digitais no dia-a-dia requer um entendimento básico de como software funciona de forma a ter consciência de como isso afeta a vida de todos. Ademais, as preocupações em relação a segurança digital e privacidade tornam ainda mais relevante a compreensão dos conceitos da ciência da computação. Esses aspectos não estão somente relacionados a programação e podem ser referidos na literatura como Pensamento Computacional. Pensamento Computacional é entendido como a aplicação dos conceitos da ciência da computação em diferentes contextos da vida cotidiana. Atentando a esse cenário, governos no mundo inteiro estão implementando novos currículos escolares que incorporam a programação como uma habilidade chave. Entretanto, a literatura mostra quer aprender a programar é uma tarefa complexa e difícil. Em alguns casos, os índices de evasão e reprovação chegam a ser alarmantes. Desta forma, afim de mitigar esses problemas de aprendizagem alguns pesquisadores sugerem o uso do paradigma de programação visual. Esse paradigma consiste no uso de elementos visuais para a construção de algoritmos. Neste contexto, os experimentos presentes na literatura afirmam ter encontrado indícios de melhora no processo de aprendizagem, como melhores notas e motivação na aprendizagem. Uma outra tecnologia utilizada no ensino e aprendizagem de programação é avaliação automática de programas. As ferramentas de avaliação automática avaliam a corretude de um algoritmo utilizando diferentes métodos. Elas permitem com que os professores consigam avaliar um grande número de exercícios e ao mesmo tempo fornecer uma retroalimentação rápida aos estudantes. Todavia, apenas dois sistemas de programação visual que fornecem avaliação automática são reportados na literatura: iVProg and Chentry. Ambos os sistemas, entretanto, oferecem métodos de avaliação bem limitados. Assim, o objetivo dessa pesquisa é propor um modelo, chamado VCAT, de avaliação automática de programas para sistemas de programação visual afim de permitir que linguagens de programação visual tenham acesso a avaliação automática. Além disso, esse estudo também busca melhorar a retroalimentação fornecida pelo método de avaliação automática conhecido como comparação de saídas. Uma instanciação do modelo foi feita tendo como base o iVProg. Uma segunda instanciação do modelo também foi desenvolvida para uma linguagem visual criada pelo autor usando o arcabouço Blockly. Melhorias foram feitas no algoritmo de comparação de saídas presente no modelo. Um experimento com estudantes do curso de verão para introdução à programação foi elaborado para avaliar como os estudantes percebem as melhorias no algoritmo de comparação de saídas e no uso de sistemas de programação visual. Os resultados indicam que o modelo apresentado é capaz de prover avaliação automática para outras linguagens visuais além do iVProg. Os dados do experimento mostram que as mudanças implementadas na retroalimentação do método de comparação de saídas foram bem recebidos pelos estudantes em comparação com a implementação tradicional do método no VPL, uma ferramenta de ensino de programação que usa comparação de saídas como método de avaliação. Além disso, os dados também apontam benefícios do uso de programação visual como uma ferramenta no processo de aprendizagem de programação textual.

Palavras-chave: avaliação automática. programação visual. ensino de programação.

Abstract

Lucas Mendonça de Souza. **VCAT: An automatic assessment model for visual programming languages**. Thesis (Master's). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

Learning how to code is becoming a crucial aspect of modern society. The presence of digital technologies in everyday life requires some basic understanding on how software works in order to be aware of how this affects everyone's life. Moreover, concerns on privacy and digital security also raises the relevance of understanding computer science concepts. These aspects are not only related to programming and can be referred in the literature as Computational Thinking. Computational Thinking is understood as the application of computer science concepts in different contexts of day-to-day life. As an answer to this, governments worldwide are implementing new school curricula that incorporate programming as a key skill. However, the literature shows that learning how to program is a complex and difficult task. In some cases, the dropout and failure rates can be very alarming. So, in order to mitigate these learning problems some researchers employed the visual programming paradigm. This paradigm consists in using visual elements to code algorithms. In this context, the experiments reported in the literature claim improvements in the learning process, such as better grades and motivation to learn. Another technology employed to support teaching and learning programming is the automatic assessment of programs. These tools can automatically evaluate a program using different methods. They allow teachers to assess a large number of exercises and also provide quick feedback to students. Nonetheless, at the best of our knowledge, only two visual programming systems that provide automatic assessment were found: iVProg and Chentry. Both systems offer a limited method of assessment. Therefore, this research proposes VCAT, an automatic assessment model to support Visual Programming Systems on providing this functionality to diverse visual programming languages. In addition to providing such a functionality, it also introduces improvements to its output matching algorithms, in order to provide a better experience with the generated feedback. The model was successfully instantiated considering iVProg and Blockly, which indicates that it is independent of the underlying visual programming language of the system. An experiment with students of the summer program for introductory programming was designed to assess how the students perceived the feedback improvements and the use of visual programming systems. Data from the experiment shows that the changes implemented in the output matching feedback was well received by the students when compared to VPL, a similar tool using the same type of automatic assessment without improvements. Moreover, data also suggests benefits of using visual programming as a learning step toward text based programming.

Keywords: automatic assessment. visual programming. programming teaching.

List of abbreviations

CT	Computational Thinking
IS	Information Systems
DSR	Design Science Research
VPL	Virtual Programming Lab
AAT	Automatic Assessment Tool
AST	Abstract Syntax Tree
IME	Instituto de Matemática e Estatística
USP	Universidade de São Paulo
I/O	Input and Output
VCAT	Visual Code Assessment Tool
JSON	JavaScript Object Notation
EBNF	Extended Backus-Naur Form

List of Figures

1.1 Design Research Method phases as described by Peffers et al. **Source**: Adapted from Peffers *et al.* (2007) 4

2.1 A program that reads an integer and prints it coded in Pascal and Assembly language 8

2.2 Pygmalion's user interface and example code. 12

2.3 Fragment of code show part of the factorial implementation in Pict Source: Adapted from Glinert and Tanimoto (1984) 13

2.4 Alice user interface and its windows 14

2.5 A program that reads two numbers and prints their sum in Alice 15

2.6 Scratch main user interface. 15

2.7 A program that reads two numbers and prints their sum in Scratch. . . . 16

2.8 Chentry main user interface. 16

2.9 A program that reads two numbers and prints their sum in Chentry . . . 17

2.10 A program that reads two numbers and prints their sum in iVProg. . . . 17

2.11 A program that reads two numbers and prints their sum in Alice 18

3.1 Dynamic assessment through output matching 24

3.2 Illustration of connection between source code and AST 29

4.1 Simplified view of VCAT model 32

4.2 Domain diagram for VCAT Assessment component 37

5.1 Domain diagram for the JavaScript implementation of $\mathcal{M}$ 42

5.2 Class diagram for the AST package JavaScript implementation 43

5.3 Class diagram for the CodeExecutor and IO packages JavaScript implementation 44

5.4 Class diagram for the Assessment package JavaScript implementation . . 46

5.5 List of command blocks available in iVProg 47

5.6 Output Matching feedback page showing the type-based assessment of the output . . . 50
5.7 User interface for Blockly instantiation . . . 51

6.1 iVProg and VPL scores comparison . . . 58
6.2 Box plots for both tools G_{final} scores to compare the values distribution shape . . . 60

7.1 Occurrence graph for Q2 regarding AAT capabilities of iVProg and VPL from very useless(-2) to very useful(2) . . . 61
7.2 Occurrence graph for Q1 on iVProg and VPL ease of usage from very hard(-2) to very easy(2) . . . 62
7.3 Occurrence graph for Q3 on iVProg and VPL usage as a study tool from very hard to use (-2) to very easy(2) . . . 63
7.4 Occurrence graph for Q4 on if they felt confident when using iVProg and VPLfrom totally disagree(-2) to totally agree(2) . . . 64
7.5 Occurrence graph for Q5 on the students perception of exercise difficulty when using iVProg and VPL from very challenging(-2) to very easy(2) . . 65
7.6 Occurrence graph for Q6 on effects of each tool over the students perception of exercise difficulty from much harder(-2) to much easier(2). 65
7.7 Occurrence graph for Q7 on if the students felt efficient while using iVProg and VPL from totally disagree(-2) to completely agree(2). . . . 66
7.8 Occurrence graph for Q8 on the students perception the contribution of the tools(iVProg & VPL) for algorithm learning from not contributed at all (-2) to contributed significantly (2). . . . 67

List of Tables

3.1 Summary of techniques designed to improve AAT . . . 27

6.1 English translated version of the questionnaire used to assess iVProg and its AAT . . . 57

6.2 Occurrence table for the percentage of answers' scores for the tools iVProg — VPL given by the 57 students 58

B.1 Built-in functions available in the Conversion library 78
B.2 Built-in functions available in the Mathematics library 79
B.3 Built-in functions available in the Mathematics library 79
B.4 Built-in functions available in the Mathematics library 80

List of Programs

4.1 Example implementation of F_{trans} for a hypothetical visual language L_H. . 34
4.2 Example of an assignment description file for output matching in VCAT. 39
4.3 Example of an assigment description file for static assessment in VCAT. . 39
5.1 Code snippet for iVProg C_{text} generation for an assignment block. 48
5.2 Example code for AST nodes generation for iVProg's Assignment command. 48
5.3 Code snippet for executing a program using F_{trans} and VCAT. 49
5.4 Code snippet for running automatic assessment in iVProg using VCAT. . 49
5.5 Code snippet of F_{trans} implementation for the visual language created using Blockly. 52
5.6 Code snippet for executing a program using F_{trans} and VCAT. 53
5.7 Code snippet for performing automatic assessment for Blockly-based visual language. 54

Contents

Introduction **1**

1 Research Method **3**

1.1 Design Science Method 3

1.2 Applying the method to this project 5

2 Theoretical Framework **7**

2.1 Computer Programming 7

2.1.1 Programming Learning 8

2.1.2 Computational Thinking 10

2.2 Visual programming languages 11

2.2.1 Formal definition of Visual Languages 12

2.2.2 Visual Programming System 14

2.2.3 Programming learning and Visual Programming 18

3 Related Work **21**

3.1 Automatic Assessment tools 21

3.1.1 Static and Dynamic Assessment 22

3.1.2 Improving Automatic Assessment Methods 24

3.1.3 Automatic Assessment and Programming 26

3.2 Abstract Syntax Tree 28

4 Visual Code Assessment Tool - VCAT **31**

4.1 VCAT's Architecture 31

4.2 Implementing F_{trans} 33

4.3 VCAT's core: $\mathcal{M}$ software component 35

4.3.1 AST component 35

4.3.2 Code Executor & IO components 36

4.3.3 Assessment component 36

4.3.4 Assignment Description 39
4.4 Model limitations 40

5 VCAT instantiation 41
5.1 Implementing software component $\mathcal{M}$ 41
5.1.1 AST package 42
5.1.2 Code executor & IO packages 43
5.1.3 Assessment package 45
5.2 Instantiating for iVProg 46
5.2.1 Implementation of F_{trans} 47
5.2.2 Inputting C_{text} into $\mathcal{M}$ 49
5.3 Instantiating for Blockly 51
5.3.1 Implementing F_{trans} 52
5.3.2 Inputting C_{text} into $\mathcal{M}$ 53

6 VCAT Evaluation 55
6.1 Experiment 55
6.2 Questionnaire 56
6.2.1 Results 57
6.3 Exercises' grades analysis 59

7 Discussions 61
7.1 Questionnaire answers 61
7.2 iVProg and VPL grade analysis 67

8 Conclusions 69
8.1 Final considerations 69
8.2 Future work 70

A Syntax rules for VCAT textual language presented using EBNF language 71

B List of the built-in functions available in VCAT 77
B.1 Conversion 77
B.2 Mathematics 77
B.3 Text 78
B.4 IO 78

References 81

Introduction

Technology has become a ubiquitous element of everyone's life. Smartphones, TVs and other home appliances, almost everything in everyday life has some computational capabilities. These capabilities are not limited to being able to compute and automated some activities, but being able to talk to other appliances and the Internet itself (Coughlan *et al.*, 2012). Besides that, in some context to access to public information or services the citizen needs to know how to operate a computer and its peripherals (Vee, 2013). Moreover, nowadays it is not only enough to understand how to use a computer, it is also expected some understanding of how a software works and how to build one (Vee, 2013; T. Dufva and M. Dufva, 2016; Eshet-Alkalai, 2004). In other words, the modern society requires some knowledge on computer programming.

As a consequence, governments and lawmakers world-wide are trying to rethink their educational process in the light of this reality. They are aware of the necessities of a connected world from productive, civic and privacy standpoint (Eshet-Alkalai, 2004; Vee, 2013). So, in order to prepare the population to this new reality, a new curriculum that inserts the computer and programming specially as a relevant subject is being implemented around the world. The literature also recognizes the need for an understanding of computer science concepts and their relevance in all areas beyond the exact sciences (Vee, 2013; Eshet-Alkalai, 2004; Wing, 2006; Papert, 1980). This conceptual framework is usually referred as Computational Thinking (CT). To Wing (2006), CT is the ability to employ computer science concepts such as algorithm design, decomposition and abstraction to solve problems in everyday life.

However, learning how to program is not an easy task. Programming is reputed as hard and complex, requiring the ability to deal with concepts of different areas depending on the problem being solved (Michael Edelgaard Caspersen, 2007; Lahtinen *et al.*, 2005; Gomes and Mendes, 2007). The students also face a variety of difficulties during the learning process ranging from the teaching method used to programming language being taught (Gomes and Mendes, 2007; Michael Edelgaard Caspersen, 2007). Moreover, this can affect not only the final grades but also the students' motivation to learn and their desire to finish the course.

Different studies have suggested the use of visual programming paradigm as a way of mitigating the problems related to programming learning. In this paradigm, a Visual Programming Language (VPL) is used to enable the user to construct their code. A VPL employs visual elements which the student use to visually build their algorithm. Studies like Brandao *et al.* (2012), Sáez-López *et al.* (2016), C.-K. Chang *et al.* (2017), and Resnick *et al.* (2009) have found that the use of VPL can improve grades, students' motivation regarding programming learning and also their desire to keep studying computer science subjects. So, visual programming systems like Scratch, Alice, iVProg and Chentry all implement the visual programming paradigm.

Additionally, another strategy used to reduce the problems in programming teaching and learning is the use of automatic assessment tools (AAT). These tool are system developed with the goal of executing and evaluating programs. The evaluation process can be of different natures. They can evaluate the functionality of a program, its code structure or performance. Reguera and Leiva (2017), Lappalainen *et al.* (2017), and S. Li *et al.* (2016) claim that the use of AAT can increase students' engagement, improve their grades. Also, by automating the assessment process, it can reduce the teacher's amount of labor regarding program evaluation.

Nonetheless, there is no report in the literature of a visual programming language that uses the affordances of AAT. Only iVProg and Chentry are reported in the literature with some sort of automatic assessment. In this context, iVProg can perform some basic functional assessment of the program by using test cases that describes a set of inputs and expected outputs (Brandao *et al.*, 2012). Chentry, in the other hand, perform some structural analysis of the code based on the logs produced during program construction. The analysis aim to identify if the constructed program is similar to the reference solution provided by the teacher (J.-H. Kim *et al.*, 2019).

So, the goal of this research project is to investigate AAT strategy and VPL to propose an automatic assessment model that would allow any VPL to use the affordances provided by automatic assessment of programs. The method used to achieve this goal is the design science paradigm (Peffers *et al.*, 2007). The method provided a theoretical framework that allows the development of an artifact, designed to solve a specific problem in a specific context. It is an iterative process that provided the methodological basis required to validate the whole process as a scientific method.

Chapter 1 will discuss the methodology employed in this research. The theoretical framework used in this project is presented in chapter 2. In chapter 3, it will be presented some related works on automatic assessment and its methods, while chapter 4 discuss the proposed solution. Finally, chapter **??** will present the schedule.

Chapter 1

Research Method

The theoretical and methodological concepts that guide this research are based upon the Design Science paradigm. The paradigm provides the foundation required to use the production of artifacts as a valid scientific method by an epistemological point of view (Peffers *et al.*, 2007; Pimentel *et al.*, 2019). Thus, this project employs the design science research method or design method, since its purpose is to provide an automatic assessment model through the production of an artifact. In this context, an artifact represents a project or engineering design designed to solve a problem. Therefore, artifact is a product aimed at a given context and planned by applying knowledge and conjectures about the world (Pimentel *et al.*, 2019).

1.1 Design Science Method

The Design Science method is divided into a set of stages, varying by author and the nature of the research area (Peffers *et al.*, 2007; Hevner *et al.*, 2004; Çağdaş and Stubkjær, 2011). This project will follow the stages described by Peffers *et al.* (2007), in his attempt to develop a framework for Design Science Research (DSR) to be used in Information Systems (IS) research. As said by Pimentel *et al.* (2019), research in Informatics in Education (IE) can be seen as a sub-area of IS since both frequently focus on developing artifacts to solve problems. In this case, IS focus on business problems, while IE focus on problems of educational nature.

Essentially, Peffers *et al.* (2007) divides the DSR research flow into six stages, as shown in Figure 1.1. First stage (Identify problem & Motivate) is concerned with problem identification and the relevance of the solution. The second stage (Define objective of a solution) results from the objectives inferred from the identified problem in the first stage, evaluating if they are possible and feasible. In the third stage (Design & Development) the functionalities and architecture of artifact is decided and the artifact created. The fourth stage is about the demonstration of the artifact regarding its potential to solve one or more instances of the problem. This stage requires "effective knowledge of how to use the artifact to solve the problem"(Peffers *et al.*, 2007).

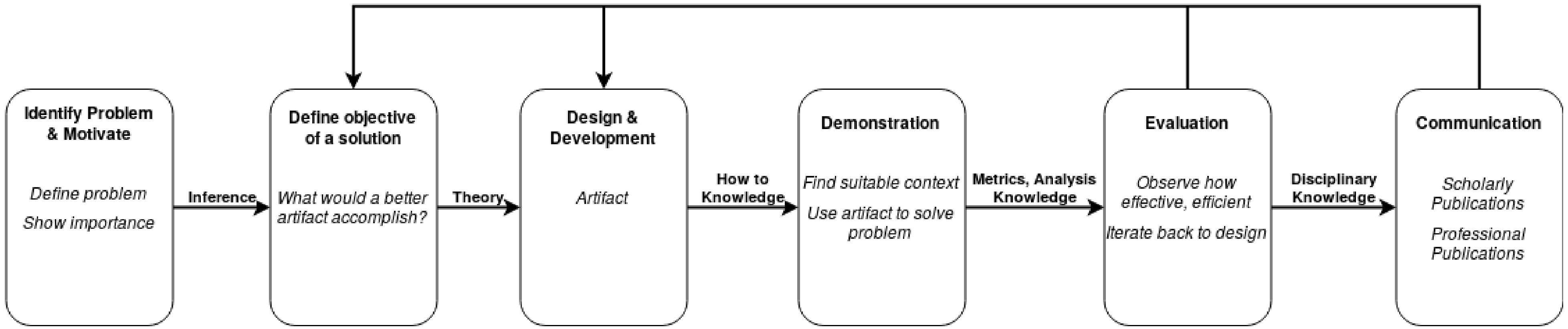


Figure 1.1: *Design Research Method phases as described by Peffers et al.*
***Source**: Adapted from* PEFFERS *et al.* (*2007*)

During the fifth stage, observations and measurements of how well the artifact solves the problem are done. In this stage, comparisons between the defined objectives and the actual results are compared and the research can iterate back to previous stages if required. Finally, in the sixth stage, the artifact is presented to the appropriate audience, scholarly or professional. During the presentation, the problem the artifact is supposed to solve must be exposed as is the novelty of the solution.

1.2 Applying the method to this project

In stage one, the lack of a visual programming teaching environment with automatic assessment was identified through the literature on visual programming languages. There were only two software found that had such capability: iVProg (Brandão *et al.*, 2016) and Chentry (J.-H. Kim *et al.*, 2019). iVProg is a visual programming environment with automatic assessment developed by the Laboratory of Informatics in Education (LInE) at the Institute of Mathematics and Statistics of Universidade de São Paulo. Chentry is a visual programming system developed to assess programming task in the www.playentry.com platform. Both tools will be discussed in Section 2.2.2.

The relevance of the problem comes from studies like Meerbaum-Salant *et al.* (2010) and C.-K. Chang *et al.* (2017) that shows how visual programming languages can potentially improve grades and engage students. Moreover, introductory programming courses are regarded as very difficult (Gomes and Mendes, 2007; Watson and F. W. Li, 2014; Michael Edelgaard Caspersen, 2007; Rapkiewicz *et al.*, 2006; Brandão *et al.*, 2016) and can also have alarming failure rates (Watson and F. W. Li, 2014; Rapkiewicz *et al.*, 2006; Brandão *et al.*, 2016) also regarded as difficult and complex task, specially for beginners. Thus, different solutions have been proposed to mitigate this problem, including visual programming languages and automatic assessment (Basnet *et al.*, 2018; Cardoso *et al.*, 2018). The aspects of visual programming languages and automatic assessment of programs will be discussed in Chapter 2 .

Regarding the automatic assessment capabilities of iVProg, the tool can only assess a program by output matching. In this method, a set of inputs is given to the program and the output generated is compared to the expected output to check if they match perfectly. This can be done case by case or by saving the output to a file and comparing it to the reference output file. There are lots of criticism in the literature regarding this method (C. K. Poon *et al.*, 2016; Yu *et al.*, 2017), since it usually means students can focus on producing the exact expected output instead of working on the problem itself. Also, according to the aforementioned works, it can also be a source of frustration, since empty spaces, punctuation or misspelling can cause the assessment to fail. Chentry, in the other hand, can only assess code based on *log* comparison which cannot guarantee that the program behaves as expected.

At this point the first stage of the method was finished, since the problem was identified and its relevance was justified by the absence of an adequate solution in the literature. Therefore, in the second stage, the objective defined for the potential solution was to provide means to extend and improve an existing solution while developing an automatic assessment model of visual programming languages that would expand and improve the availability of their assessment methods. In this stage, iVProg was chosen as the subject of the research since its source code is publicly available and the author is one of its active developers. A better artifact would allow not only for an improved output matching functionality but provide better feedback to students. Moreover, it is desirable that the artifact also allows for other methods of assessment to be integrated to it.

The improvements in the output matching functionality are aimed to reduce the known binary behavior of the algorithm, making room for partially correct solutions instead of only correct and wrong. On the improved feedback aspects, the objective is to design an artifact that provides context to its error messages making it easier to students to identify the source of the problem. The feedback provided by the improved output matching algorithm is also enhanced to provide more details on the failing text cases, showing the differences between the expect output and the generated one. The details regarding the designed solution can be found in Chapter 4

During the third stage, the artifact was effectively implemented using different web technologies like the HTML5 stack (HTML, CSS and JavaScript) and the NodeJs tool. This decision was motivated by the fact that our subject, iVProg is already developed as a decentralized web applications and the author wanted to keep that same design philosophy. The output matching algorithm was improved to better deal with different types of outputs. Now, it breaks down the output assessment into groups of the standard types (text, boolean and numeric). The final grade of a test case is the mean of the grade of the assessment of each standard type in the output. In case of textual output, the algorithm will penalize the grade based on the textual differences (in number of characters) between the output generated by the student program and the output expected by the test case.

In fourth and fifth stages the experiment was devised. The experiment consisted of collecting qualitative and quantitative data from participating students from the summer program of the Institute of Mathematics and Statistics at USP. The students were enrolled in the introductory programming course using C and have fully remote classes using iVProg, instantiated using the proposed model, and visual programming language, a Moodle plugin that performs automatic assessment of code for languages like C, C++ and Java, to name a few. To collect qualitative data, 0a questionnaire was created, inspired by Savi *et al.*, 2011 questionnaire for educational games. The quantitative data were derived from the grades of the students in the exercises and the logs generated by interacting with both tools, iVProg and visual programming language. The analysis will consist in extracting information from the data that shows if the implemented artifact does achieves the expected improvements on feedback and output matching algorithm. Both qualitative and quantitative data will be cross-checked to identify if the quantitative data supports the findings from the qualitative data. In the next Chapter, it will be discussed the concepts and theoretical framework used to develop the proposed solution.

Chapter 2

Theoretical Framework

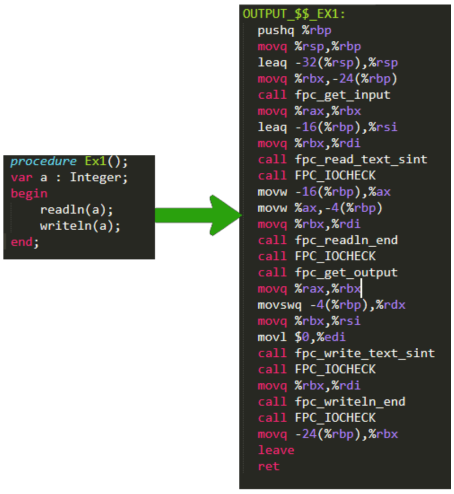
2.1 Computer Programming

A program is a set of instructions written in a specific language that must undergo transformations to be run by the computer (OUALLINE, 1997). According to Caspersen (Michael Edelgaard CASPERSEN, 2007), programming can be "understood as the process of inventing suitable structures" to solve a problem at hand. In this context, the program represents a set of well defined instructions expressed in a computer language, describing how to carry out a specific task (LOPES and GARCIA, 2002; TAYLOR, 1982). In order to execute the program as expected, the instructions must be sequenced and expressed in an adequate form.

The adequate form here are mainly defined by two rules: syntax and semantics. Therefore, in order to be valid, a program must first follow the syntactic rules of a programming language(TENNENT, 1976). Then, the syntactically valid program must also obey the semantic rules defined by the language. The semantics of a programming language is very important, since it specifies the meaning of each and every command used. In other words, it defines how the computer should interpret the coded program (TENNENT, 1976).

Programming languages can be classified according to their level. The level of a programming language is a measurement of the amount of details the programmer has to give the computer in order to make the program work (SHU, 1986). They can vary from a very low to a high level. Low level languages are closely related to the hardware and its architecture and requires much more detailed instructions. In higher levels, the languages are closer to how we humans express things, usually by text, and also required much less instructions or piece of code to achieve the desired result (TAYLOR, 1982; OUALLINE, 1997).

Figure 2.1 shows a program that reads a number and prints it coded in the Pascal language (left - high level), and the same program compiled to Assembly language (right - low level). Note that part of the resulting Assembly code has been omitted for simplicity. In the figure, it is possible to notice that any human could read and get the idea that something is being read and written by looking at the Pascal code. However, the same cannot be said by looking at the Assembly code. It is a series of instructions aimed directly



```
procedure Ex1();
var a : Integer;
begin
    readln(a);
    writeln(a);
end;
```

```
OUTPUT_$$_EX1:
  pushq %rbp
  movq %rsp,%rbp
  leaq -32(%rsp),%rsp
  movq %rbx,-24(%rbp)
  call fpc_get_input
  movq %rax,%rbx
  leaq -16(%rbp),%rsi
  movq %rbx,%rdi
  call fpc_read_text_sint
  call FPC_IOCHECK
  movw -16(%rbp),%ax
  movw %ax,-4(%rbp)
  movq %rbx,%rdi
  call fpc_readln_end
  call FPC_IOCHECK
  call fpc_get_output
  movq %rax,%rbx
  movswq -4(%rbp),%rdx
  movq %rbx,%rsi
  movl $0,%edi
  call fpc_write_text_sint
  call FPC_IOCHECK
  movq %rbx,%rdi
  call fpc_writeln_end
  call FPC_IOCHECK
  movq -24(%rbp),%rbx
  leave
  ret
```

Figure 2.1: *A program that reads an integer and prints it coded in Pascal and Assembly language*

to the computer hardware, instructing it how to manipulate the processor registers and memory addresses. Thus, there are a variety of programming languages that can be easy or hard to understand at glance value. Consequently, the programming language can directly interfere in the success of a student.

2.1.1 Programming Learning

Computers, and programming itself, are becoming more and more central to daily life (Vee, 2013). The ability to program is also regarded as literacy to some authors due its relevance in the 21st century society (Vee, 2013; Eshet-Alkalai, 2004; T. Dufva and M. Dufva, 2016). In his work, Seymour Papert (Papert, 1980; Papert, 1996) discussed how it was important to teach kids how to program computers instead of being programmed by them. This was a criticism on the methods of how computers were used in classrooms, as mere repositories of knowledge. Papert believed children should be allowed to use the computer as a tool to construct knowledge and change how they could use it to learn (Papert, 1980).

Thus, in order to materialize his idea, Papert developed a programming language called Logo that could be used to control a virtual turtle (Papert, 1980). The Logo language allowed for an easy way to communicate with the computer "so that learning to communicate with them can be a natural process, more like learning French by living in France"(Papert, 1980). The point was to give control of the machine through the use of programming. However, the literature show there is still some problems regarding the learning of programming by the novice.

Different studies report on the difficulties of learning programming logic: Michael Edelgaard Caspersen (2007), Tan *et al.* (2009), Vainio and Sajaniemi (2007), Lahtinen *et al.* (2005), Gomes and Mendes (2007) and Milne and Rowe (2002). All authors agree that programming is a very complex and highly abstract process, but each of them focus on different aspects of this process.

In his thesis, Michael Edelgaard Caspersen (2007) discusses the challenges and difficulties in learning how to program. To the author, understanding the programming process and how to transfer acquired skills are the main challenges faced by students. Tan *et al.* (2009) findings corroborate Caspersen assumptions. They collected data on undergraduates through an on-line questionnaire in order to identify difficulties and prior experience with programming. In their study, Gomes and Mendes (2007) list a variety of problems ranging from teaching methods to students psychological state. They argue that some students are not motivated enough to study programming, while they were also facing problems with complex language syntax. In addition, the authors state that many students incorrectly employ study methodologies, focusing on memorizing algorithms instead of learning them. Milne and Rowe (2002) investigate students and tutors perception on some programming concepts like pointers, functions and polymorphism. They also conclude that the lack of understanding how a program works is directly responsible for students failure. On the same note, Vainio and Sajaniemi (2007) report on students difficulties on tracing single variables states during programming. Once again, not being able to mentally execute a program is reported as main source of difficulties.

However, Michael Edelgaard Caspersen (2007), Gomes and Mendes (2007) and Lahtinen *et al.* (2005) also bring to the discussion the effect of the programming language on students performance. The complex syntax can become an obstacle to some students and cause lots of confusion during the learning process. Complex and obscure syntax can provide a tremendous cognitive load to the learning process, thus affecting the final performance (Michael Edelgaard Caspersen, 2007). Also, in students point of view, some languages can be easier to understand than others (Lahtinen *et al.*, 2005). And since most languages used on programming courses are developed with professionals in mind, they are not appropriate to a teaching environment (Gomes and Mendes, 2007). As a result, they demand higher levels of memorization.

A direct consequence of these difficulties is the high dropout and failure rate some programming courses face. According to Bosse and Gerosa (2015), in Brazil, most introductory programming courses have a high dropout and failure rate. Their research reports a failure rate higher than 50% at Universidade de São Paulo. This rate includes students who did not achieve the required grade and those who dropped out. Other studies also discuss these high rates as a big concern in programming teaching research (Tan *et al.*, 2009; Rapkiewicz *et al.*, 2006; Lahtinen *et al.*, 2005).

Nonetheless, Bennedsen and Michael E. Caspersen (2007) report a global pass rate of 67%. Although they reported an 33% of failure rate, they could not provide any insight if the failure rates represented only effective students, i. e. students who did not abandon the course. In a follow-up study, Watson and F. W. Li (2014) confirmed these rates through a literature review. They also identified an approximate global dropout rate of 3%. However, both studies show that these rates can vary widely, with existing cases of 95% failure rate (Bennedsen and Michael E. Caspersen, 2007). The authors also discuss that although 67% is not alarmingly low, it must be taken with caution since the rates of enrollment and retention in computer science as a whole are known problems.

Although programming is a very complex task, programming skills are currently in high demand. Different studies have been done in an attempt to find a solution to the problems presented above. Among them, there is the visual programming paradigm which will be discussed in Section 2.2. Still, governments worldwide are trying to implement introductory programming courses as integral part of elementary and high school curricula. This movement is powered by the idea of computational literacy. Different authors argues on how should this literacy be called (Vee, 2013; T. Dufva and M. Dufva, 2016), but in this work we will use the term Computational Thinking as defined by (Wing, 2006).

2.1.2 Computational Thinking

The core concepts present in the idea of Computational Thinking were first identified by Seymour Papert (Papert, 1980). The author believed the computer could be a tool to pave new ways of thinking and learning, beyond the computer. Thus, the computer would stop being only a machine to become a source of social transformation. Jeannette Wing (Wing, 2006) coined the term Computational Thinking (CT) as the ability to employ computer science concepts to solve problems in everyday life. According to the author, CT is a fundamental skill that everyone should be eager to learn.

In an attempt to provide a clear definition, Aho (2012) presents CT as the "thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms"(Aho, 2012). In their study about bringing CT to K-12, Barr and Stephenson (2011) discuss the problem of multiple definitions of CT present in the literature. The authors argue that CT definition must be clear enough so that its application is straightforward. Thus having K-12 in mind, they define CT as "an approach to solving problems in a way that can be implemented with a computer".

Recognizing the multitude of definitions to CT and also presenting a new approach to the idea, Annette Vee (Vee, 2013) brings everything related to CT, digital (Eshet-Alkalai, 2004) and code literacy (T. Dufva and M. Dufva, 2016) under the umbrella of Computational Literacy. In her work, the author shows how all those definitions points to the same direction: programming as a literacy. Since computer and programming are inevitable elements in the society, constructing programs has become a powerful mode of written communication (Vee, 2013). To the author, being a literacy means that programming "can be used for creative, communicative and rhetorical purposes" (Vee, 2013). Therefore, Computational Literacy is the set of abilities to break a complex process into small procedures and then express these procedures through code, which can be read by a human or computer (Vee, 2013).

Vee (2013) argues that programming is a human facility with its own symbolism that allows people to communicate through it. This understanding aligns directly with Papert's idea of using programming as a source of transformation, giving the programmer control over the machine (Papert, 1996; Papert, 1980). Although, Vee (Vee, 2013) uses the term Computational Literacy to talk about CT, in this project the term CT will be used as a synonym to Computational Literacy. This choice is motivated to the fact that CT is a more used term in the educational literature.

As shown above, CT goes beyond just programming. It is about applying the strategies and models used to solve problems algorithmically to your everyday life. Thus, Wing (Wing, 2006) divides it into a set of 11 processes such as abstraction, algorithm design and decomposition. The objective then is to apply one of these processes to a given problem, not matter if the solution will by employed by a computer or human. The processes were later expanded by other authors to include more granular concepts (Hsu *et al.*, 2018; Grover and Pea, 2013; Barr and Stephenson, 2011).

However, the literature shows that the main used strategy to foster CT is programming (Grover and Pea, 2013; Hsu *et al.*, 2018). But as presented in this section, programming teaching and learning is not an easy endeavor and students may have lots of difficulties. So, a solution employed in many studies devoted to programming teaching or CT is the visual programming paradigm. As the name suggest, it is the use of visual elements instead of text to construct a program. Programming languages that adopt this paradigm are known as Visual Programming Languages. The paradigm will be discussed in the next section by means of its associate programming languages.

2.2 Visual programming languages

Visual programming languages (VPL) come as an answer to the difficulties students face when learning to program with traditional text-based languages. It is, as said by Glinert and Tanimoto (1984), a departure from current programming styles in order to make programming more accessible. And as such, VPL aims to mitigate the steep learning curve related to programming since it has proven to be rather difficult to learn (Boshernitsan and Downes, 1997). In this regard, studies have shown very positive effects of VPL in the learning process like engagement and self-efficacy for instance (Booth and Stumpf, 2013; C.-K. Chang *et al.*, 2017; Brandao *et al.*, 2012). In this context, self-efficacy refers to the student's confidence on his capacity to solve a problem (Flammer, 2001). Moreover, VPL are the main used teaching media to develop Computational Thinking according to different reviews of the literature, specially due to its low cognitive load (Hsu *et al.*, 2018; Grover and Pea, 2013; Lye and Koh, 2014). Besides that, the Horizon Report also identified visual programming languages as important tools to foster programming skills (Freeman and Hall Giesinger, 2017).

A visual programming system is a programming environment that implements a VPL (Nascimento *et al.*, 2019). The presence of these systems dates back to 1960, when Sketchpad was developed (Glinert and Tanimoto, 1984). The system allowed users to create simple 2D objects using basic geometric primitives like lines and circles along with operations such as copy and user-defined constraints (Boshernitsan and Downes, 1997). Even though the creator of Sketchpad himself did not conceived it as a programming tool, there is an understanding that it is possible to teach basic programming concepts through geometric algorithms (Oliveira Brandão and Isotani, 2003). Another system created during the same period was a data-flow language that allowed for debugging and execution of data-flow diagrams (Boshernitsan and Downes, 1997). The first visual system developed with programming in mind was Pygmalion, developed by David Smith as a PhD project (Glinert and Tanimoto, 1984; Boshernitsan and Downes, 1997).

Pygmalion was an icon-based programming environment which employed the programming-by-example concept. In this concept, students would show the system how to perform a specific task, guiding the computer step by step. Then, the system could generate a program that would perform that task for general cases. Figure 2.2

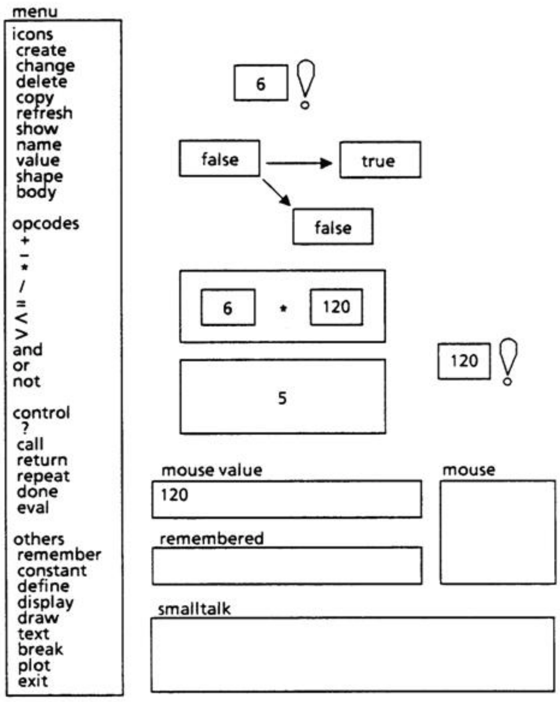


Figure 2.2: *Pygmalion's user interface and example code.*

shows the Pygmalion's user interface with its menus and programming area. The figure presents an example a user demonstrating to the system how to calculate 6!. After all the steps are presented to the system, it can then calculate the factorial of any number. It is possible to notice that the visual programming language used by Pygmalion had support to flow control structures and loops. It was also possible to create your own icons and functions.

Another relevant system to the development of VPL is Pict by Glinert and Tanimoto (1984). Pict was the first system to allow for a complete visual programming experience. According to the creators, all system interactions would be through the mouse without the need for a keyboard. Besides the visual elements, Pict would also use audio cues to indicate the correct usage of a command or errors. Visually, the code constructed inside Pict would resemble a directed graph, with each node representing a programming structure. The fragment of code shown in Figure 2.3 represents a piece of the factorial function. The presence of different color schemes is another contribution given by Pict.

Modern systems like Alice3D, Scratch, Chentry and iVProg improve or expand the concepts developed by the cited tools. Alice3D, for instance, focus on game building and animations in a 3D world. Scratch follows a similar approach but in a 2D stage. iVProg uses VPL affordances to enable students to build general purpose software. All these systems will be discussed in Section 2.2.2.

2.2.1 Formal definition of Visual Languages

Visual language refers to any graphical system equipped with syntax and semantic rules, where the language alphabet is consisted of visual symbols (Shu, 1986; Boshernitsan and

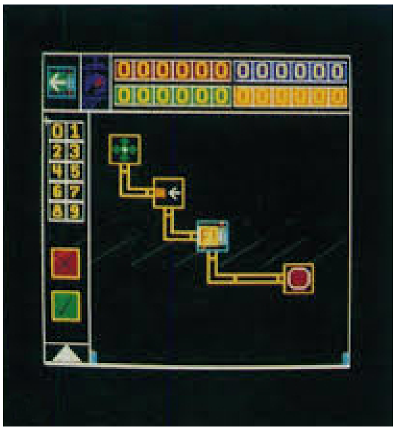


Figure 2.3: *Fragment of code show part of the factorial implementation in Pict*
Source: Adapted from Glinert and Tanimoto *(1984)*

Downes, 1997). There are different ways to categorize the visual languages (Boshernitsan and Downes, 1997) like, for example, the one defined by S. Chang *et al.* (1986, p. 3). In their work, the authors divide the visual languages in four categories:

I. Visual Programming Languages

II. Iconic and Visual Information Processing Languages

III. Languages supporting visual interaction

IV. Visual information processing languages

Each category indicates how objects belonging to the language are composed. Categories I and III have logical objects without a clear visual representation that are mapped to one. Categories II and IV have naturally visual objects which then have a logical meaning imposed over them.

Thus, the formal definition of a visual language L_v can be expressed as the triple $L_v = (ID, G_0, B)$ (Boshernitsan and Downes, 1997), where:

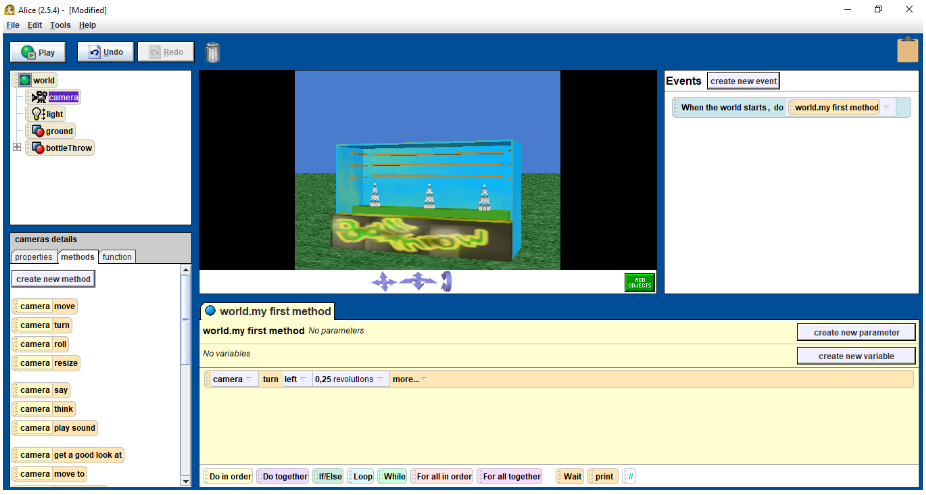
- ID is the image (or icon) dictionary,
- G_0, the language grammar and
- B, the domain-specific knowledge base.

Following the characteristics defined by S. Chang *et al.* (1986), it is possible to infer that a visual language is a mapping of domain-specific objects to visual elements. Taking VPL as an example, ID is defined as the mapping $X_m \mapsto X_i$, where X_m represents the logical object meaning and X_i its visual representation. Then, G_0 specifies how to construct composite visual objects through spatial operations with elementary objects from ID (Boshernitsan and Downes, 1997). Here, spatial operation defines how the user can move the objects around since the language is visual in nature. Thus, a visual sentence is an arrangement of pictorial elements related to two or more dimensions (Ferrucci *et al.*, 1998). In this context, the domain-specific knowledge base B is responsible for giving meaning to the composite objects. By analogy with textual programming languages, ID represents the alphabet, G_0 the syntax rules and B the semantic rules.

2.2.2 Visual Programming System

A number of visual programming systems have been developed through the years. Since the Pict system, others interpretation of visual programming have been implemented, some even mixing textual and visual. In this section, four systems will be presented: Alice3D, Scratch and iVProg. They are all visual programming systems with different approaches on how should the user build the code.

Alice is a visual programming environment developed by the Carnegie Mellon University where the user can create animations and games. It has been developed since 1992 and currently is in its third version. It is a standalone Java program that includes a variety of graphical assets to be used by students. The idea behind Alice was to create a novice-friendly 3d simulation system that supports exploratory programming (Conway, 1998). In Alice, the student has access to a 3d environment that can be populated with different objects and images. It also includes a set of built-in actions that allows the user to move objects, dynamically create new ones or change their appearance (Nascimento *et al.*, 2019). These actions are presented as visual blocks that can be dragged and dropped to describe the desired behavior. The system is developed around an object-based approach, where each element in the system is treated as an instance of a general class. It is a common practice to use Alice as an introduction to object oriented programming (Nascimento *et al.*, 2019).



When the world starts, do world.my first method

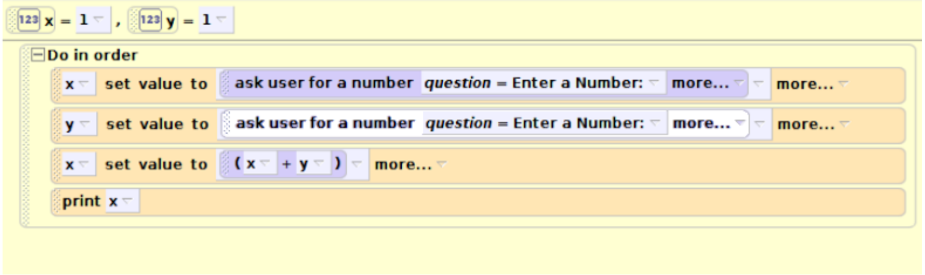
world.my first method *No parameters*

No variables

camera turn left 0,25 revolutions more...

Figure 2.4: *Alice user interface and its windows*

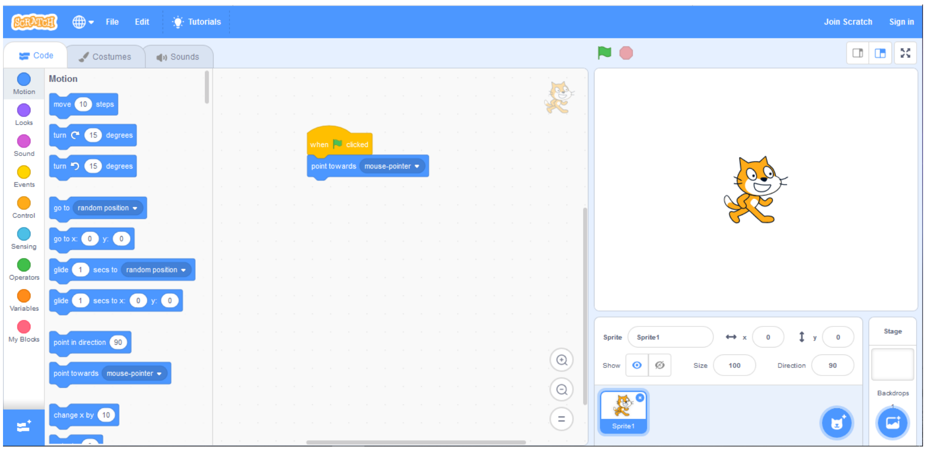
Figure 2.4 presents Alice user interface and its windows. In the top left corner are the objects in the scene window and in the bottom left corner are the selected object methods and properties. The main scene window is in the middle of the screen while the selected object function body is at the bottom of the screen. From the selected object method and properties window it is possible to drag and drop all the actions the selected object can perform. Figure 2.5 shows an example of a program that reads two numbers and prints their sum in Alice.



```
123 x = 1 , 123 y = 1
Do in order
    x set value to ask user for a number question = Enter a Number: more... more...
    y set value to ask user for a number question = Enter a Number: more... more...
    x set value to ( x + y ) more...
    print x
```

Figure 2.5: *A program that reads two numbers and prints their sum in Alice*

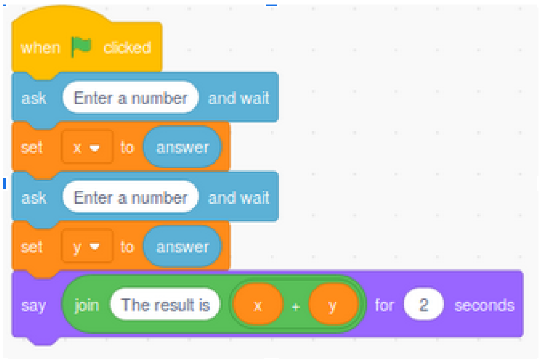
Following a similar design, the Scratch environment is actively developed by the Lifelong Kindergarten Group at the MIT Media Lab. First launched in 2007, Scratch has been under active development since then and has recently launched its third version. Scratch is mainly a web application that also includes free assets like images and audio files. There is also an off-line version that can be downloaded. The system target audience are children and teenagers without programming knowledge and is designed to be ludic and easy to use (Maloney *et al.*, 2010). Scratch is heavily influenced by Papert's constructionism (Maloney *et al.*, 2010; Resnick *et al.*, 2009; Nascimento *et al.*, 2019), in which case the VPL system provides the students with tools to construct virtual stories and worlds. As a consequence, Scratch is mainly used in middle schools projects and with kids, although some experiments in higher education do exist Nascimento *et al.*, 2019; C.-K. Chang *et al.*, 2017. Different from Alice, Scratch provides a 2D stage where users can easily create games and animations by connecting and nesting blocks (Resnick *et al.*, 2009; Maloney *et al.*, 2010). It also provides access to different multimedia sources like web-cams and audio files.



```
when 🏴 clicked
point towards mouse-pointer
```

Figure 2.6: *Scratch main user interface.*

The Scratch user interface is presented in Figure 2.6 . On the left is the list of command blocks divided by categories which can be used to manipulate movable 2D graphical objects called sprites and the stage (Maloney *et al.*, 2010). In the middle is the programming area, where the user drags and nest the blocks to code their algorithm. On the right is the stage, where the user can insert sprites and background elements. In this window is also possible to adjust manually some properties of the sprites like rotation and position. Figure 2.7 presents the same example in Figure 2.5 , but now implemented in Scratch.



```
when 🏴 clicked
ask Enter a number and wait
set x ▾ to answer
ask Enter a number and wait
set y ▾ to answer
say join The result is x + y for 2 seconds
```

Figure 2.7: *A program that reads two numbers and prints their sum in Scratch.*

Chentry is a novel system designed to provide automatic assessment to an online teaching web-platform (www.playentry.com). The platform is a Korean website aimed at teaching children basic concepts of CT (J.-H. Kim *et al.*, 2019). Although it uses a user interface similar to Scratch, it just provides a limited amount of blocks based on the assignment. When creating a project, the teacher can decide which blocks will be visible. The Chentry system supports loops, variables, lists and input/output, such as Scratch (J.-H. Kim *et al.*, 2019). Moreover, the system can export code to the Arduino system and also supports device communication.

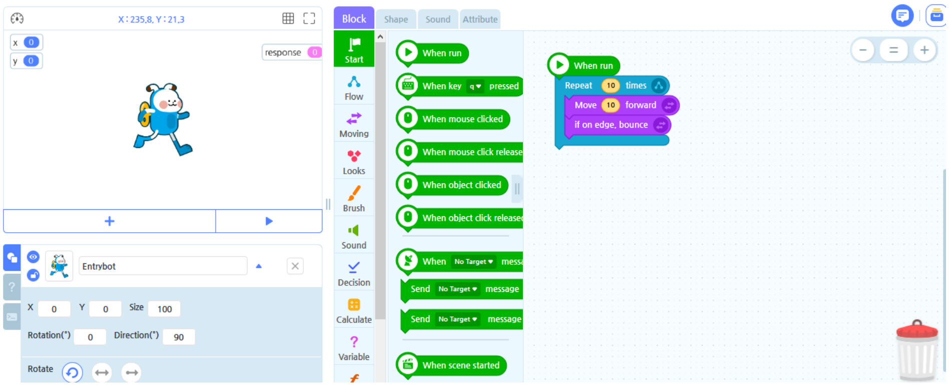
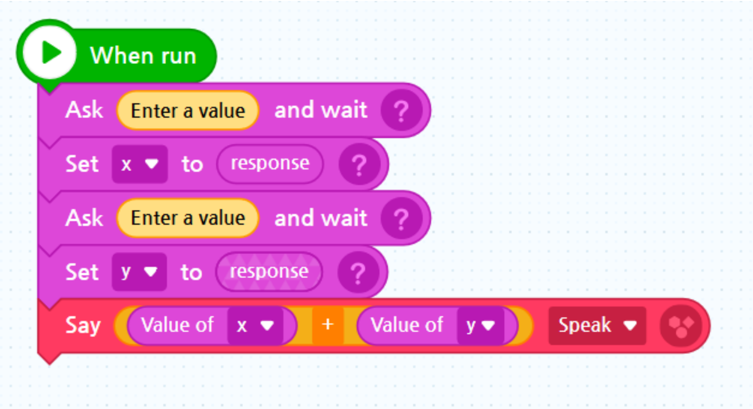


Figure 2.8: *Chentry main user interface.*

Figure 2.8 presents the main interface of Chentry. The system provides a stage, which cannot be edited by the user, that performs the code constructed at the block assembly area on the left. There is also the block box, where all available blocks for the given assignment will be presented. In the figure it is possible to see all blocks since it is the teacher view. The example code implemented using Chentry is shown in Figure 2.9

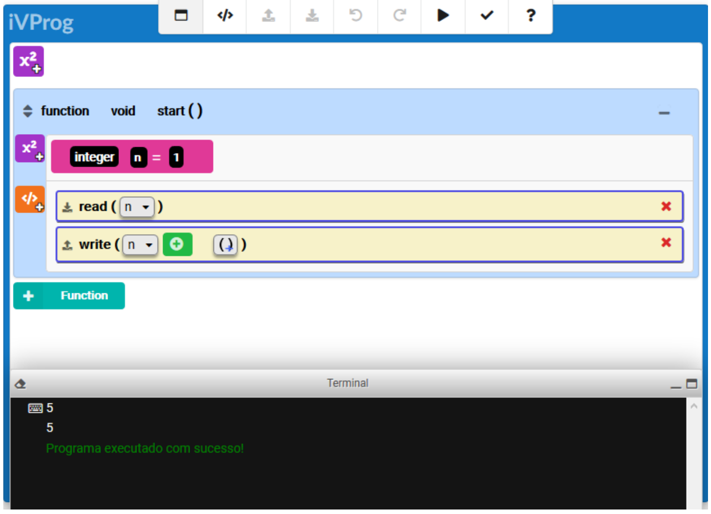
Initially conceived as a modification of Alice, iVProg is now an independent system developed by the Laboratory of Informatics in Education (LInE) at Universidade de São Paulo. Its first version was launched in 2009 and was distributed as a Java applet Nascimento *et al.*, 2019. However, since 2015 the system is implemented with HTML5 technology to be fully web-compatible and portable. The motivation behind iVProg was to create a system to aid programming teaching and learning that could also be easily integrated into learning management systems. Moreover, iVProg ships with automatic assessment capabilities. As



```
When run
Ask Enter a value and wait ?
Set x ▾ to response ?
Ask Enter a value and wait ?
Set y ▾ to response ?
Say Value of x ▾ + Value of y ▾ Speak ▾
```

Figure 2.9: *A program that reads two numbers and prints their sum in Chentry*

Nascimento *et al.* (2019) report, iVProg is mainly used in high school and undergraduate courses.



```
function void start ( )
integer n = 1
read ( n ▾ )
write ( n ▾ ( ) )
```

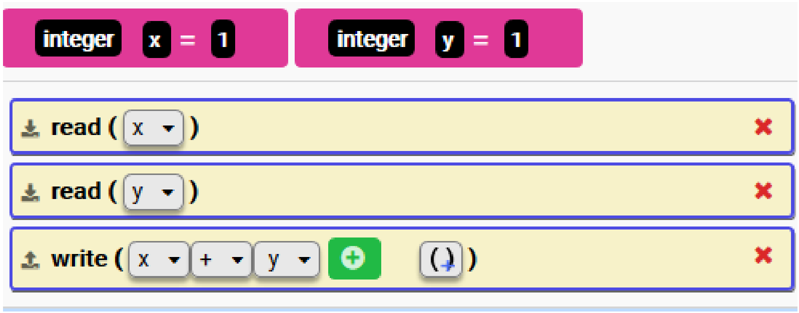
```
5
5
Programa executado com sucesso!
```

Figure 2.10: *A program that reads two numbers and prints their sum in iVProg.*

The iVProg user interface is presented in Figure 2.10 . The top bar presents a list of commands related to code execution, assessment and help. iVProg also allows the user to switch between a visual and textual representation of his code in a language inspired by the PortugolStudio language[1]. The buttons on the left of the screen allows the user to create variables in the global and local context and also insert commands to the function body. The resemblance between the Alice code and iVProg is still very noticeable. Moreover, the system also presents a terminal window responsible for managing input and ouput as presented in the figure. It is also possible to create other functions with different return types and parameters. Figure 2.11 shows the same example code for the other visual systems now implemented in iVProg.

All the presented tools implement the visual programming paradigm in different approaches and aimed different groups of users. In their work Nascimento *et al.* (2019)

[1] http://lite.acad.univali.br/portugol/



```
integer x = 1    integer y = 1
read ( x )
read ( y )
write ( x + y      ( ) )
```

Figure 2.11: *A program that reads two numbers and prints their sum in Alice*

discuss the suitability of each one of them for some educational contexts like school level and subject. Only Scratch and iVProg are web-compatible but only iVProg can be fully integrated in a learning management system. Also, iVProg is the only tool present in the literature that have support for some sort of automatic assessment of code. Automatic assessment of programs will be discussed in Section 3.1.

2.2.3 Programming learning and Visual Programming

As mentioned earlier, the visual programming paradigm represents an advancement towards a more accessible programming learning. Through its visual elements, VPL provides a way to express code closer to how humans think (Lye and Koh, 2014). Reinforcing this idea, Glinert and Tanimoto (1984) claim that the way humans mind process information is often multidimensional and visual. Thus, VPL allows students to program through objects that are closer to how they process information. This way they can concentrate on the basic aspects of their solution instead of complex syntactic structures. As a consequence, improvements in programming learning is reported by different studies.

Glinert and Tanimoto (1984) report an experiment where graduates and undergraduates were asked to use the Pict system to develop some programming assignments. The students found Pict's visual system much easier to use than textual programming. Also, some participants believed that they had just created a program without having to learn a programming language. Moreover, students regarded the system as a useful tool to understand program execution and debugging.

Investigating the empiric evidence for and against the claims made about VPL, Whitley (1997) findings indicates that VPL is better when dealing with problem-solving situations. According to the author, the reason is that the visual can express information in a more consistent and organized way. He also claims that the benefit of using a VPL grows as the complexity of the problem grows. However, he also points out that efficacy of visual notation depends on the task to be performed, where the task may require more detailed information from the visual system which is not always possible. Since he performed his research in 1997, he was also concerned about the limits imposed by the graphics systems of the time and other problems that are already overcome.

In his study, Sykes (2007) developed an experiment with three groups of computer science students, two of them programming with C (comparison group) and one with Alice3D (experimental group). The comparison among the groups showed that Alice3D eliminated syntax errors and was able to motivate students. They also noticed that the visual nature of Alice3D allowed students to focus on problem-solving skills instead of syntax and compilation errors. Moreover, when comparing all the groups, the experimental group had a better performance overall.

Likewise, Meerbaum-Salant *et al.* (2010) developed an experiment with a group of middle school students where Scratch was used to teach programming concepts. They concluded that the use of a VPL improved students internalization of CS concepts like loops, message passing and concurrency. Also, they applied a combination of Bloom's and SOLO taxonomies to measure students cognitive level after the experiment and found indications of improvement. On the same note, Sáez-López *et al.* (2016) describe a case study with 107 primary students using Scratch. The case study was developed in a 21h sessions integrated in arts and science class. Their findings align with Meerbaum-Salant *et al.* (2010) while also presenting advancements in other aspects related to logic and mathematics. They also report an increase in motivation and enthusiasm.

C.-K. Chang *et al.* (2017) attempted to identify how VPL can affect motivation in a data structure course. They analyzed the data of two groups, one experimental and another control. They employed a pretest and post-test method to collect data on student motivation. The experimental group had classes using the tradition method followed by VPL, the control group had it the opposite way. Results show that VPL can improve students motivation and outcome, which align with Meerbaum-Salant *et al.* (2010) and Sáez-López *et al.* (2016). Also, the order in which VPL is introduced, before or after traditional text programming, seemed irrelevant.

Brandao *et al.* (2012) performed an experiment with undergraduates in mathematics through a blended learning approach. Their study attempted to identify how iVProg could improve students problem solving with algorithms. Also, they investigated how a VPL can affect students learning of a text-based programming language. The results indicate that students found the visual language much easier to learn when compared to C. Students also reported that the iVProg allowed them to focus on the solution instead of the issues related to syntax and other language structure. The authors also claim students are more motivated when using VPL and that iVProg can smooth text-based programming introduction.

Additionally, Booth and Stumpf (2013) developed a study with 11 students, where each student would have 2h in lab with a textual and visual environment. Their findings also confirms the claim that students perform better and are more motivated when using a VPL system. Their results are also aligned with C.-K. Chang *et al.* (2017) regarding textual and visual combination. Students also reported that they felt visual programming was a good introduction to programming concepts but that once they understood the textual approach was better. This is also related to Brandao *et al.* (2012) findings that a VPL system can make textual programming easier.

Another approach that can yield good results in an introductory programming course is automatic assessment of programs. The literature reports a number of benefits for students, lecturers and instructors. The following section will discuss automatic assessment tools and how they work.

Chapter 3

Related Work

3.1 Automatic Assessment tools

The nature of programming makes it possible to employ automation in most cases. In the industry, it is a common practice to use automated testing, deployment and integration of huge software modules (Osherove, 2009). However, in an educational context, the use of tools to automatically assess the students solutions dates back to 1960 (Douce *et al.*, 2005; Hollingsworth, 1960). Automatic Assessment Tools (AAT) can not only assess numerous solutions at once but also provide quick feedback (Ala-Mutka, 2005). These features can be very valuable for both students and lecturers (Ihantola *et al.*, 2010). To Pears *et al.* (2007), AAT are adaptable and have diverse applicability in a course context. They can be used for summative or formative assessment, or even an impartial form of evaluation. To the student, AAT provide a valuable and quick feedback, allowing them to learn through their coding mistakes (Pears *et al.*, 2007).

Instead of using compilers and text editors, the first AAT was developed to assess Assembly code written on punch cards (Douce *et al.*, 2005). The assessment process consisted in checking if the outputs stored in specific memory address were the correct ones. The only possible results at the time was “wrong answer” or “program complete” (Douce *et al.*, 2005). With the advancements of computing, new system were developed and still in 1960, a new AAT designed for Algol was created. The system was dived in three parts: a module responsible for the assessment, another to keep track of running time and the last one to maintain a grade book (Douce *et al.*, 2005).

According to Douce *et al.* (2005), the advantages of using an AAT consisted of a good use of tutor and computer resources, already enabling a great number of students to learn programming. Also, the authors claim that the concern with plagiarism and harmful code was already present. Other tools were developed following the evolution of operating systems (OS). These tools would use the preexisting tools and utilities provided by the OS like testing engines and programming environment (Douce *et al.*, 2005). Moreover, with the emergence of these tools the concept of dynamic and static assessment took shape.

For example, the first system to provide such hybrid approach was developed in 1989 as reported by Douce *et al.* (2005) and Isaacson and Scott (1989). Besides checking the program functionalities, the tool also verified if the code followed the code style expected by the lecturer. In her review on AAT, Ala-Mutka (2005) analyze them according to the nature of the assessment performed by the tool: dynamic or static. It is important to mention that an AAT can provide assessments of both nature. Static assessment features the evaluation of the code without the need to execute it. This type of assessment is able to verify aspects ranging from coding style to code structure and other software metrics. In dynamic assessment, the evaluation is focused on functionalities and the program behavior during runtime. Dynamic assessment is often run inside "jails" in order to protect the host system from malicious code (Ihantola *et al.*, 2010).

3.1.1 Static and Dynamic Assessment

The automated assessment tools for programs have, in general, two types of assessment: static and dynamic. In an educational context, each type of assessment aims to analyze different aspects of the student solution. According to Arifi *et al.* (2015), in static assessment the program structure and content are examined to collect the required information. This examination is done without the need to execute the code (Ala-Mutka, 2005). On the other hand, dynamic assessment is characterized by the execution and verification of functionalities. As a consequence, dynamic assessment usually requires some security measure to ensure the safety of the system running it (Ihantola *et al.*, 2010).

Static Assessment

Static assessment consists in the static verification of the student code. This is done in order to check if the code provided follows the style rules or expected metrics. It is important to note that any static verification requires the code to be sintatically valid (see Section 2.1). This approach has a set of methods, many of them present in the industry (Ala-Mutka, 2005; Arifi *et al.*, 2015):

- code style: this approach analyzes the program readability in order to measure his quality (Arifi *et al.*, 2015). Some other aspects like line spacing and variable names are also considered. Moreover, unused variables, usage of global variables, type conversion and other language features are also considered(Ala-Mutka, 2005).

- programming errors: although most errors are found during runtime, there are some heuristics that allow for some of them to be identified in a static verification (Ala-Mutka, 2005; Ihantola *et al.*, 2010). This includes semantic errors (see Section 2.1) like incompatible parameter type when calling functions or performing operations. Another problem very common with novices that can be spotted with static assessment is never-ending loops (Arifi *et al.*, 2015). Also, static analysis can be used to identify potential malicious or problematic code and block its execution.

- software metrics: assessing a source code statically can also measure its complexity (Arifi *et al.*, 2015). The complexity of a code can indicate students performance or misunderstanding of concepts and, as such, it should be used a sound pedagogical purpose. Besides that, it is possible to measure comments frequency and number of lines of code, i. e. lines that represent programming instructions, the number of statements and their size (Arifi *et al.*, 2015). The size of a statement is represented by average number of operators and operands.

- design: the assessment of the design (or structure) of a code is employed to assert that the given code conforms to the required interface or structural requirement (Ala-Mutka, 2005). Thus, a student code can be compared to a reference solution and graded according to the similarities. Also, the same approach can be used to detect possible plagiarism cases.

However, static assessment cannot provide reliable information regarding code behavior during runtime. For example, if the code provided can perform the expected behavior during its execution. Thus, dynamic techniques can be used to assess this aspect of the code.

Dynamic Assessment

According to Ala-Mutka (2005), it is not possible to consistently and thoroughly assess a student's code without automation. This is specially true for dynamic assessment since even a small piece of code can have numerous execution paths (Ala-Mutka, 2005). To Arifi *et al.* (2015), dynamic assessment consists in executing a program with a battery of test-cases. Each test-case can be designed to check an execution path and how the program allocate and use processing power and memory (Ala-Mutka, 2005). Dynamic assessment also have a set of approaches that can be divided into the following methods (Ala-Mutka, 2005):

- functionality: this is the most common form of dynamic assessment. In most cases, the system uses a black-boxing approach in which the program is analyzed as a single entity. After that, the outputs generated by the program are examined and compared to the expected ones which there is only two possible results: "correct" or "incorrect" (Arifi *et al.*, 2015). This assessment strategy is usually referred to as output matching, which consists in comparing the program output text to the model output text (Ala-Mutka, 2005). Another strategy used to assess program functionality is unit testing (Ihantola *et al.*, 2010). Unit testing is a piece of automated code that invokes a unit of work in the system and then verifies if a given assumption about that unit of work is true (Osherove, 2009). In this case, it is possible to evaluate the functionality of smaller entities instead of complete programs. The unit of work can be a function or any other block of code depending on the language used.

- efficiency: likewise the functionality evaluation, efficiency depends on a set of test-cases. However, instead of checking outputs or assumptions about the program, the system measures the program behavior during execution (Ala-Mutka, 2005). Usually the measurements consists in CPU time or clock, how many times a block or statement is executed. This approach normally requires a model solution to compare the values measured from the student's solution. Moreover, memory usage can also be tracked and used as part of the assessment (Ihantola *et al.*, 2010).

- testing skills: some system allows for students to upload their own test-cases. The student program is run against his test-cases and the assessment is done on their quality. The automated capabilities of the system are made available to the students so they can practice test designing and think about their own code (Ala-Mutka, 2005).

There are also some approaches the use a mixed-methods strategy. In some cases, it is possible to specify moments during the program execution to run an specific assessment (Ala-Mutka, 2005; Ihantola *et al.*, 2010). Nonetheless, a set of test-cases are still required. This approach can be useful when evaluating a program memory allocation strategy, to make sure the memory being allocated is released at the right moment during execution (Ala-Mutka, 2005).

3.1.2 Improving Automatic Assessment Methods

As already said by Ala-Mutka (2005), Ihantola *et al.* (2010), and Arifi *et al.* (2015), dynamic assessment through output matching is the most common approach used. Nonetheless, this approach does not tolerate deviations from the expected output which can be very unforgiving when taking the student into consideration. It is very common to a human instructor to ignore misspelled words or to identify synonyms, but the AAT may be able to do so (Ala-Mutka, 2005; Ihantola *et al.*, 2010). Since the system evaluates the outputs generated by the student's program, any extra character or whitespace can be enough to evaluate a program as "incorrect" (Ihantola *et al.*, 2010). There are some workarounds that can be used to remove whitespaces, punctuation marks and accent marks, but it is not easy to eliminate the synonyms and other writing styles problems. For example, the expected output can be "the sum of the two numbers is 5" but the actual output is "The sum is 5" which would be assessed as incorrect.

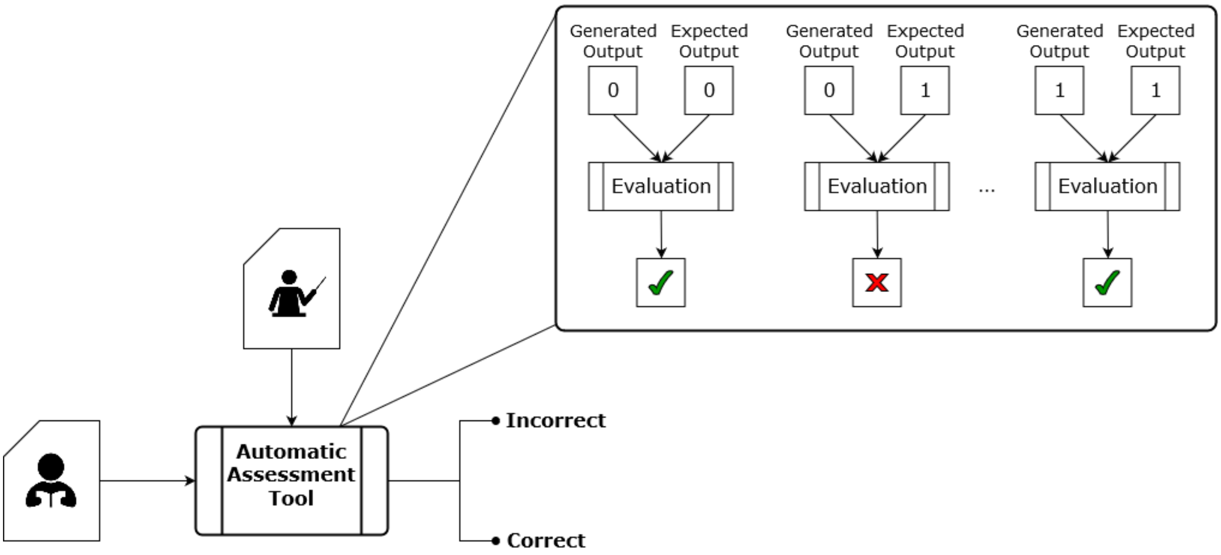


Figure 3.1: *Dynamic assessment through output matching*

Figure 3.1 shows how output matching works. The student submits a source code that is executed using the teacher's test-cases. Each test-case consists in a set of inputs and expected outputs. The AAT then runs the student's program against each test-case and its respective inputs. An evaluation module then checks if the output(s) generated by the students program matches the one present in the test-case. Note that the output must be strictly equal to the expected one. If they do match, the test-case is labeled correct and incorrect otherwise. The student's solution will be evaluated as correct if and only if all the test-cases are labeled as correct. In the case presented in Figure 3.1 , the solution would be evaluated as incorrect since it has at least one test-case labeled as incorrect.

Besides the methods described by Ala-Mutka (2005), there are other alternatives to mitigate the output matching problem in the literature. For instance, C. K. Poon *et al.* (2016), Yu *et al.* (2017), and Chung Keung Poon *et al.* (2018) propose a token pattern approach which consists in a pre-processing step in order to identify key elements of the output. They argue that "most of the deviations from expected outputs are associated not with individual characters, but with groups of characters such as words and numbers" (Yu *et al.*, 2017). So, the output text is decomposed into tokens that represent an "atomic" piece of meaningful information (Yu *et al.*, 2017). The idea is that the system could automatically identify how relevant each of the tokens are and they match the student's output against this tokens. A token can have different properties like: if it must match precisely the word it represents or not; if the number it represents can have a different precision; if it allows variations of the same word (synonyms); and being optional.

The token pattern approach attempts to give students some freedom, but as reported by Yu *et al.* (2017) and C. K. Poon *et al.* (2016), the properties of each token group must be set manually by the teacher. The systems presented in their study allows for some global configurations in which certain groups can have default properties like whitespace and punctuation.

Some studies focus on the feedback provided to the student, using technology and tools that can aid the student locate errors easier. Araujo *et al.* (2016) employs a method of spectrum-based fault localization (SBFL) and automatic code repair to produce better feedback. SBFL is explained by the authors as a technique that relies on "program spectra: program traces that reveal which parts of the code are active during a failed or successful execution"(Araujo *et al.*, 2016). SBFL dynamically assess the code in order to calculate the likelihood of a given component, a line of code in this context, to by faulty. Like most dynamic assessment, Araujo *et al.* (2016) requires a suite of well-designed test-cases in order to locate the faulty components. The system ranks the each component based on the rank given by the SBFL technique and use this information to generate a feedback.

However, Araujo *et al.* (2016) report that the technique can fail in given some situations where the student solution did not pass at least one test-case. Also, the authors reports that employing SBFL can be costly and, as such, improvements must be made in the process.

Following a similar approach, Gulwani *et al.* (2018) employs clustering instead of a fault localization technique. But, they still focused on feedback generation and code repair. The clustering process uses an idea of dynamic equivalence of code. To Gulwani *et al.* (2018), two programs P and Q are said to be equivalent, $P \sim Q$, if and only if they have the same control flow structures and their variables have the same values, in the same order with the same input. Based on this clustering, their algorithm then attempts to find the minimum number of modifications required to transform a faulty program P' into a correct one.

With the repair information in hand, the system them construct a feedback informing the student which should be changed and to what. It is important to note that the whole system relies on the idea of *wisdom of the crowd* and as such requires a number of correct solutions to be available. For instance, given the code present in the Listing 3.1, Gulwani *et al.* (2018) approach would generate a feedback like: "In assignment expression in line 4, change sum[i] += vec to sum += vec[i]". According to the authors, clustering and repair achieved good results, comparable to a human. They also found some situations where the algorithm could find a valid repair due to unsupported features and some problems with control flow structures.

```
1  int sumVec (int len, int vec[len]) {
2     int i, sum = 0;
3     for (i = 0; i < len; ++i) {
4        sum[i] += vec;
5     }
6     return sum;
7  }
```

Program 3.1: *Example of a faulty code in C*

In their study, S. Li *et al.* (2016) developed a system that employs a random input sample based on a reference solution. The system uses random test generation to produce inputs using the reference solution and then perform symbolic execution with these inputs. The idea is to identify which set of inputs executes most blocks of the reference solution. With this information, the system then executes the student solution with a random selection of the produced inputs. The generated output is then matched against the outpus generated by the reference solution. Besides that, S. Li *et al.* (2016)'s system also verifies if the student solution activates the same paths as the reference solution during symbolic execution.

Experiment results indicates that the approach presented by S. Li *et al.* (2016) can produce good results. However, it is a complex implementation and in the end it still performs an output matching approach. Therefore, the technique is still subjective to all the problems already mentioned. Table 3.1 summarizes the techniques presented in this section and what the aim to solve. Next section will discuss how AAT can affect programming learning and teaching.

3.1.3 Automatic Assessment and Programming

The automatic assessment tools(AAT) can provide a number of benefits to students and lecturers. The possibility of giving quick feedback to students' solution and to manage many students are very attractive. However, the benefits of using an AAT are not limited to these and in some cases the AAT was used as 0a tool to implement a novel teaching methodology.

Reguera and Leiva (2017) developed an experiment with a group of students in an introductory object oriented programming course. According to the authors, the AAT was central to the development of didactic problem sequences. To the students, the quick feedback provided by the AAT was very helpful and made them more motivated towards the

Technique	*Study*	*Improvement*
Token Pattern Approach	Yu *et al.* (2017), C. K. Poon *et al.* (2016), and Chung Keung Poon *et al.* (2018)	Attempts to improve output matching by processing the output into tokens
Fault Localization	Araujo *et al.* (2016)	Employs fault localization to give context to feedback message
Code clustering	Gulwani *et al.* (2018)	Clustering of equivalent code to better identify errors
Code Repair	Araujo *et al.* (2016) and Gulwani *et al.* (2018)	Code repair as a source of context to improve feedback messages
Random Test Generation	S. Li *et al.* (2016)	Improve AAT with automatic test generation through a reference solution
Symbolic Execution	S. Li *et al.* (2016)	Improve code evalution by checking all execution paths and the quality of test-cases

Table 3.1: *Summary of techniques designed to improve AAT*

course. Moreover, the increased motivation also resulted in better grades when compared other classes in past years.

Lappalainen *et al.* (2017) investigated how AAT can affect students results during exams. The authors claim that programming courses should have computer-based exams instead of text-based. In their study, they analyzed different groups of students' scores when sitting exams with or without a computer. The group that sit the exam with a computer used an AAT and had a clear advantage when solving more complex problems. The group using AAT also produced less erroneous code. Besides that, they claim that automated assessment gives reliable results when compared to manual assessment. However, the design of quality test-cases can be very difficult as mentioned by S. Li *et al.* (2016).

In their study, D. M. D. Souza *et al.* (2015) developed a web-based AAT called ProgTest in order to investigate the effects of the feedback provided by the tool. The experiment consisted a group of 34 undergradute students of an introductory programming course. The students found the constant and concrete feedback provided to be useful for developing better comprehension of their code and analysis skills. In the instructor point of view, the AAT was a viable way of identifying students difficulties regarding the concepts discussed during the course.

A study attempting to investigate student performance when using AAT is reported by Prather *et al.* (2018). They performed an intervention in two computer science courses on programming: an introductory and an advanced course at a university in Argentina. Both courses were taught using Haskell and the study employed the AAT called Mumuki developed by the university. Their findings indicate that the use of an AAT can reduce the dropout and failure rates since the immediate feedback provided by the tool can help students find errors and correct them on their own. Another important finding was that by using automated assessment the students can have the freedom to pace their study.

Likewise, Wilcox (2015) presents the effects of introducing an AAT in an undergraduate course through the years. The author claims that the automation of the grading process can significantly save resources in introductory courses without impacting them negatively. The AAT can save resources like time spent grading exercises and money paid to teaching assistants to grade tests. According to the author, "the benefits of automation are both tangible, such as higher exam scores, and intangible, such as increased student engagement and interest" (Wilcox, 2015).

Following a similar approach, Cardoso *et al.* (2018) investigated how the use of an AAT could impact students motivation and engagement. They developed an experiment with 318 students of an introductory programming course during the year of 2017-2018. As a result, students really enjoyed using an AAT and found the quick feedback provided to be useful to improve their solution. Also, the majority of the students wished there were more exercises to be done through the AAT.

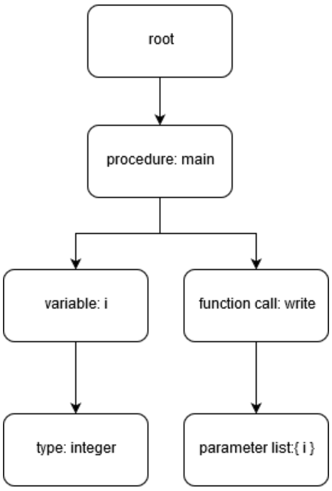
Maguire *et al.* (2017) also developed a study on the effects of AAT and programming learning. The objective regarding AAT was to identify if it could improve students outcome. The authors then designed an intervention to analyze the effects of an AAT on students' performance. Their findings indicate that an AAT can not only improve students grades but also reduce failure rates. Their results aligns with those found by the already mentioned studies like Prather *et al.* (2018). Also, they found that AAT saved a considerate amount of labor on the part of lecturers and demonstrators, which also confirms Wilcox (2015) findings. Besides that, the authors claim that the quick feedback helped students to develop better programming skills.

As shown in this section, different studies shows the benefits of AAT to teaching and learning to program. The tool can not only help the student improve their grades and knowledge but also help the teacher to focus on didactic and pedagogical matters. The automation can reduce the amount of labor required from the lecturer and other resources. Then, how can one join the benefits of visual programming language and automatic assessment of programs in a single package?

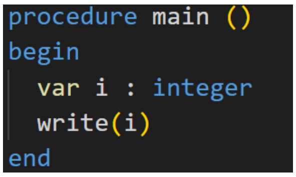
As cited before, iVProg is a visual programming tool that also offers automatic assessment, however it suffers all the basic problems related to output matching. Moreover, there is also Chentry (J.-H. Kim *et al.*, 2019), another visual programming environment that perform static analysis in order to match a given solution to a reference solution. Next section will discuss this project proposed solution to improve visual programming languages automatic assessment capabilities using iVProg as an example.

3.2 Abstract Syntax Tree

An Abstract Syntaxt Tree (AST) is usually the output of a syntactic analysis tool, very relevant for compiled languages. The AST is, in this case, usually associated with a textual code for programming language. According to Falleri *et al.* (2014), an AST is a labeled ordered rooted tree where the nodes can have a string value. In an AST, the label of a node represents its name according the language grammar production rules (Tennent, 1976). These production rules dictates how each syntactic structure should be represented in the tree. Each node value corresponds to actual tokens in the code like function definitions, variable declarations or literal values. The nodes also encode the flow-control structures in the code and information about the data types present in the language.



(a) *AST representation of the code from language L*



```
procedure main ()
begin
  var i : integer
  write(i)
end
```

(b) *Textual representation in language L from the AST in Figure 3.2a*

Figure 3.2: *Illustration of connection between source code and AST*

An AST T can be formally defined as $T = \{t : \ t\ is\ a\ node\}$, where a tree T has one root node denoted by $root(T)$ (Falleri *et al.*, 2014). Following the author's definition, each node $t \in T$ must have a parent $p \in T \cup \emptyset$ (denoted by $parent(t)$) where only $root(T)$ has $\emptyset$ as parent. The children of a node t is denoted as $children(t)$. Figure 3.2 shows an example of AST for the generic language L. Besides representing the relevant information about the code and its structure, an AST can also be executed by interpreting it (Kalibera *et al.*, 2014). Also, an AST can also be transformed into other AST by using specific algorithms (Falleri *et al.*, 2014). This allows, for example, the transformation of an AST of one language to another.

Chapter 4

Visual Code Assessment Tool - VCAT

This chapter describes the solution developed in this research to enable different visual programming languages to use automatic assessment features. The solution is divided into three parts: the model, the instantiation and application. In this chapter it will be discussed the model, the following chapters will discuss the instantiation and application respectively. Here it will be presented the model and the purpose of each component. Additionally, it describes how the components implementation relates to the theories and concepts discussed in previous chapters.

4.1 VCAT's Architecture

The solution presented in this section joins visual programming and automatic assessment of programs. As explained in Section 2.1, programming is a challenging skill to acquire. Among the difficulties faced by students, literature shows that the complexity of textual programming languages is one of the most common. Visual programming languages were designed as an alternative to traditional programming and to make programming more accessible (J.-H. Kim *et al.*, 2019). However, as presented in Chapter 2, most visual programming languages do not lack automatic assessment features that allows students faster feedback and autonomy. The only exceptions were iVProg and Chentry which offer an automatic assessment features.

Still, both of them have limitations in what they can evaluate. For instance, iVProg only offer the standard implementation of output matching with all the problems associated with it. In Chentry's case, the tool performs a static evaluation based on the logs produced during code construction. The evaluation consists in comparing the logs produced by the student with the logs of the reference solution. Although it is a valid form of assessment, it does not guarantee that the code functions as expected (Ala-Mutka, 2005).

So, in order to allow other visual programming languages easier access to automated assessment features and improved feedback, this work proposes the visual code automatic test(VCAT). It is a model of automated assessment of code for visual language designed to allow different visual programming languages to use a common core that provides automatic assessment and improved feedback. VCAT design is inspired by iVProg's later version, focusing on decoupling assessment and execution logic from the visual programming system. As a result this creates a new software artifact which can be used in other visual programming systems besides iVProg. It is important to note that VCAT's initial version is focused on visual languages aimed towards high-school students and undergraduates in their first steps on learning programming. The initial version is a proof of concept and thus, limiting its scope was necessary to make its development and implementation feasible. As a consequence, the design decisions taken at this moment are aligned with this scope in mind.

Let L_v be a visual language defined as $L_v = (ID, G_0, B)$ where ID represent the visual icons dictionary for the language; G_0 the language grammar rules for combining visual icons or visual objects into new visual objects; and B the knowledge base responsible for giving logical meaning for objects constructed in L_v (see Section 2.2.1). Let $\mathcal{L}(L_v)$ be the set of all valid constructions from L_v and C_{visual} be a valid construction of L_v such that $C_{visual} \in \mathcal{L}(L_v)$.

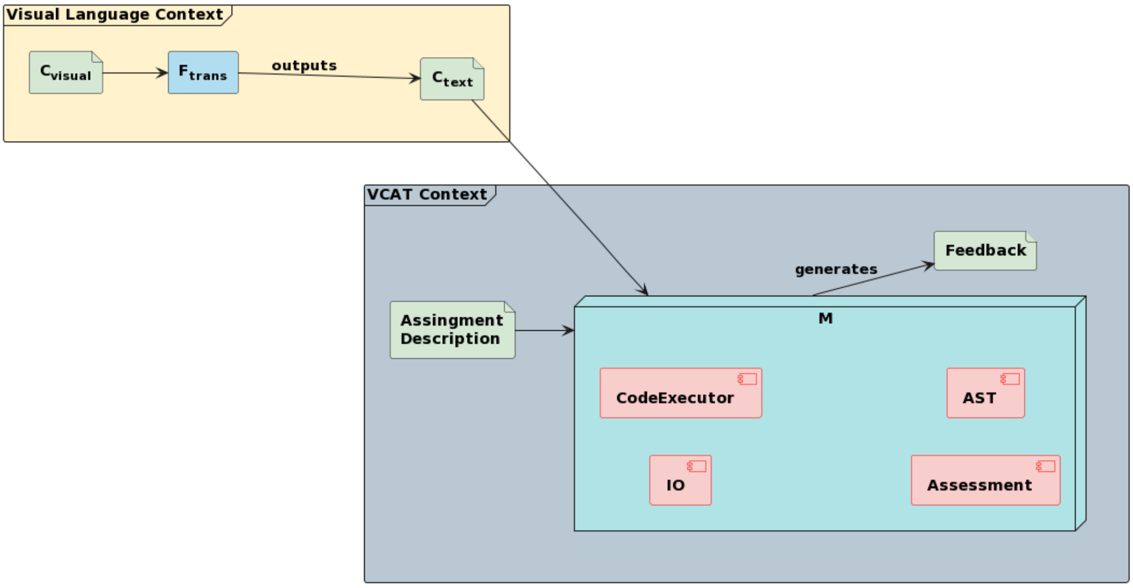


Figure 4.1: *Simplified view of VCAT model*

Let $\mathcal{M}$ be a software component that receives as an input a code and an assignment description both in a predefined format which $\mathcal{M}$ is able to execute, and perform automatic evaluation according to the assignment description provided. And as output, $\mathcal{M}$ provides a feedback on the code it just assessed. Here $\mathcal{M}$ represents the VCAT's core, responsible for not only execution but also code assessment. Now, let F_{trans} be a function defined as:

$$F_{trans} : C_{visual} \longrightarrow C_{text},$$

where C_{text} is a textual representation of C_{visual} which can be executed by $\mathcal{M}$. The representation C_{text} is the translation of each visual object that composes C_{visual} to their corresponding logical meaning given by the language knowledge base B.

Thus, as shown in Figure 4.1 , the model consists in a number of processes that occur in two contexts: the visual language context and VCAT context. The translation of the structure representing a visual code C_{visual} constructed from L_v into an textual representation C_{text} occurs in the visual language context and as such needs to be implemented for each visual language that wants to use VCAT. Then, in VCAT´s context, the representation C_{text} is executed and evaluated by $\mathcal{M}$, providing a feedback that can be presented to the student.

In Figure 4.1 , $\mathcal{M}$ uses an assignment description provided by the teacher which describes how the assessment should be performed. $\mathcal{M}$ then uses the information present in the assignment description and evaluates the code accordingly using both Code Executor and Assessment component. Through the Assessment component, $\mathcal{M}$ can have access to different types of evaluators like output matching, unit testing, static analysis, etc. Here, the IO module in $\mathcal{M}$ allows communication between $\mathcal{M}$ and the visual programming system, specially for feedback presentation and, in case of code execution, requesting input from the user. Next section will provide more details for the AST component along side information on F_{trans} function implementation.

4.2 Implementing F_{trans}

The implementation details of F_{trans} is heavily dependent on the visual language. So, providing a generic implementation that would work for all is not feasible. Given that each visual programming system can represent and manage the visual code in a variety of ways, the need for a function responsible for translating the visual code from the visual programming system context (C_{visual}) to a representation that is known by VCAT (C_{text}) becomes evident. Thus, it is required that any visual programming language that wishes to use all functionalities provided by VCAT must implement their own F_{trans}.

Let L_H be a hypothetical visual programming language defined as $L_H = (ID, G_0, B)$ and let O_1, O_2 and O_3 be visual objects such that $C_{visual} = \{O_1, O_2, O_3\}$ which were constructed by operating over icons from ID given the rules defined in G_0. Thus, it follows that $C_{visual} \in \mathcal{L}(L_H)$ since it is a valid construction from the language L_H. Let's assume that C_{visual} in this case represents the following algorithm represented in natural language:

1. Create a variable named *val* of type integer (O_1).
2. Read a user input into variable *val* (O_2).
3. Write to the output the value from variable *val* times 2 (O_3).

Then, by using the knowledge base B it is possible to map each construction O_i in C_{visual} to their logical meaning and as a consequence generate its equivalent construction in the representation of C_{text}. In VCAT, the AST data structure was chosen to represent C_{text}, and as such the process of creating C_{text} can be defined as a mapping of each visual object logical meaning to a valid AST node.

The AST structure was chosen because it can be executed and it can also be used to encode all commands and expressions of any programming language. And so, C_{text} must be constructed using the AST nodes present in the AST component of $\mathcal{M}$. Therefore, the

AST component must provide all nodes for a general-purpose language in order to cover as many visual programming languages as possible. It must also be feasible to extend the AST component such that new nodes can be created to contemplate logical constructions not covered by the original AST node set.

Taking the visual code $C_{visual} = \{O_1, O_2, O_3\}$ into consideration, Program 4.1 presents a possible implementation of F_{trans} for L_H using JavaScript.

Program 4.1 Example implementation of F_{trans} for a hypothetical visual language L_H.

```
function translateVaribleDef(variableDefBlock) {
   var_ame = variableDefBlock.name
   var_type = variableDefBlock.type
   ast_var_def = new AST.Declaration(var_name, var_type)
   return ast_var_def;
}

function translateReadInput(readInputBlock) {
   var_name = readInputBlock.varName
   ast_var_ref = new AST.Variable(var_name)
   ast_func_read = new AST.FunctionCall("read", ast_var_ref)
   return ast_func_read
}

function translateWriteOutput(write_block) {
   // translateExpression code is omitted for simplicity
   ast_expression = translateExpression(write_block.expression)
   return new AST.FunctionCall("write", ast_expression)
}

function Ftrans(c_visual) {
   root = []
   for (visual_object in c_visual) {
      // from first visual object to last
      // B is the knowledge base of the language
      switch(B.getObjectType(visual_object)) {
         case VARIABLE_DEF:
            root.push(translateVaribleDef(visual_object))
            break;
         case READ:
            root.push(translateReadInput(visual_object))
            break;
         case WRITE:
            root.push(translateWriteOutput(visual_object))
            break;
         // omitted for simplicity
      }
   }
   return root
}
```

Program 4.1 exemplifies how each translation function is tied to the visual object logical meaning the function is supposed to translate. In Program 4.1, F_{trans} implementation, it is possible to notice the knowledge base B guides the function to call the correct translation function given the visual object. Then, each translation function is fully aware of the type of the visual object it is attempting to translate and as such can access the proper fields and select the correct AST node. Moreover, as discussed before a visual object can be a composition of other visual elements as exemplified by *translateWriteOutput* in Program 4.1. To properly translate the write block, the function needs to translate the visual elements present in the expression provided to the write block. This shows that recursive calls will happen during the execution of F_{trans} for any visual language. It is important to note that this is example based on the concepts presented in this text so facilitate the comprehension. In a concrete scenario, it is very likely that there is not special object representing the knowledge base B and the logical meaning of a visual object is actually encoded in itself.

4.3 VCAT's core: $\mathcal{M}$ software component

VCAT's purpose is to provide automatic assessment of code for visual programming languages. To do so, the model needs to provide its users with some ready to use tools which requires minimum implementation on the users' side. The software component needs to be independent of the visual programming language being used and be able to execute and evaluate the code represented by C_{text}. This is very important for it allows $\mathcal{M}$ to be shared with any visual language that wants to instantiate the model.

As discussed in Section 4.2, it was decided to use an AST data structure to represent C_{text} since it can be executed. It can potentially represent any code which meet the expectation of the model. Since the visual programming system itself controls the contruction of the visual code and generates C_{text} itself, there is no point of performing any type of code style assessment. And as such, the use of an AST does not impact negatively the model.

4.3.1 AST component

The AST used in this project is able to represent most general purposes languages. It has nodes for variables, vectors and matrices, block of codes, loops and control flow structures. It has also support advanced structures like pointers, which allows visual languages to work with function parameters as references. Moreover, it can recognize all basic types like boolean, integers, float-point numbers, characters and strings. However, the AST does not support user-defined types like C struct or Pascal record, even though there are internal modules that would support this feature. Since an AST can be linked to a textual code form, it was decided to also allow $\mathcal{M}$ to execute code in traditional text format.

This is influenced by the idea that visual programming should be a helping tool in learning how to code using text (Brandao *et al.*, 2012). The objective is to use visual programming to teach students programming logic concepts like variable, loop and conditional, and show how they are universal for all languages. The textual programming language implemented was based on PortugolStudio's(Dos Anjos *et al.*, 2016; Noschang *et al.*, 2014), sharing some similarities and compatibility where possible. PortugolStudio´s programming language is designed towards beginners and was a good addition to the model.

4.3.2 Code Executor & IO components

The code executor as the name suggests is responsible for executing C_{text}, specially during functional assessment. It allows visual programming languages to dedicate effort on the visual coding aspect and delegate to $\mathcal{M}$ code execution. The code executor is an interpreter designed to execute the algorithm encoded using the AST nodes defined in VCAT. Also, it can fully execute the textual language defined for the AST component. Another important aspect of the code executor is its flexibility. Adding new functionalities at runtime without causing breaking changes to its API should be relatively simple. This can be very useful when providing support for other visual programming languages with some particular requirements.

Another very important aspect of code execution is the ability to interact with the user. By definition, an algorithm is a set of operations performed over an input to provide an output, thus making the IO component a very important part of $\mathcal{M}$. The IO component provides the interfaces required by the code executor to request an input from the user and also to present any output generated during execution or other verification by $\mathcal{M}$. The interfaces provided are expected to fulfill the IO needs of the AST (and as a consequence its corresponding language). As discussed in Section 4.3.1, the VCAT's AST aims to realize the requirement of a general purpose programming language. Thus, IO interfaces are designed to provide the standard capabilities of reading all the basic types defined for the AST (integer, floating-point, character, string and boolean) and also printing them to the desired output. Moreover, in case a visual programming system needs a special kind of IO, the interfaces, the AST and the code executor itself can be extended to support it.

4.3.3 Assessment component

The assessment component as the name suggest is responsible for code evaluation. Its purpose is to house all assessment methods available for $\mathcal{M}$. Figure 4.2 shows the domain diagram for the Assessment component. In VCAT, the AssessmentRunner has the responsibility of running the correct assessment defined in the assignment description file. For this, the AssessmentRunner should provide a static function where assessment methods can be register using the same name as the one expected in the assignment description file. The AssessmentRunner is expected to inform of any issues during the assessment process, and as a result it needs access to the output interface. This same output interface is passed on to the assessment method during its execution in order to provide feedback. Since the AssessmentRunner is designed as a composition of different assessment methods, an abstraction called Assessment was devised.

The Assessment abstraction defines an application programming interface (API) for all assessment methods inside VCAT. As such, it must be implement by any new assessment method since, otherwise the AssessmentRunner will not be able to execute it. The Assessment interface API defined aims to allow the AssessmentRunner to provide the Assessment instance all information inside the assignment description(**setAssignmentDesc**) and execute it (**eval**) and provide feedback. Note in Figure 4.2, that the AssessmentRunner does not have any dependency to the CodeExecutor component, this dependency is pushed to the class implementing the Assessment interface since there is also static types of program assessment which does not require code execution. In Figure 4.2, the dashed line from the NewAssessment class to the CodeExecutor component is to indicate that this dependency is not mandatory.

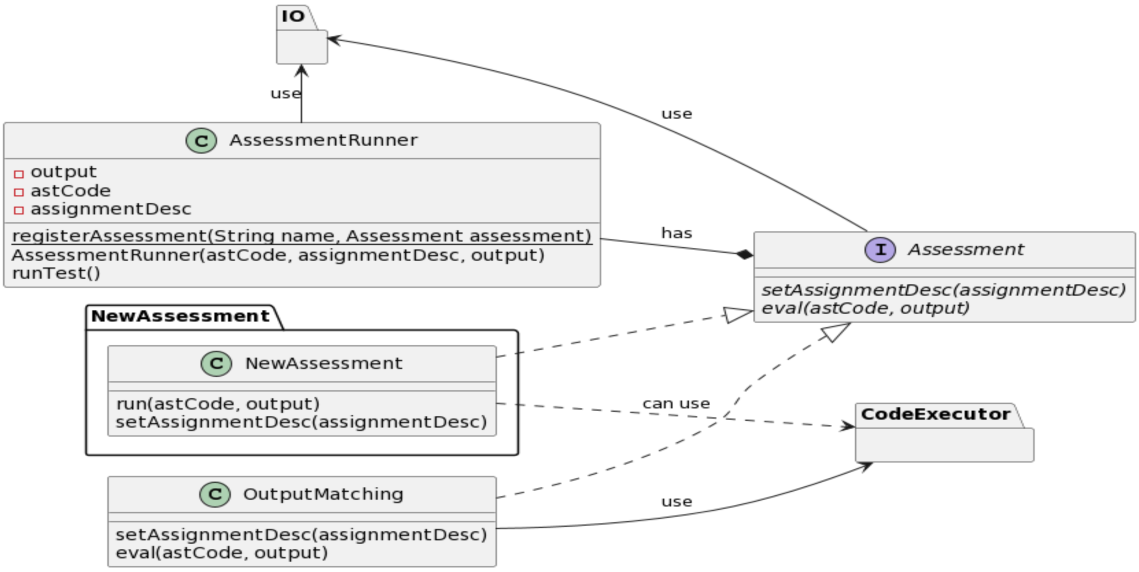


Figure 4.2: *Domain diagram for VCAT Assessment component*

In this initial version, VCAT has integrated to $\mathcal{M}$ an output matching assessment. This output matching class is an improved implementation over the standard approach as presented in Section 3.1.2 and is based on iVProg's original version. The new algorithm can now properly deal with the problem of numeric precision and treat typed values independent of text. As discussed in Section 3.1.2, it is common to output matching algorithms to label solution as wrong if the student's output have a higher precision than the teacher's solution. Inspired by the literature and the author's own interactions with students during classes, VCAT's output matching algorithm focus not only on improving how the algorithm treats the generated output but also on the feedback provided to the students The improvements on the presentation of the inconsistencies to the students are designed so they can understand their mistakes and easily take action at the likely source of the issue.

One very common problem for traditional output matching algorithms is to mislabel as incorrect solutions where the generated numeric value has different precision than the one expected even though a human would label it as correct. In VCAT's output matching algorithm, students' numeric outputs are truncated if they have higher precision than the expected output. When dealing with textual output, the new algorithm computes the differences between the produced output and the expected one, so students can easily identify typos or missing words. The solutions presented by Yu *et al.* (2017), C. K. Poon *et al.* (2016), and Chung Keung Poon *et al.* (2018) for these textual issues are very complex to implement and require a lot of work from the instructor side.

Additionally, in VCAT the output matching algorithm will break down the students solution in parts, one for each type(numeric, text and boolean) identified in the expected output. The assessment will be performed in each part individually, avoid situations where the students will score 0, even though they had got really close to the correct solution. Moreover, the text output is assessed character by character and the grade is penalized by the number of missing/wrong characters. With this in place, the student will get a percentage of the grade based on how many parts are correct instead of a straight zero. For instance if the expected output is "area: 25" but the student code outputs only "25", VCAT will score it as 50% of the grade since the numeric output matches the expected.

In VCAT, a simpler solution to the problem was designed. To better visually indicate the textual issues in the output, a textual difference tool (*diff*) was implemented as part of VCAT's output matching code. The *diff* tool is heavily used in programming and textual edition environments. It uses colors and strike-trough text formatting to indicate text that must be inserted, kept or removed. So, VCAT feedback highlights the differences between the output produced by students solution and output expected by the assignment using the *diff* tool. It makes it easier for students to spot the issues in the output more quickly and also immediately know if they had extra text added or missed something. The output processing checks the generated output line by line to facilitate this process. Even though the current version of VCAT only has output matching assessment, the system is robust enough to accept new modules with new assessment strategies. This is a result of an initial focus on providing a better and clear assessment feedback to the students in order to be more helpful when errors occurred.

Talking about new assessment methods, a new method that could be implemented is a static analysis assessment. As presented in Figure 4.2 , the main requirement is to implement the Assessment interface. For instance, a static analysis module can use the AST representing the code to verify the design and software metrics like code complexity, presence of loops and recursion. Once another assessment option is available it would then be possible to perform compounded assessment of multiple types. The static analysis could, for example, be used to prevent a program from being evaluated by the output matching algorithm if it does not meet an specific requirement or vice-versa. This would be an improvement on the method presented by J.-H. Kim *et al.* (2019), since in Chentry only the log was analysed and not the actual code produced by the student.

As discussed in Section 4.1, the Assessment component has to be able to deal with the visual languages types of I/O. VCAT's initial version has support for text-based I/O which is aligned with the most traditional languages and the general nature of the exercises for the learning level the initial implementation is aimed for. This is was a necessity since providing a global solution to all possible scenarios would be unfeasible. VCAT was created as a flexible and extensible tool which allows visual languages to adapt it to their needs if required. Also, any new feature needed can be added as a new component under VCAT's $\mathcal{M}$ and as a consequence all visual languages using VCAT would also be able to use the new feature.

VCAT's output matching assessment requires a set of test-cases to be executed. The test-case is provided to VCAT by an assignment description file which can also be extended to set some configurations in VCAT's modules. Next section will discuss the format and the parts of the assignment description file.

4.3.4 Assignment Description

The assignment description is an important element of VCAT's model. It is responsible for instructing $\mathcal{M}$ how it should perform the automatic assessment of the provided code. The assignment description file format is influenced by the automated assessment algorithm being used. However, VCAT expects the file to be a valid JSON object, which is a portable, descriptive file format and easy to read, be it by a human or machine. For instance, Program 4.2 describes how the assignment description file could look like for using with output matching algorithm.

Program 4.2 Example of an assignment description file for output matching in VCAT.

```json
{
  "assessment": "output_matching",
  "test_cases": [
    {
      "input": [2,2],
      "output": [4]
    },
    {
      "input": [3,3],
      "output": [27]
    }
  ]
}
```

It tells VCAT to perform automatic assessment using the output matching algorithm. Given that in this algorithm it is mandatory that test cases, Program 4.2 also has a set of pairs of inputs and the expected outputs which are used to assess students' code. The values inside the fields input or output can be anything as long as the $\mathcal{M}$ can handle it. So, in case of the addition of a new type of input, the only requirement is that the new input abides to the rules set by the IO module. The same principle applies for the output.

Program 4.3 Example of an assigment description file for static assessment in VCAT.

```json
{
  "assessment": "static_assessment",
  "check_rules": {
    "variable_naming": "snake_case",
    "code_complexity": 45,
    "max_nesting_level": 2,
  }
}
```

To illustrate the relationship between the assignment description file and the automatic assessment method used, Program 4.3 presents a potential file format for a static assessment algorithm. As mentioned in Section 3.1.1, generally speaking static assessment checks the source code style, programming errors and software metrics. Program 4.3 shows an example of a static assessment algorithm that checks variable naming style and code metrics like its complexity and the level of nesting of flow control and looping structures. Static assessment can be a great tool in developing good coding practices by pointing out issues that are not easy to spot and even avoid crashes during runtime. Even though VCAT is quite flexible and open for extensions, there are somethings that it cannot handle or requires some specialized class. Next section will present the known limitations of the model and how they affect the model.

4.4 Model limitations

Unfortunately, the proposed model cannot be applied in all contexts and for all code that can be produced by visual languages in general. Some cases can be solved by implementing a new functionality that allows VCAT to provide the missing feature, like inputs and outputs that are not the traditional keyboard and text. Other limitations, however, cannot be easily solved by implementing a new functionality. For instance, parallel-like execution of some solutions present in visual programming languages like Scratch and Chentry.

These parallel-like executions have a set of independent blocks of code to have individual lifetime while allowing them to communicate between each other. This make dynamic assessment of code extremely difficult. Nonetheless, this difficulty is not limited to VCAT itself but computer science as a whole, since parallel code usually has non-deterministic behavior (Y.-J. Kim *et al.*, 2005), meaning that each execution with the same input can produce the output in a random order each time. Output matching algorithm and as other dynamic assessment algorithms like unit testing, cannot handle this kind of behavior properly. So, currently, any solution that presents a similar behavior to a parallel execution cannot be dynamically assessed with VCAT. However, static analysis would still be feasible since there is no execution of the code and it is a deterministic process. Next chapter will report the steps for the instantiation of VCAT for iVProg and another visual language designed by the author.

Chapter 5

VCAT instantiation

This chapter describes the required steps to instantiate VCAT for a given visual language. It presents how F_{trans} was instantiate for the languages presented and the usage of VCAT IO module to allow interaction between the user and $\mathcal{M}$. First, the process for instantiating VCAT for iVProgis presented since it is the main visual programming language for VCAT and one of the flagships of our research group. The other language presented in this chapter is a simple general-purpose visual programming language developed by the author. The idea behind this second language is to demonstrate that VCAT is a general model applicable to other visual languages and not only iVProg.

5.1 Implementing software component $\mathcal{M}$

As discussed in Section 4.3, the software component $\mathcal{M}$ is the core of VCAT. Among its functionalities is the ability to execute the output provided by F_{trans} and perform automatic assessment. Since in last chapter presented VCAT from an abstract point of view, here $\mathcal{M}$ will be presented in its concrete form for the model instantiation. It is important to keep in mind that $\mathcal{M}$ is part of the VCAT context and thus does not change as a consequence of the language it is being instantiated for.

Aligned with the research group approach of developing decentralized web applications, VCAT's $\mathcal{M}$ was developed using HTML5 stack (HTML, CSS and JavaScript) to allow it to be executed independent of a web server. As a consequence, all components were also developed using JavaScript and its super-set language TypeScript. This not only allows students' web browsers to execute everything but also provides the flexibility required by $\mathcal{M}$. The JavaScript language makes it possible for objects to be extended with new functionalities during runtime without actually needing to modify the original source code. This makes $\mathcal{M}$ very flexible and able to be extended to fulfil any special needs of a new visual language, although in VCAT's case, some small intervention would be needed in the original source code to any modification to take effect. For security reasons, it was decided to not allow free modification of $\mathcal{M}$ during runtime since this could affect students' data and the trustworthiness of the automatic assessment result.

Figure 5.1 presents the domain diagram of the main packages, interfaces and classes

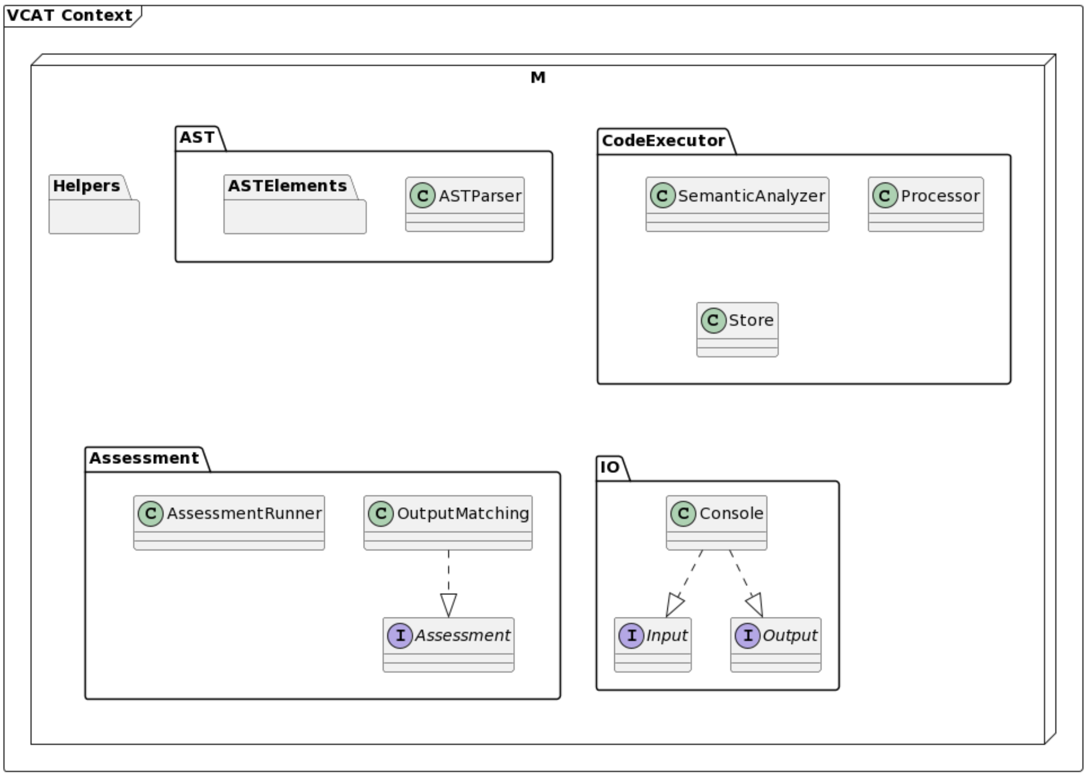


Figure 5.1: *Domain diagram for the JavaScript implementation of* $\mathcal{M}$

that composes $\mathcal{M}$ JavaScript implementation. It expands the contents resented in Figure 4.1 and discussed in Section 4.3 on Chapter 4. Do note that the actual implementation contains more packages and classes, but they are relevant to the intended discussion. In Figure 5.1 there is a new package called Helpers which houses some of the code used to improve the feedback provided by the OutputMatching class. For example, the Levenshtein distance algorithm is implemented inside the Helpers package so it is available for others packages to reuse easily. There is also the code for formatting the output of the Levenshtein algorithm like the commonly used textual difference tool *diff*. In general, the package contains reusable implementations of common or useful tasks that can be used anywhere inside VCAT's $\mathcal{M}$.

5.1.1 AST package

Figure 5.2 shows the implementation of the AST component described in Section 4.1 with more details of the ASTParser class and one interface available in the ASTElements package. ASTElements is a package created to hold all the elements representing the AST nodes recognized by VCAT. The elements are divided into two categories: expressions and commands. Command nodes represents all instructions from the language that can alter the program state like: function and variable declarations, assignments, control flow, loops, I/O and other function calls. On the other hand, expression nodes represents all parts of any expression in the language like "$i + 2$", "$5 + (3 * i)$" or "$pow(i, 2) + sqrt(i + 1)$"

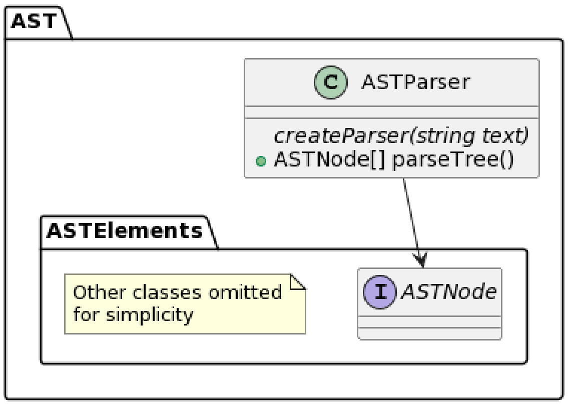

AST
ASTParser
createParser(string text)
ASTNode[] parseTree()
ASTElements
Other classes omitted for simplicity
ASTNode

Figure 5.2: *Class diagram for the AST package JavaScript implementation*

which produces values but does not alter the program state. The expression nodes contains: functions calls that return values, unary and binary operations, constant and variable values. Among all the expression nodes, the only one that can break the rule regarding not altering the program state is the function call expression. This is because a function in VCAT can receive a parameter by reference and thus all modifications to that parameter will be reflected outside of the function. So, given that the functions *pow* and *sqr* above are not by reference, the consecutive evaluation of them would produce the same result always as long as not command altering the state of variable *i* is executed in between. One can see expression node as idempotent operations while command nodes are not.

All commands and expression nodes have the ASTNode interface as a common superclass, this helps abstract away the role of each node where the information is not necessary. Then, through object-oriented programming inheritance mechanism, the ASTParser can decode textual code into a list of ASTNodes that will be later executed by the Processor class. The text-based language, as mentioned in Section 4.3.1, is inspired by PortugolStudio and should be seen as a stepping tool to move from a visual programming language to a text-based language. As shown in Figure 5.2, the ASTParser has a static method called *createParser(string text)* used to create a ASTParser instance. There are many other methods inside a ASTParser instace, but from the final user point of the the most relevant method available is *parseTree()*. This method will parse the text fed to the *createParser* function and generate a list a ASTNodes in a tree-like approach, where each element in the list is a child of the root node. Another important aspect of ASTParser is that it supports internationalization and can currently parse code written in English or Portuguese. Appendix A presents the syntax rules for English of the text-based language supported by VCAT in the Extended Backus–Naur form (EBNF). Additionally, the list of all built-in functions and their description can be found at Appendix B.

5.1.2 Code executor & IO packages

The class diagram in Figure 5.3 describes the JavaScript implementation of the packages CodeExecutor and IO. The diagram in Figure 5.3 shows the classes contained inside the CodeExecutor package: SemanticAnalyzer, Processor and Store. The SemanticAnalyzer is capable of performing semantic analysis of the code which checks if all nodes have been correctly constructed while also performing type checking. Among the checks performed, the SemanticAnalyzer can identify if a function that must return a value but are missing a return command; if the number and types of the parameters passed to a function meets what is expected by the function; if an expression evaluates to the correct type of the variable it is being assigned to, be it a atomic variable or an array. If any issue is found, the SemanticAnalyzer will throw an exception and attempt to inform to the best of its capability what is wrong and where.

Responsible for code execution, the Processor class is a very important element of VCAT since dynamic assessment, like output matching, requires code execution to be performed. The class has dependency with the Input, Output and ASTNode(Figure 5.2) interfaces. The Input and Output interfaces enables the Processor to perform I/O operations independent of the environment it is being executed. This enables each visual programming system to implement the interfaces accounting for their particular needs.

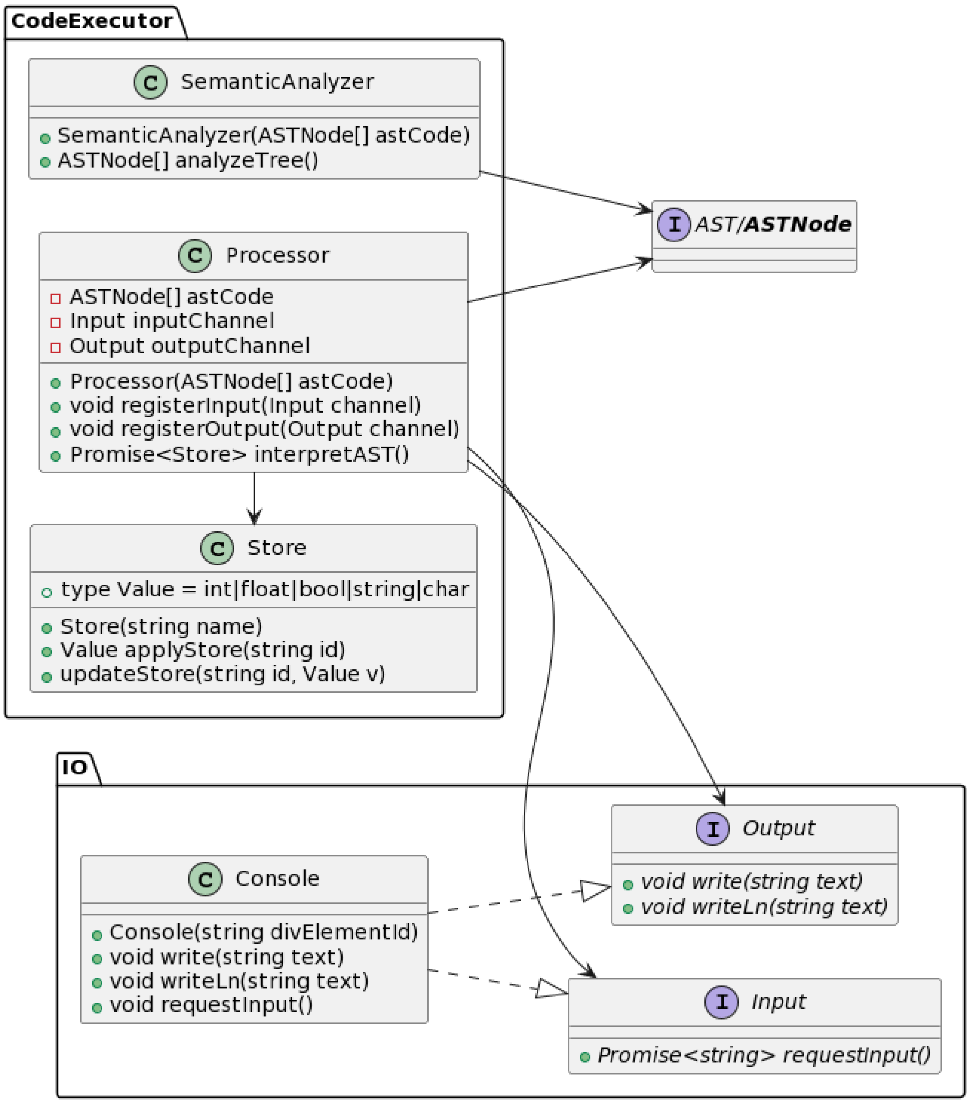

CodeExecutor
C SemanticAnalyzer
SemanticAnalyzer(ASTNode[] astCode)
ASTNode[] analyzeTree()
I AST/ASTNode
C Processor
ASTNode[] astCode
Input inputChannel
Output outputChannel
Processor(ASTNode[] astCode)
void registerInput(Input channel)
void registerOutput(Output channel)
Promise<Store> interpretAST()
C Store
type Value = int|float|bool|string|char
Store(string name)
Value applyStore(string id)
updateStore(string id, Value v)
IO
I Output
void write(string text)
void writeLn(string text)
C Console
Console(string divElementId)
void write(string text)
void writeLn(string text)
void requestInput()
I Input
Promise<string> requestInput()

Figure 5.3: *Class diagram for the CodeExecutor and IO packages JavaScript implementation*

Also, the Processor is dependent on the Store class which represents the program state. In VCAT, the program state is a stack of all active scopes of the program where an active scope is a function being executed. The Store in the bottom of the stack represents the global scope where all global variables are defined. The function *Processor.interpretAST* is responsible for the execution of the program and it returns the final state after complete execution. It is a non-blocking asynchronous function which is represented in JavaScript by the type **Promise**. Since executing any code involves operations that are dependent on user interactions(I/O) or resource being available in the network or disk, the Processor had to be designed with that in mind and JavaScript is a very good language to deal with asynchronicity.

All IO abstractions can be found on the IO package like the interfaces Input and Output. Notice that the Input interface *requestInput* function also has **Promise** in its return type signature. AS mentioned before, this is inherent of the I/O operations in a regular computer, and making this synchronous will mean that user browser would just freeze until an input was feed to the $\mathcal{M}$. Also, the IO package ships a class called Console which implements both Input and Output interfaces. The Console class is a ready-to-use HTML implementation with a set of functions to allow easy integration to any web page. It uses HTML, CSS and JavaScript to create the look-and-feel of a regular console where data input can be request and outputs can be shown.

5.1.3 Assessment package

The Assessment package JavaScript implementation does not differ from the model design presented in Section 4.3.3 besides the details regarding functions return types and the Result class as shown in Figure 5.4. The AssessmentRunner class is responsible for reading the assignment description file and performing the instructed assessment. It uses an internal mapping of assessment name and their corresponding implementation. An assessment is added to this mapping by calling the static function *registerAssessment*. The function *runTest* returns a int value representing the final grade for the whole assessment and is an asynchronous function.

In the JavaScript implementation, VCAT's OutputMatching class has two internal private fields with special implementations of the I/O interfaces. These implementation are used only during dynamic testing and they provide the input in the test case to the program. Also, the output interface implementation for dynamic testing captures all outputs generated by the program during execution. The captured output is then used top verify if it matches with the expected output from the test case. The OutputMatching class will use all the improvements mentioned on Section 4.3.3 to generated a better feedback to the student, using its access to the system output interface to present it. Besides performing the assessment of the code represent by the ASTNode, the function *eval* will also return a Result instance wrapped inside a **Promise**.

The Result class encapsulates the grade of the performed assessment and a JSON object called *evalResult* that stores data on the feedback provided. Taking the OutputMatching class as an example, the object *evalResult* contains the input used, and generated output and the expected output along side the *diff* formatted texts to present to the student. Additionally, the *evalResult* can also be used to provide metadata from the assessment to

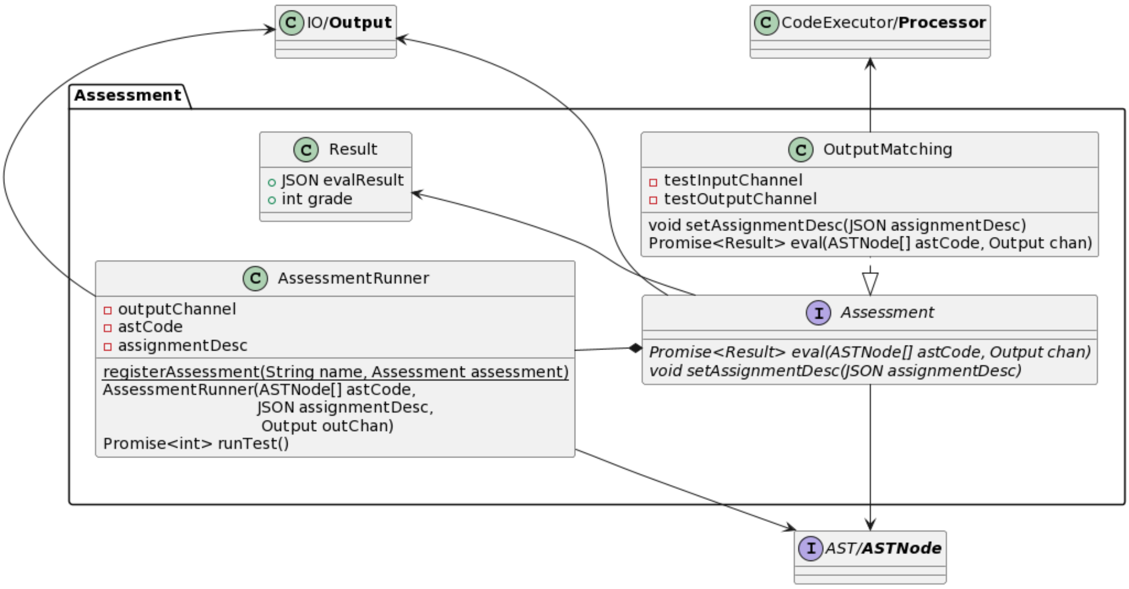

C IO/Output
C CodeExecutor/Processor
Assessment
C Result
+ JSON evalResult
+ int grade
C OutputMatching
- testInputChannel
- testOutputChannel
void setAssignmentDesc(JSON assignmentDesc)
Promise<Result> eval(ASTNode[] astCode, Output chan)
C AssessmentRunner
- outputChannel
- astCode
- assignmentDesc
registerAssessment(String name, Assessment assessment)
AssessmentRunner(ASTNode[] astCode, JSON assignmentDesc, Output outChan)
Promise<int> runTest()
I Assessment
Promise<Result> eval(ASTNode[] astCode, Output chan)
void setAssignmentDesc(JSON assignmentDesc)
I AST/ASTNode

Figure 5.4: *Class diagram for the Assessment package JavaScript implementation*

the AssessmentRunner like runtime. Having presented the implementation details on $\mathcal{M}$, the next step is to use it to instantiate the model for a visual programming language. The first instantiation will be for iVProg, followed by an instantiation for a visual programming language created by the author using Google's visual language framework Blockly. The next section will discuss iVProg's implementation of F_{trans} and the steps necessary to connect its output to VCAT's $\mathcal{M}$.

5.2 Instantiating for iVProg

iVProg is one of the main flagships of the Laboratory of Informatics in Education - LInE and as such is the main instantiation for VCAT. It is part of a framework developed for Moodle called iAssign which provides learning modules for different subject like Geometry, Programming and Mathematics. Given its compatibility with Moodle, iVProg is a visual programming system developed with the Web in mind. During this research, iVProg and VCAT development went hand-to-hand even though they are two different software. There were many features introduced to iVProg that were inspired by VCAT model and vice-versa. For example, the possibility of using a textual code as an alternative to constructing the AST tree directly was inspired by iVProg. On the other hand, iVProg's adoption of function parameters passing as a reference was influenced by VCAT implementation of the functionality.

For iVProg visual language each block defined is an element of the icon dictionary ID. It is important to note that iVProg handling of variables is different from other visual programming languages like Alice, Scratch and Chentry. In iVProgvariables are not blocks themselves. Instead, they are incorporated in the blocks, i.e. the *read* block has a list of variables that you can read values into. Additionally, all variables in iVProg have a defined type much like traditional programming languages. Its user interface controls how each type of block can interact with one another fulling the role of the language grammar G_0.

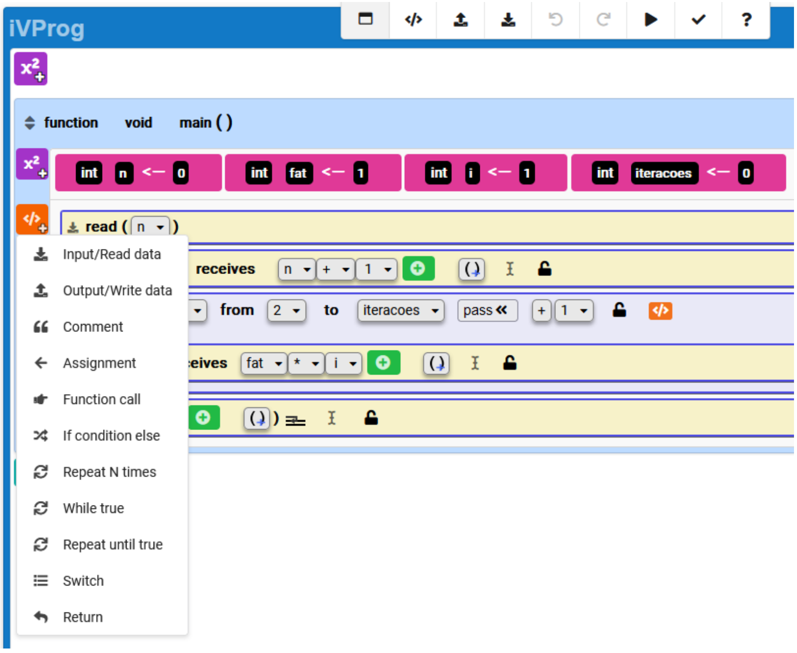


Figure 5.5: *List of command blocks available in iVProg*

Each block has encoded in itself its type and role, and collectively this data produces the knowledge base B, where it is possible to map each block to a logical meaning. Figure 5.5 presents the list of available commands in iVProg. A noticeable detail in the figure is that each block name already hint to their role and logical meaning.

As already presented in Section 2.2.2, iVProg working area is well defined and consists, by default, of a main function body. The user interface allows students to create new functions, which can have parameters and their own variables. iVProg also supports global variables and all basic types supported by VCAT, including vectors and matrices. In iVProg, C_{visual} is a tree-like structure very similar to an AST, that is stored by the visual programming system. The advantage of being a tree-like is that the implementation of F_{trans} becomes straight forward since it can be implemented as a recursive descent parser as discussed in next section.

5.2.1 Implementation of F_{trans}

The F_{trans} function for iVProg was implemented as part of its user interface module. The function F_{trans} walks through C_{visual} using a recursive descent parser. It parses C_{visual} in a top-down approach, checking all leaves from the current node before going to its next sibling. Even though iVProg handles the variable definition in a different way, this does not interfere in the translation process from C_{visual} to C_{text}.

Thus, each node is transformed into its textual representation using the textual language format since iVProg has a textual mode. As mentioned before, the idea is to use visual programming language as a bridge to textual programming since this will be the main form of programming students will face in their professional lives. The visual aspect is to facilitate students assimilation of the programming concepts and be able to apply them using a text-based language.

Program 5.1 Code snippet for iVProg C_{text} generation for an assignment block.

```
1   function assignmentCode (command_obj, indentation) {
2     let ret = "\n";
3     ret += " ".repeat(indentation);
4     ret += variableValueMenuCode(command_obj.variable) // Generate code for
          leaf node of Variable type
5     ret += " <- ";
6     ret += elementExpressionCode(command_obj.expression); // Generate code for
          leaf node Expression type
7     return ret;
8     }
```

Program 5.1 presents a part of iVProg F_{trans} implementation for the variable assignment block. Besides the basic textual formatting presented in lines 2 and 3, the code first checks the leaf node for the variable block contained by the assignment block(line 4). Then, it will check the inner elements of the assignment command block, starting with the variable. Although a variable in iVProg is not a block the user can manipulate, it behaves and is encoded like one.

After calling the function responsible for generating the textual code for the variable in the command, it then calls the function that will generate the text for the expression in the assignment command. The *elementExpressionCode* function will recursively call itself for each expression element in *command_objėxpression* until it reaches the atomic elements of an expression like variable name or constant value. The final product of all these function calls is the textual code(C_{text}) for the assignment block being translated.

Program 5.2 Example code for AST nodes generation for iVProg's Assignment command.

```
1   function assignmentASTNode (command_obj) {
2     const variableNode = variableValueASTNode(command_obj.variable);
3     const expressionNode = elementExpressionASTNode(command_obj.expression);
4     const assignmentNode = new vcat.AST.AssingntmentNode(variableNode,
          expressionNode);
5     return assignmentNode;
6     }
```

Program 5.2 shows an example implementation of F_{trans} using AST nodes instead of generating the textual language. The main difference is that the functions now return ASTNode objects instead of text. The steps are the same: 1) first the node for variable is generated; 2) then the node representing the expression being assigned to the variable is created; 3) and last the assignment node is created having the variable and expression nodes as children. Although the expression node on line 3 is a single object, it can contain many other expression-type nodes as children, in order to represent complex expressions. As with Program 5.1, the final result will be a C_{text} in AST structure.

Once C_{text} is generated, the next step is to input it into $\mathcal{M}$. This step is very straight forward, specially if the only desired functionality is the AAT. SinceiVProg also uses VCAT as its code executor next section will describe the steps used in iVProg to feed C_{text} into

$\mathcal{M}$ for execution and automatic assessment.

5.2.2 Inputting C_{text} into $\mathcal{M}$

After C_{text} is created from the recursive descent parsing, it is then inputted to VCAT M for execution or assessment. Program 5.3 shows a snippet code of how to input C_{text} into $\mathcal{M}$ be executed. iVProg uses VCAT not only as a source of AAT but also as its main code executor. For this reason, iVProg does not need to implement code execution since it can use $\mathcal{M}$ instead.

As presented in Section 4.3 and by the class diagram in Figure 5.3, code execution requires the setup of the I/O channels that must be used by the Processor. iVProg makes use of the Console class in the IO package(Figure 5.3 as Input and Output interface. Since iVProg is a complete web application, it has a dedicated area where the Console class can inject its HTML nodes to mimic the console look-and-feel. As a result, this simplifies the integration process between iVProg visual programming system and VCAT.

Program 5.3 Code snippet for executing a program using F_{trans} and VCAT.

```
1   const ast = vcat.CodeExecutor.SemanticAnalyser.analyseFromSource(c_text);
2   const proc = new vcat.CodeExecutor.Processor(ast);
3   // vcatConsole is an instance of Console created earlier, it works as both input and output
4   proc.registerInput(vcatConsole)
5   proc.registerOutput(vcatConsole);
6
7   //Focus the console
8   vcatConsole.focus();
9
10  proc.interpretAST().then((_finalProgramState) => {
11    console.log("Program executed sucessfully");
12  }).catch(console.error);
```

In Program 5.3, the first line uses C_{text} to generate the AST tree but also checks for semantic issues in the code. In the scenario where C_{text} is already in the AST format, it is possible to go straight to line 2. However, if the semantic check is still desired the SemanticAnalyser can be instantiated using the constructor presented in Figure 5.3. Then, a new instance of the Processor class is created, using the generate AST. The final step before actual execution is to register the input and output channels that will be used during execution. In this case, an instance of VCAT's Console is used. The execution of the code is triggered by calling *Processor.interpretAST* function which in turn, returns a *Promise<Store>*. The variable *_finalProgramState* is an instance of Store and represents the final state of the program.

Program 5.4 Code snippet for running automatic assessment in iVProg using VCAT.

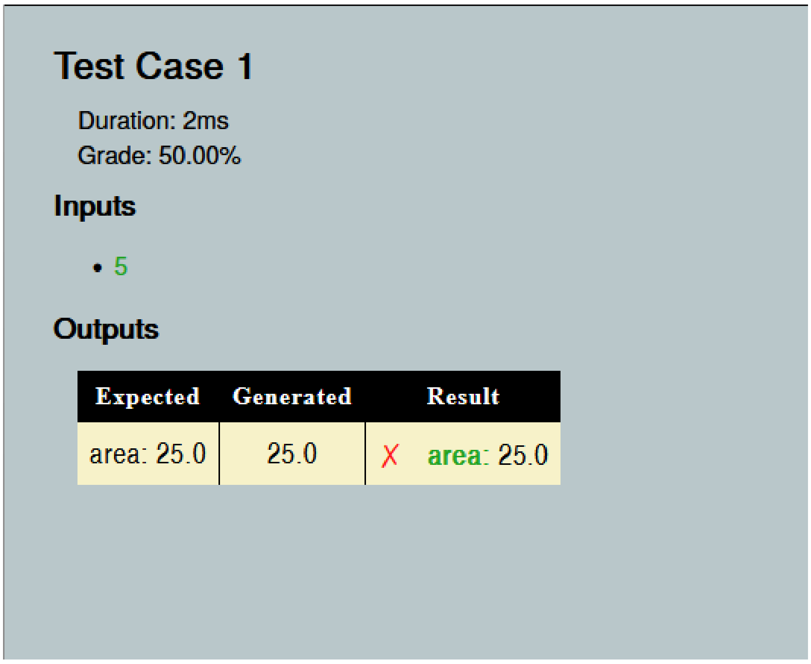
```
// assignmentDesc is a JSON object in VCAT's format
const ast = vcat.CodeExecutor.SemanticAnalyser.analyseFromSource(c_text);
```

cont ⟶

⟶ *cont*

```
3   const assessment = new vcat.Assessment.AssessmentRunner(ast,
4       assignmentDesc, vcatConsole);
5   assessment.runTest(),then((final_grade) => {
6       // The feedback has already been printed by the AssessmentRunner using vcatConsole
7       // final_grade stores the final grade for the exercise calculated as a mean of all grades of each test case
8       // The value is in the interval [0, 1], where 1 represents 100\%.
9       // This can be used to register the information on database or other storage for later use.
10      // The final grade will be presented by default using the output interface provided
11      console.log(‘Final grade: ${final_grade}‘);
12  }).catch(console.error);
```

The steps required to perform automatic assessment does not differ much from the steps to execution. In Program 5.4, it is possible to see that C_{text} is also checked for semantic issues before being passed to the assessment runner. Now, instead of feeding the generated AST to the Processor class, it used as parameter in the constructor of the AssessmentRunner. The AssessmentRunner will be the one responsible of providing the AST structure to the proper assessment method as defined in the assignment description file. The automatic assessment is trigger by line 5 and when it is done the AssessmentRunner will use the vcatConsole instance to print the feedback to the student. The Promise returned by the runTest function, contains the final grade of the assessment which can be however the instructor see fit like storing it into a database for later retrieval.



Test Case 1

Duration: 2ms
Grade: 50.00%

Inputs

- 5

Outputs

Expected	Generated	Result
area: 25.0	25.0	✗ area: 25.0

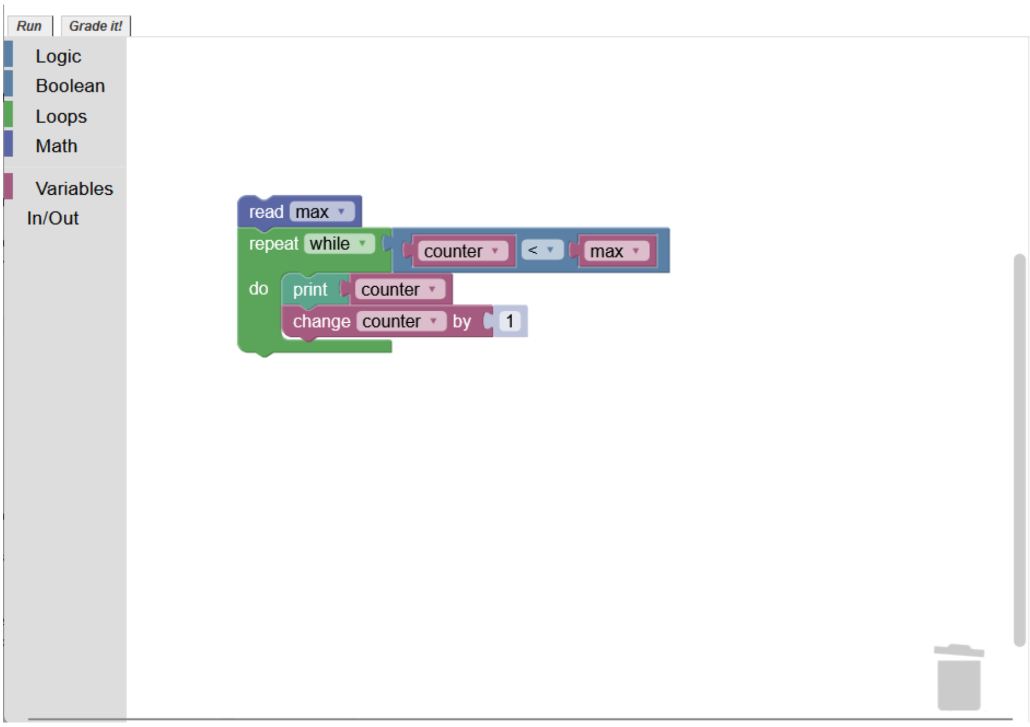
Figure 5.6: *Output Matching feedback page showing the type-based assessment of the output*

Figure 5.6 shows the feedback page for an assignment where the student’s program did not produce the correct output. It is possible to see that even though the output was not correct, the solution was graded with 50% of the grade since the numeric part of the output was correct. Also, the result column shows that the output is missing the text "area: " which is highlighted in green.

This described all the steps performed to integrate VCAT to iVProg in order to provide code execution and code assessment. Although iVProg uses the textual language approach to implement F_{trans}, using the AST node directly is still an alternative. Next section will discuss the steps required by the special visual programming language developed by the author using Blockly to demonstrate VCAT generality.

5.3 Instantiating for Blockly

To demonstrate model generality, the author created a visual programming language using Google´s Blockly. Blockly is a framework for developing visual programming system that allows the user to create different kind of blocks using XML or JSON format. It already comes with a user interface that has an area for the blocks defined for the language and an area for code construction. Blockly user interface is heavily inspire by Scratch´s but can be slightly changed to provide a different look and feel.



```
read max
repeat while counter < max
do  print counter
    change counter by 1
```

Figure 5.7: *User interface for Blockly instantiation*

Figure 5.7 shows Blockly user interface for the language of this instantiation. The figure shows the HTML page created by the author with Blockly inserted to it. Apart from the two top-left buttons: *run* and *grade it!*, everything else is from Blockly. On the left is the tool box with all the blocks available for the language and on the right of it is the working area. In the working area the user places and combines blocks to create the algorithm as presented in the figure.

Let´s recall the formal definition of a visual language: $L_v = (ID, G_0, B)$(Section 2.2.1). Here we have that each block defined is an element of ID. Also, when creating a new block using Blockly, the user needs to define how this new blocks interacts with the existing blocks which fulfils G_0 requirement. The creation process also populates the knowledge base B, since the framework requires the user to add each block to a category that maps to a logical meaning, i.e flow control, loop, variable, expression, etc.

For simplicity, the language implemented for this instantiation had support to only a numeric data type which can represent integer and float-pointing numbers. To fulfill the requirements of a general purpose language, blocks for flow control, loop, variable definition and I/O functions were also added. The language also had support for arithmetic and logical expressions where the logical expression could use equality and logical operators (and, or and negation).

5.3.1 Implementing F_{trans}

The Blockly framework comes with a Generator class that fits perfectly the purpose of F_trans. This class is designed to allow the translation of the blocks created to any desired structure or textual language. Thus, the Blockly Generator was used to translate the blocks to the textual language used by VCAT. For that, the Generator class behaves as a dictionary where the user associate one kind of block to a function responsible for generating textual code given the logical meaning of that block.

As mentioned in last section, the visual language created has a numeric type to represent all integers and floating-points numbers. This is relevant because VCAT AST requires a type to be provided to all variables. This is because in order for the SemanticAnalyser to properly check the AST structure, the type information is crucial to assert that the code is semantically correct. In the event that the visual language needs to provide a polymorphic type that can represent all types in one(numbers, strings, characters, etc.), it would be required to implement a new library responsible for reading and writing this new type. VCAT internal type system is advanced enough that it already provides internal support for polymorphic types making it easier for such changes. Although the support is already present, its usage is limited to special inner functions cases.

Program 5.5 Code snippet of F_{trans} implementation for the visual language created using Blockly.

```
CODE_GENERATOR['variables_set'] = function(block) {
  // Variable assignment
  const varName = CODE_GENERATOR.nameDB_.getName(block.getFieldValue("VAR"),
      Blockly.Names.NameType.VARIABLE);
  const argument0 = CODE_GENERATOR.valueToCode(
                    block, 'VALUE', CODE_GENERATOR.NONE) || '0';
  return varName + ' <- ' + argument0 + '\n';
};
```

Thus, the step of translating C_{visual} into C_{text} in this case is a matter of walking through the blocks present in the working area and producing their respective textual code. This is done every time the user clicks on the *run* or *grade it!*. Program 5.5 shows a snippet of the code generation for the variable assignment block. The implementation is very similar to iVProg's in Program 5.1. The *CODE_GENERATOR* variable is an instance of Blocky **Generator** class and it behaves as a dictionary where the keys are elements of *ID* and are mapped to the functions responsible for generating the final code. The Blockly implementation of F_{trans} also use the textual language as the format for C_{text} since it is must faster to implement and the creationg of the AST nodes can be delegated to VCAT.

One of the main differences between this implementation and iVProg's is that in the language created with Blockly, the variables are blocks themselves. And as such, to access the variables names the framework provides a name database that can be used to access the name of all variables. This is required because inside the blocks the framework does not store the actual name of the variable, but an id hash instead. Additionally, instead of calling a particular function for expression code generation, line 4 calls a function defined by the framework itself to generate code for expressions which are referred to as **values**. Behind the scenes, the function *valueToCode* will call the appropriate functions in order to generate the code.

Now that F_{trans} is implemented and can generate C_{text}, the next step is to input C_{text} into $\mathcal{M}$. As iVProg, this visual language will use VCAT's $\mathcal{M}$ for both execution and automatic assessment. Next section will present the implementation steps employed to integrate $\mathcal{M}$ to the Blockly framework.

5.3.2 Inputting C_{text} into $\mathcal{M}$

Much like iVProg, the implementation for code execution follows the same steps. After generating C_{text}, the SemanticAnalyzer is used to create the AST code structure. Then, a instance of the Processor class is created using the AST code and the desired I/O interfaces. Program **??** shows part of the implementation for code execution. First step is to generate C_{text} by calling **translate()**, a function created to call *CODE_GENERATOR.blockToCode* function. From line 2 on wards the lines are the same in the iVProg implementation.

Program 5.6 Code snippet for executing a program using F_{trans} and VCAT.

```
1    const c_text = translate();
2    const ast = vcat.CodeExecutor.SemanticAnalyser.analyseFromSource(c_text);
3    const proc = new vcat.CodeExecutor.Processor(ast);
4    // vcatConsole is an instance of Console created earlier, it works as both
         input and output
5    proc.registerInput(vcatConsole)
6    proc.registerOutput(vcatConsole);
7    vcatConsole.focus();
8
9    proc.interpretAST().then((_finalProgramState) => {
10     console.log("Program executed sucessfully");
11   }).catch(console.error);
```

It is also possible to note in Program 5.7 that to use the automatic assessment, the steps are also the same from iVProg. Here, the snippet also includes the parsing of the assignment description file on line 4. Given the that the assignment description is a text file, one can use the **JSON** native JavaScript library to convert the text into a JSON object. In this instantiation, a button named *grade it!*(Figure 5.7) was created to run VCAT AAT. The button executes the code present in Listing 5.7 and shows the feedback and final grade in the console and also in a alert window. In the next chapter the application of the proposed solutions which represents the final part of this three part design.

Program 5.7 Code snippet for performing automatic assessment for Blockly-based visual language.

```
1    const c_text = translate();
2    const ast = vcat.SemanticAnalyser.analyseFromSource(c_text);
3    // vcatConsole is an instance of Console created earlier, it works as both
         input and output
4    const assignmentDescription = JSON.parse(/*this should be the JSON file
         contents*/);
5    const assessment = new vcat.Assessment.AssessmentRunner(ast,
6        assignmentDescription, vcatConsole);
7    assessment.runTest().then((final_grade) => {
8        alert(‘You grade: ${final_grade}‘);
9    }).catch(console.error);
```

Chapter 6

VCAT Evaluation

Section 5.3 last chapter described the necessary steps for a given visual programming language to instantiate VCAT. The instantiation process was executed for two different visual programming languages: iVProg and a language designed with the Blockly framework. This demonstrates that the proposed model is general enough to work with different visual programming languages. This chapter will present the application of the solution through an experiment developed to evaluate if the changes in VCAT's output matching algorithms in comparison to output matching traditional implementations actually improved the feedback provided. The analysis is done using a questionnaire answered by the students and a quantitative analysis using students grades in iVProg and VPL exercises.

6.1 Experiment

The experiment was conducted with students from an introductory programming course in the summer program at the Institute of Mathematics and Statistics (IME-USP). The summer program at IME-USP is run every year from January to February, and it offers a range of courses including introductory programming courses. The classes were conducted not conducted by the author, but he was part of the support group for the whole course with some interactions with the students answering questions on programming logic. A total of 104 students were divided into two classes, one in the morning and another in the evening from January 4th to February 26th. The course in general had a load of 60 hours with lectures from Monday to Friday, where one day of the week was dedicated to help students with their doubts by a teaching assistant. The morning the classes went from 08:00 to 10:00 and the night classes from 19:00 to 21:00. The morning class was compromised of 37 students while the night class had 67 students enrolled. Also, both classes were lectured by the same teacher and the teaching assistant was also the same for both. The course structure was the same as previous editions where the following topics are discussed: basic computer architecture, input & output command, expressions, conditional commands, loops, functions, vectors and matrices.

For each subject exercises using both tools, iVProg and VPLwere created. In the beginning, the exercises were equally distributed between iVProg and VPL, and gradually had the proportion of VPL exercises increased until they were the only ones. This change happened every time a new subject was introduced and was also part of past editions. Thus, the only intervention done in the course was the use of iVProg with VCAT instantiated, everything else went as it would have in previous editions. It is important to note that all classes were delivered online via Moodle and the BigBlueButton virtual class room as a result of the Covid-19 pandemic.

During the first 4 weeks of the course, the students were introduced to iVProg and then to an AAT plug-in for Moodle called Virtual Programming Lab (VPL). Right after this 4 week period, the students had to answer a questionnaire designed to assess if the developed artifact and its improvements successfully achieved the desired goals regarding AAT improvements.The answers were all voluntary and the students were informed about the nature of the research and the contributions it could produce. The questionnaire pages inside the Moodle course also contained the terms of how the data collected would be used and that they could withdraw their consent at any time. From the 104 students who were participating in the introductory course, a total of 57 effectively answered both questionnaires, one for each tool used. Out of this total, 36 students were from the evening class and 21 from the morning class. The next section will present the questions used to evaluate the artifact and their motivation.

6.2 Questionnaire

The questionnaire was composed of 8 questions each intended to collect data regarding the tools being used, iVProg or VPL, and how the students perceived the quality of the feedback provided. It was inspired by Savi *et al.* (2011) evaluation model for educational games. In their model, Savi *et al.* (2011) proposed a set of 27 questions designed to assess students' motivation, user experience and learning. Although iVProg and VPLare not educational games they are still educational software developed with a view of aiding the teaching-learning process.

Thus, the questionnaire applied in this research used Savi *et al.* (2011) original proposal as a starting point. New questions were devised regarding user experience of the tools and how it affected the learning process. Additionally, questions on the students' trust in the automatic assessment were also included, along with questions on short term learning. Table 6.1 presents the questions, translated to English, used to evaluate iVProg and VPL.

It is important to emphasize that the questions had their text adapted accordingly when used to evaluate VPL. Also, there was an discursive question related to Q1 where the students were ask to give more context to the answer they gave. This was motivated from past experiences where students would assess the tool not based on the tool itself, but by associating it to the topics being learned with the tool, in this case algorithms and programming.

Each question was answered using a five points Likert-like scale as proposed by the original model (Savi *et al.*, 2011). Each answer was assigned a value ranging from −2 to 2 where a negative value does not inherently mean a negative answer and vice-versa. The resulting score was calculated by averaging each question's answers. Thus, the interpretation is directly related to the answer format: the closer a question average is to 2

Question	*Text*	*Purpose*
Q1	The tool iVProg for the construction and execution of algorithms using blocks was easy to understand.	Ease of usage
Q2	In the exercises that the automatic assessment tool detected an error, how do you classify the utility of this functionality?	AAT feedback quality
Q3	It was easy to start using iVProg as a study tool.	Usage of the tool as a study aid
Q4	While solving exercises using iVProg I felt confident that I was learning.	Confidence in the tool as a learning aid
Q5	How do you classify the difficulty level of the exercises in the current block?	Perceived difficulty
Q6	Using iVProg to solve exercises made it difficult or easy ?	Tool effect on the perceived difficulty
Q7	I had positive feelings of efficiency while solving exercises using iVProg.	Tool effect on perceived efficiency
Q8	In your perception, how much do you believe iVProg with automatic assessment contributed to your algorithm learning?	Short term learning

Table 6.1: *English translated version of the questionnaire used to assess iVProg and its AAT*

the better it was evaluated by the students (Savi *et al.*, 2011). Moreover, the questionnaire format also allowed for a comparison of how the students perceived different aspects in the tools iVProg and VPL. Next section will present the results of the assessment of both tools and also a breakdown of each question comparing how each tool of evaluated by the students.

6.2.1 Results

After aggregating the data from both classes, the average for the answers for each tool was calculated as it is shown in Figure 6.1. As displayed in the figure, both tools were positively evaluated, with some question reaching an average around 1.5.

Starting with Q1, it is possible to see that both tools were evaluated almost identically regarding the ease of usage, with VPL getting a slightly higher score. In Table 6.2, its is possible to see the distribution of each score for Q1 for both tools. On the comments provided, the general sentiment was that both tools were intuitive. However, some students felt that they needed some initial guidance to properly use iVProgwhich aligns with the data shown in Table 6.2.

When evaluating the quality of AAT feedback (Q2), iVProg got a higher score than VPL, with the difference being above 0.30. Looking at the answers distribution, it shows that more than 70% of the students thought the AAT feedback from iVProg was really good against ≈ 57% from VPL. Regarding the usage of the tools as a study aid (Q3), both were similarly evaluated with iVProg receiving a slightly higher score. On Q4, the students were asked to assess how confident they felt they were learning when using the tools. Both tools scored higher than 1.5 but VPL got a higher score even though the difference is not so big (≈ 0.19). Looking at the values in Table 6.2, we can see that more than 80% of

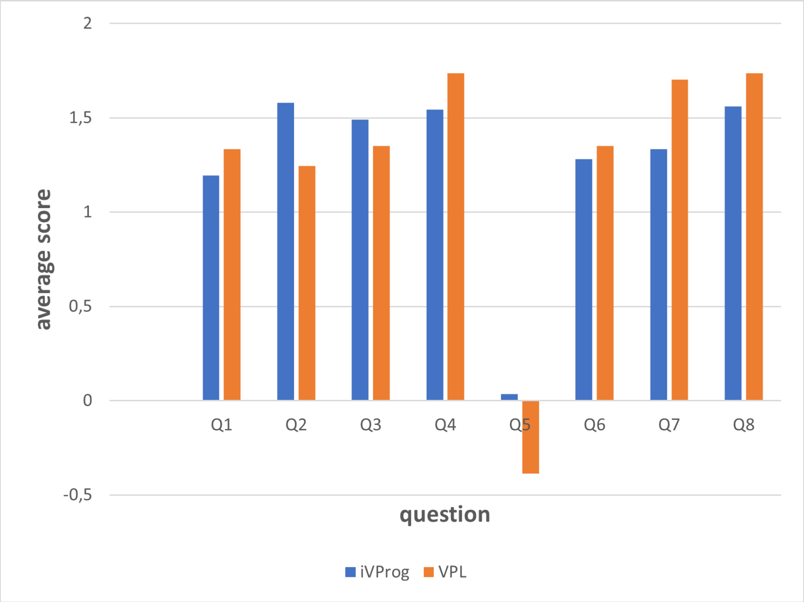


Figure 6.1: *iVProg and VPL scores comparison*

Question	*-2*	*-1*	*0*	*1*	*2*
Q1	3.5% — 1.8%	1.8% — 0%	8.8% — 10.5%	43.9% — 38.6%	42.1% — 49.1%
Q2	0% — 0%	5.3% — 10.5%	5.3% — 12.3%	15.8% — 19.3%	73.7% — 57.9%
Q3	0% — 1.8%	5.3% — 5.3%	5.3% — 5.3%	24.6% — 31.6%	64.9% — 56.1%
Q4	0% — 0%	3.5% — 1.8%	5.3% — 5.3%	24.6% — 10.5%	66.7% — 82.5%
Q5	7% — 17.5%	21.1% — 33.3%	38.6% — 22.8%	28.1% — 22.8%	5.3% — 3.5%
Q6	0% — 0%	3.5% — 3.5%	12.3% — 12.3%	36.8% — 29.8%	47.4% — 54.4%
Q7	1.8% — 0%	5.3% — 0%	14% — 8.8%	15.8% — 12.3%	63.2% — 78.9%
Q8	0% — 0%	0% — 1.8%	12.3% — 3.5%	19.3% — 14%	68.4% — 80.7%

Table 6.2: *Occurrence table for the percentage of answers' scores for the tools iVProg — VPL given by the 57 students*

the students felt very confident using VPL to solve the exercises.

The only question to score lower than 1 was Q5, regarding the perceived difficulty of the exercises of the period for the given tool. For this question a -2 implies that the student found the exercise very difficult, while a 2 implies very easy. Even though the exercises were designed with similar difficulty in mind, students considered iVProg exercises to be more neutral. VPL on the other hand was considered slightly difficult with an average score of ≈ -0.4. Additionally, when examining the perceived difficulty of the exercises taking the tool used to solve them into consideration (Q6), the students' answers suggests that the tools did not affect them negatively. The data suggests that most students perceived that the tools actually aided them to solve the exercises. Both iVProg and VPL scored similarly with a negligible difference. In VPL's case, more than 50% of the students considered that the tool made it very easy to solve the exercises.

On Q7, the students were asked about how efficient they felt when using the tools to solve the exercises. In general both tools scored well. Nonetheless, VPL score was higher than iVProg with a difference around ≈ 0.36. Also, around 78% of the students classified VPL as the tool the felt most efficient using. The last question was about long term learning of algorithms and how the students felt the tool used helped them achieve that. Once again the tools scored really well, with VPL getting a slightly higher score. Additionally, approximately 80% of the students thought the using VPL contributed a lot to their algorithm learning. The next chapter brings some discussions on the results found and some other insights gathered from the data collected.

6.3 Exercises' grades analysis

In order to gauge the effects of the improvements of the output matching feedback in VCAT and also the use of visual programming languages with automatic assessment from a quantitative perspective, the author decided to analyse the grades students scored in iVProg and VPL exercises. The exercises chosen for this analysis were the ones that coincided with the week the students answered the questionnaire. Since there was no limitations on the number of submissions a student could make until he got the max grade for a given exercise, some treatment of the final grade was needed to take into account how long it took to the student to get that score. The following formula was used to compute the final grade (G_{final}) from the students original grade:

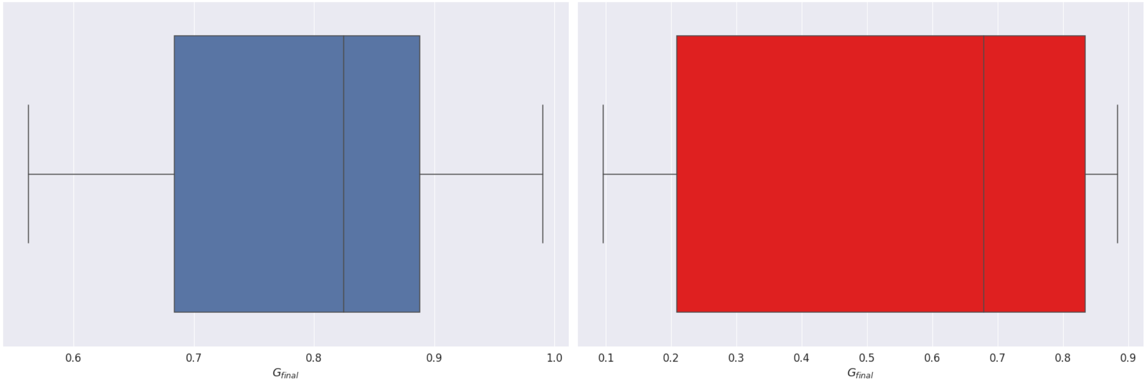
$$G_{final} = grade * (1 - \frac{1}{\Delta T}) \tag{6.1}$$

where $grade$ is the students grade for the exercise, and ΔT is defined as the difference between the students' recorded timestamps for last and initial interaction of the tool in minutes (L. d. Souza *et al.*, 2021). It is important to note that ΔT is strictly higher than 1. ΔT represents the amount of time of tool usage the student had until he solved the problem.

The dataset consisted of 9 exercises from VPL, they were the exact same for both classes. In iVProg's case, 11 exercises were included from the morning class while the evening class had only 9 included. It is important to note that only the data for the students who answered both questionnaires and had submissions on all exercises were processed. The total of students from both classes who met the criteria was 16. Once the dataset was defined, it is was pre-processed following the same steps presented in L. d. Souza *et al.*, 2021 to make sure all timestamps and grades were aligned.

At first, a visual analysis was performed on the data to check the value distribution as shown in Figure 6.2. The box plots shows that iVProg's scores were more concentrated on the middle with some variance on the extremes(Figure 6.2a), with a minimum value above 0.5 and maximum ≈ 1.0. On the other hand, VPL had a bigger spread of scores with some small variance on the extremes, specially towards the lower whisker(Figure 6.2b). The lowest value was slightly above 0.2 and the highest close to 0.9. Comparing the mean values between the two box plots, it is noticeable that iVProg's has a higher mean value compared the VPL. In fact, iVProg's scores in general are much higher than VPL.

To confirm that the behavior seen in the box plots were not a random event, the non-parametric Wilcoxon signed-rank test was performed in the data. The test was chosen



(a) *Box plot for iVProg's G_{final} scores*

(b) *Box plot for VPL's $G_{final}scores$*

Figure 6.2: *Box plots for both tools G_{final} scores to compare the values distribution shape*

since the data is not normally distributed and both samples came from the same population. For this test, the following hypothesis were defined:

- H_0: "iVProg's G_{final} grades are less than VPL's"(null hypothesis).
- H_1: "iVProg's G_{final} is greater than VPL's" (alternative hypothesis);

The test result produced a statistic of 123.0 with p-value = 0.00135. Therefore, it is possible to reject H_0 since there is a probability of $\approx$ 0.002 for the results seen to be the product of random chance. So, it can be concluded that there is a statistically significant difference between iVProg's and VPL's G_{final} where iVProg's grades are higher.

Taking into consideration that Equation 6.1 encodes in G_{final} the time spent in the tool, the results show that students were able to solve the exercises much faster using iVProg when compared to VPL. This indicates that solving exercise using iVProg was less complicated than solving them with VPL. The discussions on these results will be presented on next chapter.

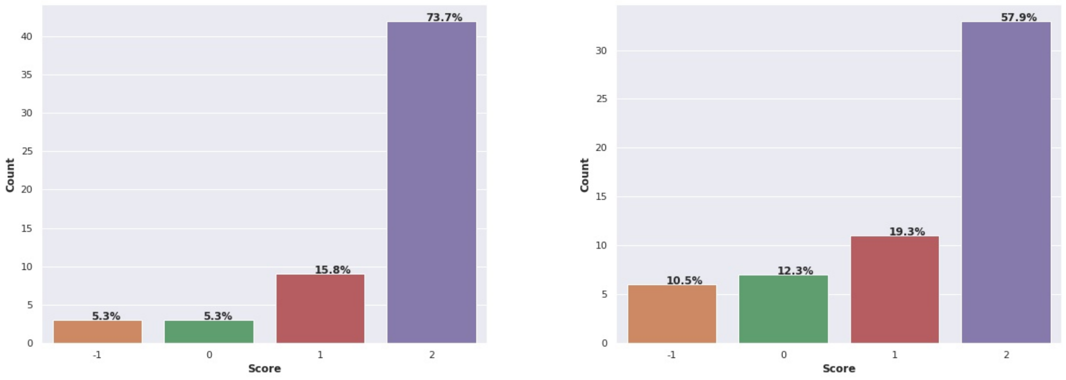
Chapter 7

Discussions

This chapter discusses the data collected during the application of VCAT. It starts with the analysis of the questionnaire answers followed by the analysis of the quantitative data of the grades comparisons. The data shows that the research objectives were achieved and provides some interesting insights on students perceptions of visual programming compared to traditional text programming.

7.1 Questionnaire answers

By analyzing the data presented in Figure 6.1, it is possible to conclude that one of the main objectives of this research has been achieved. The students felt that the feedback provided by iVProg was much better to inform them of the issues in the code when compared with VPL.



(a) *Occurrence graph for Q2 answers for iVProg* **(b)** *Occurrence graph for Q2 answers for VPL*

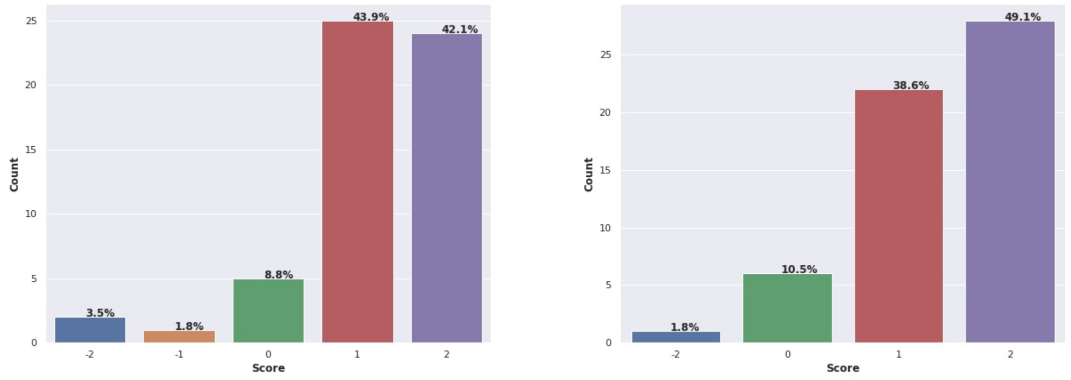
Figure 7.1: *Occurrence graph for Q2 regarding AAT capabilities of iVProg and VPL from very useless(-2) to very useful(2)*

Figure 7.1 presents the distribution of each score for the question between both tools. It is relevant to note that VPL uses by default output matching dynamic assessment as an AAT strategy. Having more than 70% (Figure 7.1a) of the answers scoring 2 for iVProg indicates that the improvements in the output matching algorithm and feedback provided in iVProg achieved the goal of being more helpful and clear when compared to VPL. Specially when the inspiration for the proposed improvements came from past experience with regular students using VPL and their complaints.

Although VPL was also well rated (Figure 7.1b), it received more neutral and negative score when compared to iVProg. The author attributes this to the known problems of output matching assessment (Ala-Mutka, 2005; Ihantola *et al.*, 2010; C. K. Poon *et al.*, 2016), notably with the difficulty some of the summer course students reported on understanding the feedback provided by VPL. The main complaint the students had were that VPL errors message were not clear enough.

In the literature, this is issue is usually related to the fact that a number of programming languages used in some courses, in this case C, provides error messages not beginner friendly (Sykes, 2007; Meerbaum-Salant *et al.*, 2010). VCAT had, on the other hand, improved how the output matching result is presented to the students and how it is evaluated, as described in Chapter 4. Moreover, it also attempted to provide more helpful error messages that could point the student towards the source of the problem.

When asked about how easy it was to use each tool(Q1), the students considered both easy enough to use. In this aspect, VPL scored slightly higher, which the author considers an expected outcome. Even though, students have commented that both tools are quite easy to use, VPL has a very clean user interface since one of its main points is to behave like a basic text editor where students can code.



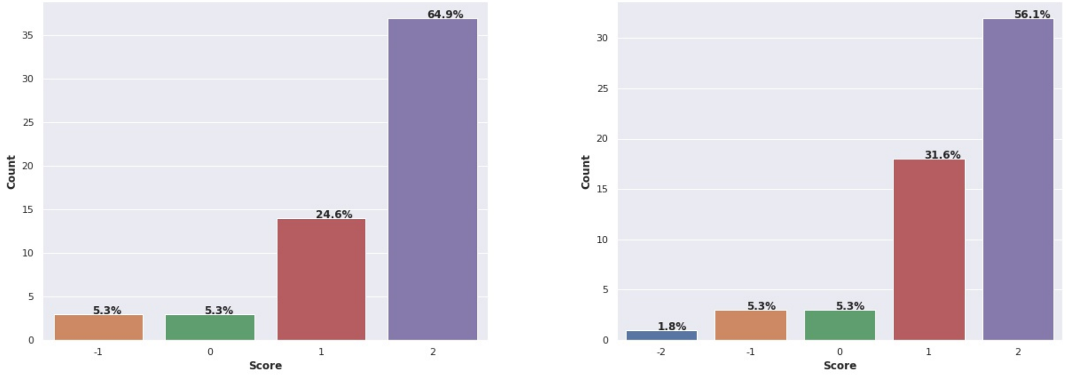
(a) *Occurrence graph for Q2 answers for iVProg* **(b)** *Occurrence graph for Q2 answers for VPL*

Figure 7.2: *Occurrence graph for Q1 on iVProg and VPL ease of usage from very hard(-2) to very easy(2)*

Figure 7.2 displays how students scored each tool for Q1. Both tools had some negative feedback given, however the positives were much higher. VPL was considered very easy to use by almost 50% of the students while on iVProg's case it they were divided between an easy and very easy evaluation. By its own purpose, iVProg is a different kind of tool with much richer user interface and different kinds of interactions when compared to VPL. As a consequence,this require students to take some time to get used to. Students' comments shows that they could use iVProg much better after the teacher gave them a basic introduction. This experience regarding iVProg was also reported in a prior study regarding the best interface style for the tool (Félix *et al.*, 2019). By analysing interaction logs and heat maps from students clicks confronted with data collected using NASA-TLX usability questionnaire. The authors were able to identify some bottlenecks in the interface and propose improvements. Although some improvements were made since then, their effects and impacts on user experience have not been validated yet.

Additionally, there is a matter of programming language used. Although visual programming can potentially facilitate the learning process of programming concepts (Sáez-López *et al.*, 2016; Sykes, 2007), coding with text is also regarded as more convenient once you learned the basics (Brandao *et al.*, 2012; Booth and Stumpf, 2013). This convenience can help explain the likelihood of students considering VPL much easier to use, specially taking into account that they enrolled in a C programming course.

Also, due to how the course was organized the exercises from iVProg were always presented first and VPL second. This was not intentional but it led to students solving iVProg exercises first and then VPL. However, the results are aligned with other researches regarding the effect of visual programming languages on students learning, specially when visual programming comes first (Brandao *et al.*, 2012; Meerbaum-Salant *et al.*, 2010; Sykes, 2007).

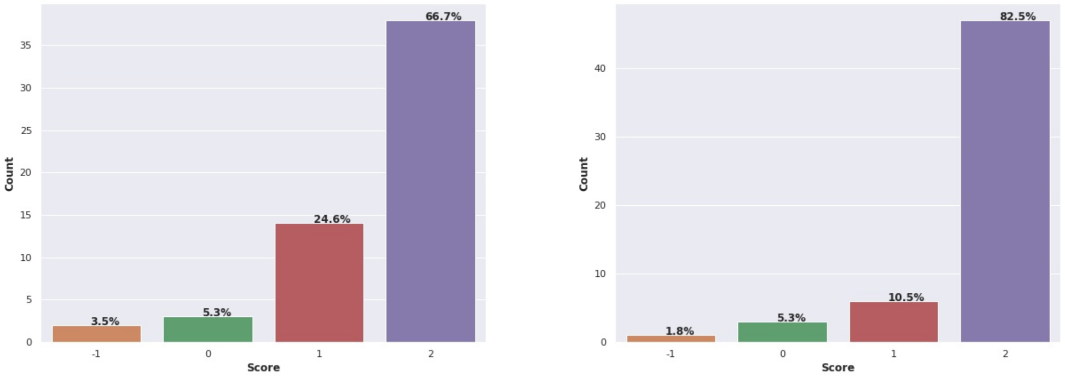


(a) *Occurrence graph for Q3 answers for iVProg* **(b)** *Occurrence graph for Q3 answers for VPL*

Figure 7.3: *Occurrence graph for Q3 on iVProg and VPL usage as a study tool from very hard to use (-2) to very easy(2)*

Another positive effect of visual programming can be seen on the results related to how easy it was to use the tools as a study aid (Q3). iVProg got mostly positive results as shown in Figure 7.3a and averaged close to 1.5, slightly higher than VPL. As mentioned before, the literature already reports on the effects of using visual programming as a tool to ease programming learning (Meerbaum-Salant *et al.*, 2010; Boshernitsan and Downes, 1997) which can explain the results see in Figure 7.3 . This benefit is also reinforced when most approaches of developing CT uses programming and visual programming (Hsu *et al.*, 2018). Even though there is no other data that could corroborate the assumptions, past experiences in the classroom suggest that the students sometimes use iVProg as a sort of *playground*, where they can try their ideas. During the summer course, iVProg and VPL exercises were similar by design to enhance the use visual programming as a study tool.

VPLscores does not come as a surprise. Considering that the course aimed to teach programming using C language, it is natural that the students would feel that VPL was also very suitable for studying. Besides being able to write C code through VPL, they were also able to use the AAT and get feedback on their solutions. As a consequence, the author attributes the lower scores seen in Figure 7.3b to the difficulties students usually report on understanding the feedback provided by VPL.



(a) *Occurrence graph for Q4 answers for iVProg*

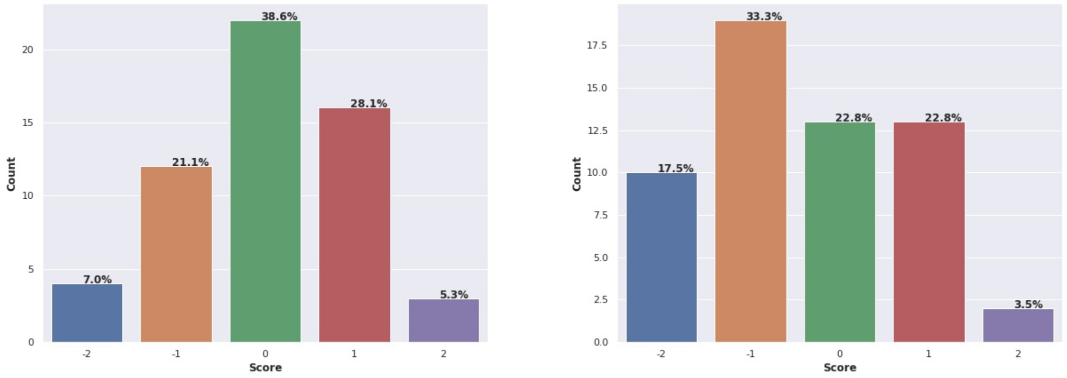
(b) *Occurrence graph for Q4 answers for VPL*

Figure 7.4: *Occurrence graph for Q4 on if they felt confident when using iVProg and VPLfrom totally disagree(-2) to totally agree(2)*

The tools assessment from the students perspective on how confident they felt while using them (Q4) show that both scored really well. The majority of students felt very confident when using any of the tools with average score above 1.5. However, VPL got a "very confident" score from more than 80% of the students. As discussed in Q3, the author associate the scores presented in Figure 7.4 with the fact that this course was aimed at the C language. By using VPL, the students most likely felt that they were *learning more* when compared to iVProg.

Figure 7.5 shows how each tool scored regarding the perceived difficulty of the programming exercises solved using them (Q5). For around 38% of the students perceived the programming exercises when using iVProg to be of adequate difficulty. This is significantly higher to the 22% achieved by VPL even though the exercises were purposely similar, sometimes the exact same. This perception again can be associated with the findings of other investigations regarding the ease of use of visual programming languages for beginners (Sykes, 2007; Meerbaum-Salant *et al.*, 2010).

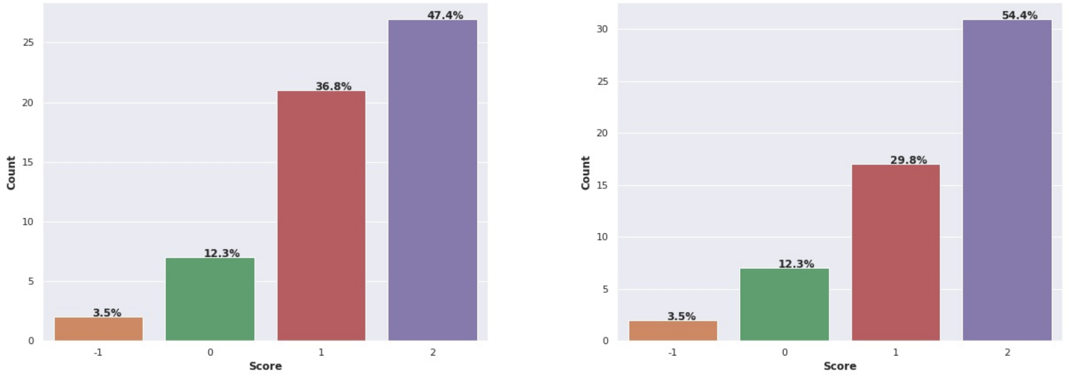
VPL scores of around 50% of the students perceiving the exercises in the tool to be more difficult than their iVProg counterparts also aligns with the findings from other researchers like Gomes and Mendes, 2007 and Vainio and Sajaniemi, 2007. As the literature suggest, the use of commercial languages i.e C and Java can pose barriers to beginners especially when dealing with error messages and the complex syntax. The scores present in Figure



(a) *Occurrence graph for Q5 answers for iVProg* **(b)** *Occurrence graph for Q5 answers for VPL*

Figure 7.5: *Occurrence graph for Q5 on the students perception of exercise difficulty when using iVProg and VPL from very challenging(-2) to very easy(2)*

7.5b can also be related to the scores presented in Figure 7.1b on the AAT capabilities of VPL. Since some students felt that the feedback provided by VPLś AAT was not as helpful as iVProgś, this could have some impact on their perceived difficulty of the exercises when errors occurred.

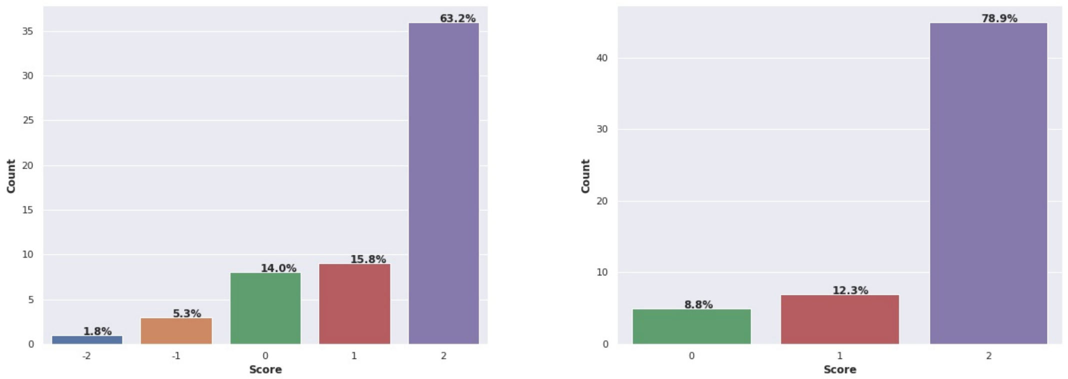


(a) *Occurrence graph for Q6 answers for iVProg* **(b)** *Occurrence graph for Q6 answers for VPL*

Figure 7.6: *Occurrence graph for Q6 on effects of each tool over the students perception of exercise difficulty from much harder(-2) to much easier(2).*

While Q5 aimed to assess the perceived difficulty of the exercise offered through each tool, Q6 on the other hand attempted to assess the effect of the tools themselves on this perceived difficulty. Figure 7.6 presents how each tool scored regarding their effect on perceived difficulty.] Curiously, even though the majority of students said the exercise difficulty was appropriate, not being too difficult nor too easy, they felt that the tools actually helped make the exercises easier to solve. In general, VPL score slightly higher than iVProg in this regard, having more than 50% of the respondents suggesting that it made solving the exercises much easier against around 47% from iVProg.

The author associate the fact that VPL had more ´´much easier” scores to the expected transition from visual programming environment to textual coding. Even though the exercises presented in iVProg were considered more balanced in Q5(Figure 7.5) when compared to VPL, the more the students became familiar with developing algorithms the more comfortable they became with textual code. The results from Q2(Figure 7.1) on ease of usage also suggests that a greater level of comfort with textual code could be a potential explanation for the results. The next question to be discussed can also be linked to this effect of the tool on the perceived difficulty.



(a) *Occurrence graph for Q7 answers for iVProg*

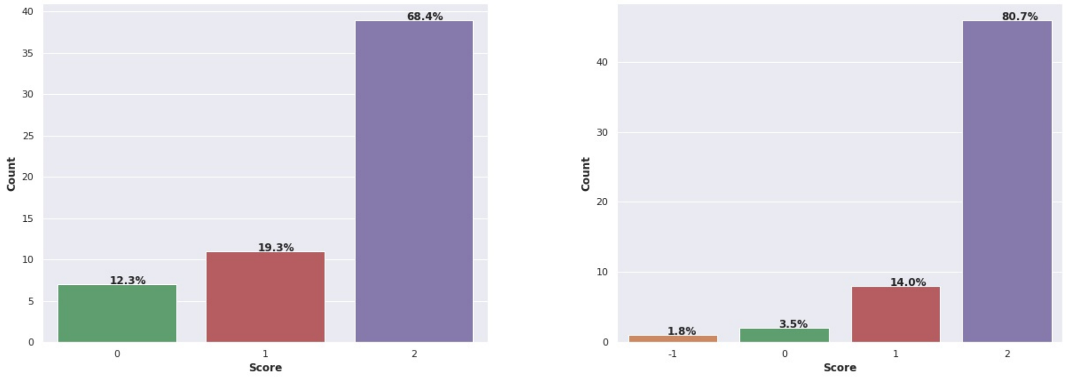
(b) *Occurrence graph for Q7 answers for VPL*

Figure 7.7: *Occurrence graph for Q7 on if the students felt efficient while using iVProg and VPL from totally disagree(-2) to completely agree(2).*

In Figure 7.7, it is possible to notice that the majority of the students felt efficient while using both tools. However, almost 80% of the students agreed that using VPL made them feel more efficient. This can be linked to the results presented for Q6 above and also to the already mentioned fact that the course was aimed to teach C language. The fact that most respondents also solved the exercises in iVProg first to then attempt to solve them in VPL, since iVProgś exercises always came first in the page, also shows the potential benefits of visual programming languages. Having already solved comparable exercises that requires a similar line of thought may have contributed to the perceived higher efficiency felt while using VPL.

Figure 7.8 presents the occurrence graph for the students answer for Q8. More than 50% of the students agree that the tools used contributed significantly for their algorithm learning. The algorithm learning from VPL was significant for 80% students against around 68% from iVProg. As with the last three discussed questions, Q8 also indicates the potential influence of the usage of visual programming systems before solving problems using a textual language.

This indicates the potential of employing visual programming as a step in conventional text-based programming courses as reported by other researches like Booth and Stumpf, 2013; Brandao *et al.*, 2012. Both system had an average score for Q8 above 1.5 showing that the students perceived that they contributed in some way for their algorithm learning. This is can also be a result of the already mentioned circumstances of the order of the exercises



(a) *Occurrence graph for Q8 answers for iVProg*

(b) *Occurrence graph for Q8 answers for VPL*

Figure 7.8: *Occurrence graph for Q8 on the students perception the contribution of the tools(iVProg & VPL) for algorithm learning from not contributed at all (-2) to contributed significantly (2).*

being iVProg first then VPL, and the fact that the students enrolled in a C programming course, thus influencing their perception of what means to learn algorithm.

7.2 iVProg and VPL grade analysis

The G_{final} data for iVProg and VPL presented in Figure 6.2 is well aligned with similar studies in the literature. The data is aligned with other studies findings on ease of usage of visual programming language when compared to traditional text languages (Sáez-López *et al.*, 2016; Meerbaum-Salant *et al.*, 2010; Brandao *et al.*, 2012). Additionally, the results can also be related to the reported difficulties with syntax which are common on textual languages (Gomes and Mendes, 2007). However, the main insight provided by the data, given the nature of this research, is that the improvements of the feedback speaks to other results on the questionnaire. It aligns with the insights provided by the answers for Q2, showing that the improved feedback has positively impacted on students ability to solve the exercises faster. G_{final} data also provides more context to the answers provided for Q5 regarding the perceived difficulty of the exercises where most students felt that solving them in VPL was harder when compared to iVProg. It indicates that this perceived easiness could be related to the speed which they could solve the problems.

Nonetheless, it contrasts with answers for Q7 where the majority of the students felt they were more efficient using VPLin comparison with VPL. Again, the need to write code directly in text could also explain this perception since typing an algorithm can take more time than visually building it, affecting the whole perception. And as pointed out before, the fact that they were enrolled in a C programming course could also have affected the answers to this question given that their main purpose would be to program in C language. The same analysis can be applied to answers of Q6, although the difference between both tools are not that expressive in this case.

Chapter 8

Conclusions

This chapter will present a condensed view of the results and objectives of this research. Like the better perception of the students for the feedback provided and how easier it was to solve the problem using a visual language. Also, potential future works pathways are also presented like improving the semantic analyser feedback and implementing a code debugger in VCAT. Moreover, the possibility of adding an artificial intelligence with a large language model to better communicate with the students is also discussed.

8.1 Final considerations

This research presents an automatic assessment model for visual programming system called VCAT which allows visual programming languages to use automatic assessment methods to evaluate students code. Besides that, VCAT also has improved the process and feedback for the output matching algorithm, dealing better with numeric and textual outputs. Another improvement is that VCAT's implementation uses color coded outputs to the student can better identify what is wrong in the output produced by his solution and can easily tackle the source of the problem.

The model was instantiated for two different visual programming languages showing that it is general. Although it has some limitation regarding variables types and I/O options, it is extensible and can be modified to support new functionalities. The experiment developed to evaluate the model and the improvements in the feedback shows that the research objectives were achieved. When comparing iVProg's output matching solution to the traditional implementation in VPL, the students perceived iVProg's feedback to be much better in helping them solve the problem.

The data collected also shows that by using visual programming languages, beginners can solve algorithmic problems much easier in comparison to traditional text languages as already reported in the literature (Brandao *et al.*, 2012; Meerbaum-Salant *et al.*, 2010). Also, the data suggests that visual languages can also ease students learning of text-based languages.

8.2 Future work

First, a better and large experiment should be designed to confirm the finding in this research and better assess the effects of the tools themselves in students perceived efficiency. Another relevant improvement is in the feedback of the semantic analyser and of the text-to-ast parser in VCAT. The better we inform students of issues in their code and the whys behind it, the better they can learn and improve. Also, the literature reports that helping students visualize code execution is a very effective tool in helping them learning(Gomes and Mendes, 2007). Thus, implementing a debugger integrated in VCAT is a good addition since this would be available to all languages that instantiate the model.

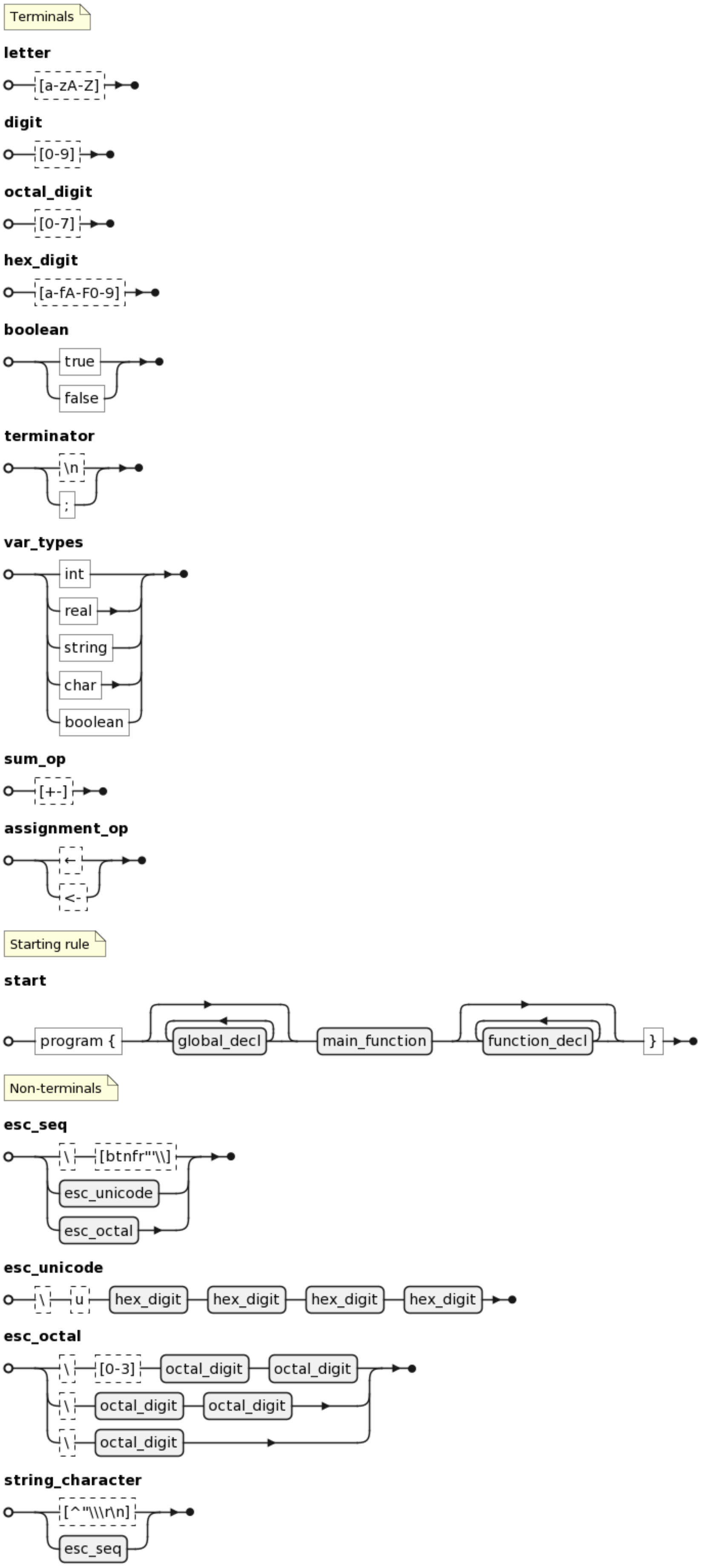
Another interesting addition would be to experiment on integrating a artificial intelligence with a large language model that is capable of better explaining the errors found during code execution. It would behave as intelligent tutor that can communicate clearly and use students historical data in the tool to better support their learning.

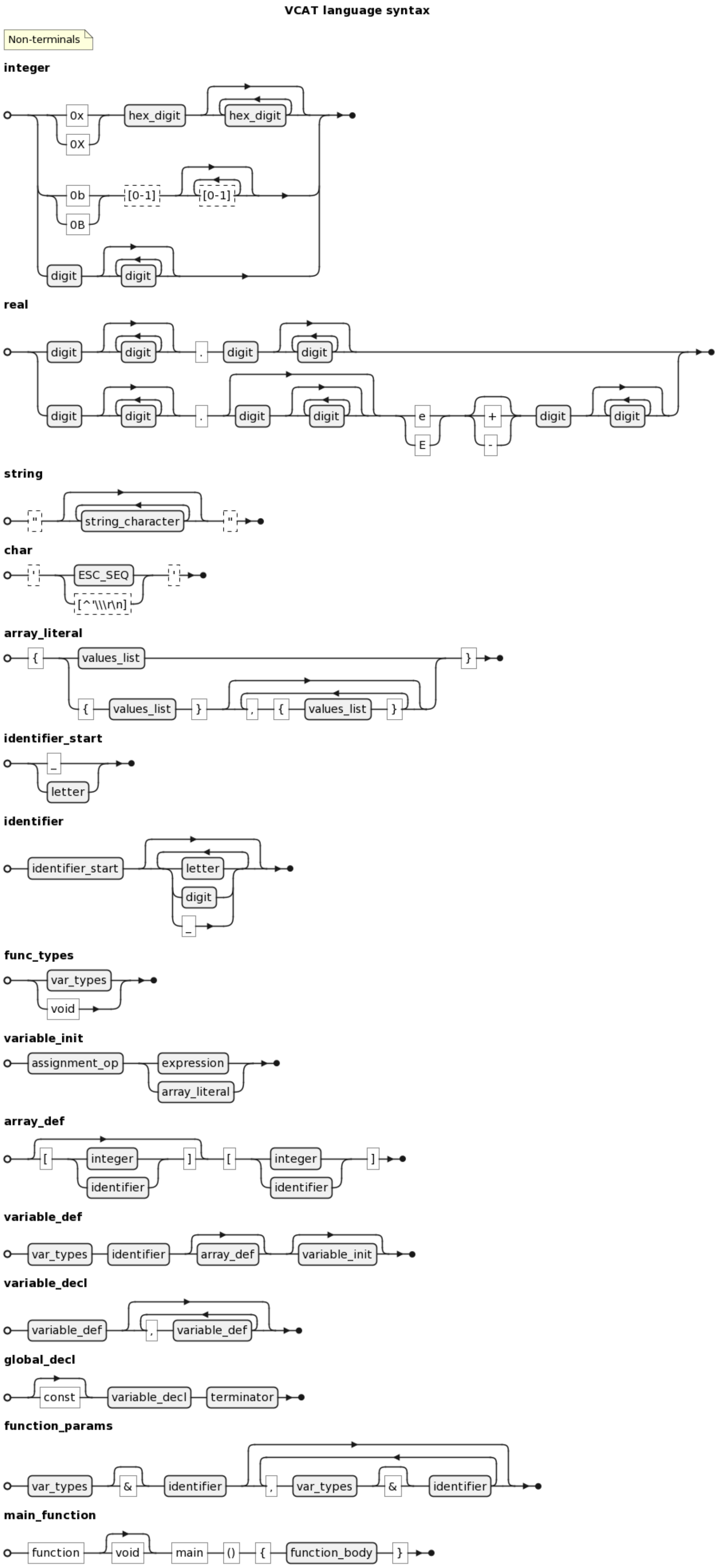
Appendix A

Syntax rules for VCAT textual language presented using EBNF language

This appendix contains the syntax rules for the textual language recognized by VCAT's $\mathcal{M}$. The rules are presented using the metasyntax language Extend Backus-Naur Form (EBNF), widely used to describe programming language syntax.

VCAT language syntax





VCAT language syntax

Non-terminals

integer

real

string

char

array_literal

identifier_start

identifier

func_types

variable_init

array_def

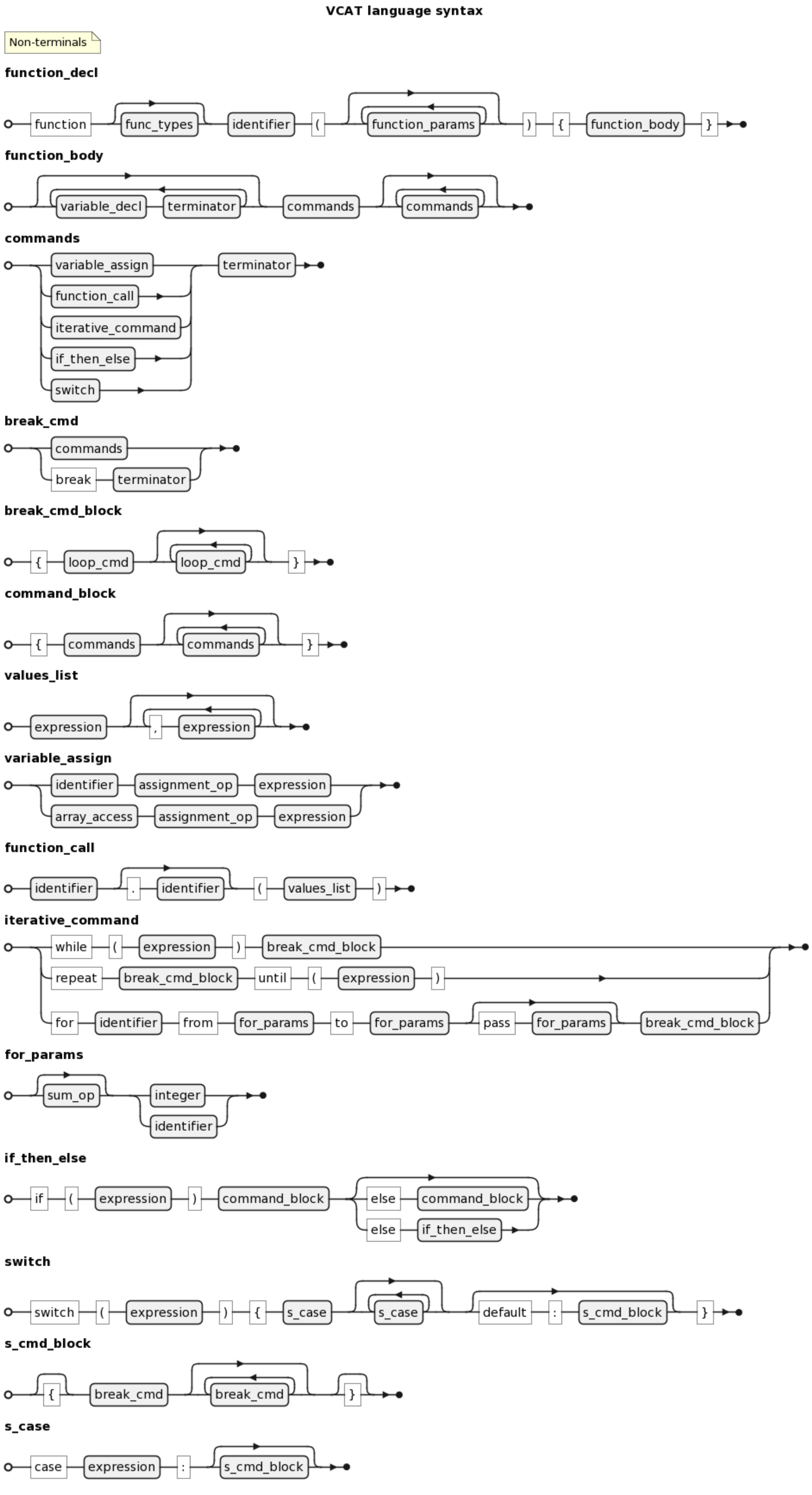
variable_def

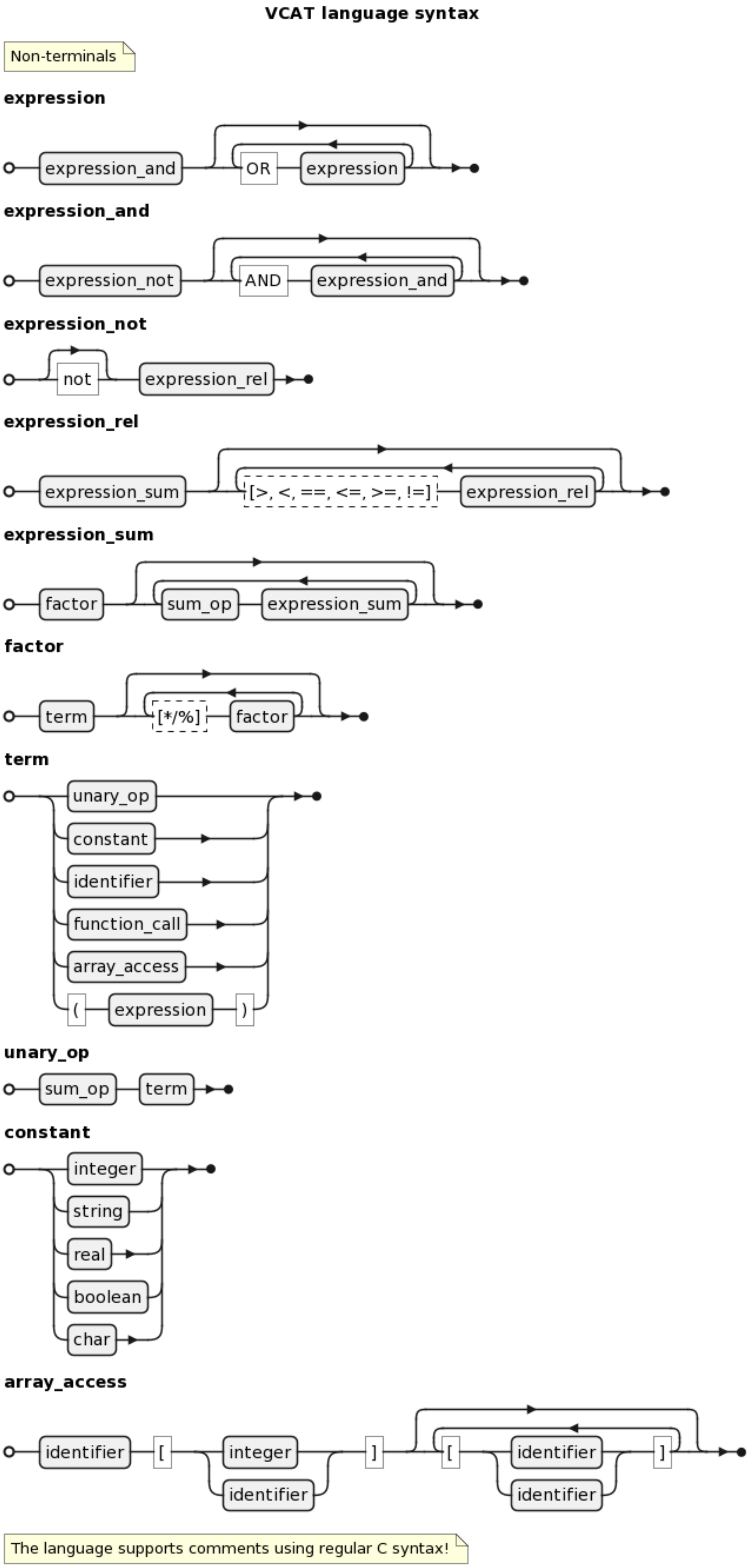
variable_decl

global_decl

function_params

main_function


VCAT language syntax
Non-terminals
function_decl
function
func_types
identifier
(
function_params
)
{
function_body
}
function_body
variable_decl
terminator
commands
commands
commands
variable_assign
terminator
function_call
iterative_command
if_then_else
switch
break_cmd
commands
break
terminator
break_cmd_block
{
loop_cmd
loop_cmd
}
command_block
{
commands
commands
}
values_list
expression
,
expression
variable_assign
identifier
assignment_op
expression
array_access
assignment_op
expression
function_call
identifier
.
identifier
(
values_list
)
iterative_command
while
(
expression
)
break_cmd_block
repeat
break_cmd_block
until
(
expression
)
for
identifier
from
for_params
to
for_params
pass
for_params
break_cmd_block
for_params
sum_op
integer
identifier
if_then_else
if
(
expression
)
command_block
else
command_block
else
if_then_else
switch
switch
(
expression
)
{
s_case
s_case
default
:
s_cmd_block
}
s_cmd_block
{
break_cmd
break_cmd
}
s_case
case
expression
:
s_cmd_block


VCAT language syntax
Non-terminals
expression
expression_and
OR
expression
expression_and
expression_not
AND
expression_and
expression_not
not
expression_rel
expression_rel
expression_sum
[>, <, ==, <=, >=, !=]
expression_rel
expression_sum
factor
sum_op
expression_sum
factor
term
[*/%]
factor
term
unary_op
constant
identifier
function_call
array_access
(
expression
)
unary_op
sum_op
term
constant
integer
string
real
boolean
char
array_access
identifier
[
integer
identifier
]
[
identifier
identifier
]
The language supports comments using regular C syntax!

Appendix B

List of the built-in functions available in VCAT

This appendix presents the list of all built-in functions already available in VCAT organizer by library. Note that the names provided here are the ones available in the English language. In total there are 5 libraries: Conversion, Mathematics, Text, IO and Array. With the exception of the Mathematics library, all functions available in the others libraries can be used in the textual language by only using their names followed by the parameters. For instance, to use the *read* function in the IO library, all that is need is to write *read(var)*.

However, the Mathematics library requires that you prefix the function call with the library name: **Mathematics.abs(-1)**. This method of calling the built-in functions can also be used for the other libraries(*IO.read(var)*) but it is not mandatory. It is important to note that VCAT allows the students to name their functions with the same name as the built-in functions, this is by design. By prefixing the built-in function with its library name, you avoid potential name conflicts. The following sections will present the libraries and their functions.

B.1 Conversion

The Conversion library contains the all functions responsible for type conversion as well as functions responsible for checking if a given string can be converted to a specific type. All functions inside the Conversion library can be used without prefixing them with *Conversion*. Table B.1 lists all the functions names with the parameter type and their description.

B.2 Mathematics

As its name suggests the Mathematics library contains basics mathematical functions like: abs, cos, sin and rnd. The functions in this library can only be accessed by prefixing

Name	*Description*
isReal(val: string)	Returns true if the string can be converted to a floating-point value.
isInt(val: string)	Returns true if the string can be converted to an integer value.
isBool(val: string)	Returns true if the string can be converted to a boolean value
castReal(val: string|int)	Converts the string or int value into a floating-point value.
castInt(val: string|real|char)	Converts the string or real value into an interger. In case of char value, it is converted to corresponding integer value according to the ASCII table.
castBool(val: string)	Converts the string value into a boolean value if the string represents a boolean value (true or false),
castString(val: int|real|char|bool)	Converts the value provided into its textual representation.
castChar(val: int)	Converts the int provided into the corresponding character in the ASCII table.

Table B.1: *Built-in functions available in the Conversion library*

their names with *Mathematics.*. Table B.2 presents all functions available in the Mathematics library.

B.3 Text

The Text library stores all functions related to string and characters manipulation. The functions in this library are presented in Table B.3 and does not require the usage of the prefix *Text*.

B.4 IO

Inside the IO library resides the two very important functions that allows users and $\mathcal{M}$ to interact with each other. The read function allows $\mathcal{M}$ to request the user to input values to the program being executed. On the other hand, the write functions allows $\mathcal{M}$ to communicate to the user data generated by the program, see Table B.4 for more information.

Name	*Description*
sin(angle: real|int)	Calculates the sine of the angle in degrees represented by the numeric value provided.
cos(angle: real|int)	Calculates the cosine of the angle in degrees represented by the numeric value provided.
tan(angle: real|int)	Calculates the tangent of the angle in degrees represented by the numeric value provided.
sqrt(val: real|int)	Calculates the square root of the numeric value provided. It will stop execution if the value is negative.
pow(x: real|int, y: real|int)	Calculates **x** to the power of **y** where **x** and **y** are numeric values.
log(val: real|int)	Calculates the log10 of the numeric value provided. If the value is negative, the function stops execution.
abs(val: real|int)	Returns the absolute value of the numeric input provided.
negate(val: real|int)	Returns the value of the numeric input provided multiplied by -1.
invert(val: real|int)	Calculates $1/val$.
max(values: real[]|int[])	Computes the maximum value in the provided vector of **values**.
min(values: real[]|int[])	Computes the minimum value in the provided vector of **values**.
rand()	Return a random value in the range [0,1] calculated using a predefined seed.
setSeed(seed: int)	Sets the internal seed used in the rand() function to the value provided.

Table B.2: *Built-in functions available in the Mathematics library*

Name	*Description*
substring(str:string, start:int, end:int)	Returns the sub-string inside **str**, starting at position **start** and ending at position (**end**-1). If the range defined by the positions **start** and **end** is not valid for the string **str**, it will return the empty string.
length(str:string)	Returns the length of the string str.
uppercase(str:string)	Converts all characters in string **str** to uppercase.
lowercase(str:string)	Converts all characters in string **str** to lowercase.
charAt(str:string, index:int)	Returns the character in string **str** at position **index**. If **index** is not a valid position for the string **str**, execution will be interrupted.

Table B.3: *Built-in functions available in the Mathematics library*

Name	*Description*
read(var&:*all types*)	Stores into **var** the input provided by the user. Note that **var** here is a reference to a variable created by the student to receive the input.
write(values...:*all types*)	Writes the list of values provided to the underlying output channel. Here, **values** is a variadic parameter which means it can have as many values as desired. This is completely transparent to the students when coding, since from their perspective they are only passing a list of comma-separated values to the *write* function.

Table B.4: *Built-in functions available in the Mathematics library*

References

[Aho 2012] Alfred V. Aho. "Computation and computational thinking". *Comput. J.* 55.7 (July 2012), pp. 832–835. issn: 0010-4620. doi: 10.1093/comjnl/bxs074. url: http://dx.doi.org/10.1093/comjnl/bxs074 (cit. on p. 10).

[Ala-Mutka 2005] Kirsti M Ala-Mutka. "A survey of automated assessment approaches for programming assignments". *Computer Science Education* 15.2 (2005), pp. 83–102. doi: 10.1080/08993400500150747. url: https://doi.org/10.1080/08993400500150747 (cit. on pp. 21–25, 31, 62).

[Araujo *et al.* 2016] E. Araujo, M. Gaudencio, D. Serey, and J. Figueiredo. "Applying spectrum-based fault localization on novice's programs". In: *2016 IEEE Frontiers in Education Conference (FIE).* Oct. 2016, pp. 1–8. doi: 10.1109/FIE.2016.7757727 (cit. on pp. 25, 27).

[Arifi *et al.* 2015] S. M. Arifi, I. N. Abdellah, A. Zahi, and R. Benabbou. "Automatic program assessment using static and dynamic analysis". In: *2015 Third World Conference on Complex Systems (WCCS).* Nov. 2015, pp. 1–6. doi: 10.1109/ICoCS.2015.7483289 (cit. on pp. 22–24).

[Barr and Stephenson 2011] Valerie Barr and Chris Stephenson. "Bringing computational thinking to k-12: what is involved and what is the role of the computer science education community?" *ACM Inroads* 2.1 (Feb. 2011), pp. 48–54. issn: 2153-2184. doi: 10.1145/1929887.1929905. url: http://doi.acm.org/10.1145/1929887.1929905 (cit. on pp. 10, 11).

[Basnet *et al.* 2018] Ram B. Basnet, Tenzin Doleck, David John Lemay, and Paul Bazelais. "Exploring computer science students' continuance intentions to use kattis". *Education and Information Technologies* 23.3 (May 2018), pp. 1145–1158. issn: 1573-7608. doi: 10.1007/s10639-017-9658-2. url: https://doi.org/10.1007/s10639-017-9658-2 (cit. on p. 5).

[Bennedsen and Michael E. Caspersen 2007] Jens Bennedsen and Michael E. Caspersen. "Failure rates in introductory programming". *SIGCSE Bull.* 39.2 (June 2007), pp. 32–36. issn: 0097-8418. doi: 10.1145/1272848.1272879. url: http://doi.acm.org/10.1145/1272848.1272879 (cit. on p. 9).

[Booth and Stumpf 2013] Tracey Booth and Simone Stumpf. “End-user experiences of visual and textual programming environments for arduino”. In: *End-User Development.* Ed. by Yvonne Dittrich, Margaret Burnett, Anders Mørch, and David Redmiles. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 25–39. isbn: 978-3-642-38706-7 (cit. on pp. 11, 19, 63, 66).

[Boshernitsan and Downes 1997] Marat Boshernitsan and Michael Downes. *Visual Programming Languages: A Survey.* 1997 (cit. on pp. 11–13, 63).

[Bosse and Gerosa 2015] Yorah Bosse and Marco Aurélio Gerosa. “Reprovações e trancamentos nas disciplinas de introdução à programação da universidade de são paulo: um estudo preliminar”. In: *Workshop sobre Educação em Computação - WEI.* SBC, 2015 (cit. on p. 9).

[Brandao *et al.* 2012] Leonidas de Oliveira Brandao, Romenig da Silva Ribeiro, and Anarosa Alves Franco Brandao. “A system to help teaching and learning algorithms”. In: *2012 Frontiers in Education Conference Proceedings.* Oct. 2012, pp. 1–6. doi: 10.1109/FIE.2012.6462374 (cit. on pp. 1, 2, 11, 19, 35, 63, 66, 67, 69).

[Brandão *et al.* 2016] Leônidas de Oliveira Brandão, Y. Bosse, and M. A. Gerosa. “Visual programming and automatic evaluation of exercises: an experience with a stem course”. In: *2016 IEEE Frontiers in Education Conference (FIE).* Oct. 2016, pp. 1–9. doi: 10.1109/FIE.2016.7757621 (cit. on p. 5).

[Çağdaş and Stubkjær 2011] Volkan Çağdaş and Erik Stubkjær. “Design research for cadastral systems”. *Computers, Environment and Urban Systems* 35.1 (2011), pp. 77–87. issn: 0198-9715. doi: https://doi.org/10.1016/j.compenvurbsys.2010.07.003. url: http://www.sciencedirect.com/science/article/pii/S0198971510000670 (cit. on p. 3).

[Cardoso *et al.* 2018] M. Cardoso, A. V. de Castro, and A. Rocha. “Integration of virtual programming lab in a process of teaching programming eduscrum based”. In: *2018 13th Iberian Conference on Information Systems and Technologies (CISTI).* June 2018, pp. 1–6. doi: 10.23919/CISTI.2018.8399261 (cit. on pp. 5, 28).

[Michael Edelgaard Caspersen 2007] Michael Edelgaard Caspersen. PhD thesis. Department of Computer Science, 2007 (cit. on pp. 1, 5, 7–9).

[C.-K. Chang *et al.* 2017] Chih-Kaia Chang, Ya-Feib Yang, and Yu-Tzua Tsai. “Exploring the engagement effects of visual programming language for data structure courses”. *Education for Information* 33.3 (Nov. 2017), pp. 187–200 (cit. on pp. 1, 5, 11, 15, 19).

[S. Chang *et al.* 1986] S. Chang, Tadao Ichikawa, and Panos A. Ligomenides. *Visual Languages.* Springer, 1986. isbn: 9781461318057 (cit. on p. 13).

[Conway 1998] Matthew John Conway. “Alice: Easy-to-learn three-dimensional scripting for novices”. PhD thesis. University of Virginia, 1998. url: https://www.learntechlib.org/p/117253 (cit. on p. 14).

[Coughlan *et al.* 2012] T. Coughlan *et al.* “Exploring acceptance and consequences of the internet of things in the home”. In: *2012 IEEE International Conference on Green Computing and Communications.* Nov. 2012, pp. 148–155. doi: 10.1109/GreenCom.2012.32 (cit. on p. 1).

[Dos Anjos *et al.* 2016] Cleverson Sebastião Dos Anjos, Duda Rodrigo, and Sani de Carvalho Rutz Da Silva. “Tecnologias gratuitas para o ensino das disciplinas de algoritmos e programação”. *Revista ESPACIOS| Vol. 37 (Nº 29) Año 2016* (2016) (cit. on p. 35).

[Douce *et al.* 2005] Christopher Douce, David Livingstone, and James Orwell. “Automatic test-based assessment of programming: a review”. *J. Educ. Resour. Comput.* 5.3 (Sept. 2005). issn: 1531-4278. doi: 10.1145/1163405.1163409. url: http://doi.acm.org/10.1145/1163405.1163409 (cit. on pp. 21, 22).

[T. Dufva and M. Dufva 2016] Tomi Dufva and Mikko Dufva. “Metaphors of code—structuring and broadening the discussion on teaching children to code”. *Thinking Skills and Creativity* 22 (2016), pp. 97–110. issn: 1871-1871. doi: https://doi.org/10.1016/j.tsc.2016.09.004. url: http://www.sciencedirect.com/science/article/pii/S1871187116301055 (cit. on pp. 1, 8, 10).

[Eshet-Alkalai 2004] Yoram Eshet-Alkalai. “Digital literacy: a conceptual framework for survival skills in the digital era”. *Journal of Educational Multimedia and Hypermedia* 13 (Jan. 2004) (cit. on pp. 1, 8, 10).

[Falleri *et al.* 2014] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. “Fine-grained and accurate source code differencing”. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering.* ASE ’14. Vasteras, Sweden: ACM, 2014, pp. 313–324. isbn: 978-1-4503-3013-8. doi: 10.1145/2642937.2642982. url: http://doi.acm.org/10.1145/2642937.2642982 (cit. on pp. 28, 29).

[Félix *et al.* 2019] Igor Moreira Félix, Lucas Mendonça Souza, Bernardo Martins Ferreira, and Leônidas de Oliveira Brandão. “A study to build a new visual programming system: fixed or contextual menu?” In: *2019 IEEE Frontiers in Education Conference (FIE).* 2019, pp. 1–8. doi: 10.1109/FIE43999.2019.9028616 (cit. on p. 63).

[Ferrucci *et al.* 1998] Filomena Ferrucci, Genny Tortora, Maurizio Tucci, and Giuliana Vitiello. “Relation grammars: formalism for syntactic and semantic analysis of visual languages”. In: *Visual Language Theory.* Springer, 1998 (cit. on p. 13).

[Flammer 2001] A. Flammer. “Self-efficacy”. In: *International Encyclopedia of the Social & Behavioral Sciences*. Ed. by Neil J. Smelser and Paul B. Baltes. Oxford: Pergamon, 2001, pp. 13812–13815. isbn: 978-0-08-043076-8. doi: https://doi.org/10.1016/B0-08-043076-7/01726-5. url: http://www.sciencedirect.com/science/article/pii/B0080430767017265 (cit. on p. 11).

[Freeman and Hall Giesinger 2017] Adams Becker S. Cummins M. Davis A. Freeman A. and C. Hall Giesinger. *NMC/CoSN Horizon Report: 2017 K–12 Edition*. The New Media Consortium, 2017 (cit. on p. 11).

[Glinert and Tanimoto 1984] E. Glinert and S. Tanimoto. “Pict: an interactive graphical programming environment”. *Computer* 17.11 (Nov. 1984), pp. 7–25. issn: 0018-9162. doi: 10.1109/MC.1984.1658997 (cit. on pp. 11–13, 18).

[Gomes and Mendes 2007] Anabela Gomes and Antonio Mendes. “Learning to program - difficulties and solutions”. In: *International Conference on Engineering Education*. Jan. 2007, pp. 283–287 (cit. on pp. 1, 5, 8, 9, 64, 67, 70).

[Grover and Pea 2013] Shuchi Grover and Roy Pea. “Computational thinking in k–12: a review of the state of the field”. *Educational Researcher* 42.1 (2013), pp. 38–43. doi: 10.3102/0013189X12463051. eprint: https://doi.org/10.3102/0013189X12463051. url: https://doi.org/10.3102/0013189X12463051 (cit. on p. 11).

[Gulwani *et al.* 2018] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. “Automated clustering and program repair for introductory programming assignments”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 465–480. isbn: 978-1-4503-5698-5. doi: 10.1145/3192366.3192387. url: http://doi.acm.org/10.1145/3192366.3192387 (cit. on pp. 25–27).

[Hevner *et al.* 2004] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. “Design science in information systems research”. *MIS Q.* 28.1 (Mar. 2004), pp. 75–105. issn: 0276-7783. url: http://dl.acm.org/citation.cfm?id=2017212.2017217 (cit. on p. 3).

[Hollingsworth 1960] Jack Hollingsworth. “Automatic graders for programming classes”. *Commun. ACM* 3.10 (Oct. 1960), pp. 528–529. issn: 0001-0782. doi: 10.1145/367415.367422. url: http://doi.acm.org/10.1145/367415.367422 (cit. on p. 21).

[Hsu *et al.* 2018] Ting-Chia Hsu, Shao-Chen Chang, and Yu-Ting Hung. “How to learn and how to teach computational thinking: suggestions based on a review of the literature”. *Computers & Education* 126 (2018), pp. 296–310. issn: 0360-1315. doi: https://doi.org/10.1016/j.compedu.2018.07.004. url: http://www.sciencedirect.com/science/article/pii/S0360131518301799 (cit. on pp. 11, 63).

[Ihantola *et al.* 2010] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. “Review of recent systems for automatic assessment of programming assignments”. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli Calling ’10. Koli, Finland: ACM, 2010, pp. 86–93. isbn: 978-1-4503-0520-4. doi: 10.1145/1930464.1930480. url: http://doi.acm.org/10.1145/1930464.1930480 (cit. on pp. 21–24, 62).

[Isaacson and Scott 1989] Peter C. Isaacson and Terry A. Scott. “Automating the execution of student programs”. *SIGCSE Bull.* 21.2 (June 1989), pp. 15–22. issn: 0097-8418. doi: 10.1145/65738.65741. url: http://doi.acm.org/10.1145/65738.65741 (cit. on p. 22).

[Kalibera *et al.* 2014] Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. In: *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE ’14. Salt Lake City, Utah, USA: ACM, 2014, pp. 89–102. isbn: 978-1-4503-2764-0. doi: 10.1145/2576195.2576205. url: http://doi.acm.org/10.1145/2576195.2576205 (cit. on p. 29).

[J.-H. Kim *et al.* 2019] Jeong-Hun Kim, Jong-Hyeok Choi, Uygun Shadikhodjaev, Aziz Nasridinov, and Ki-Sang Song. “Chentry: automated evaluation of students’ learning progress for entry education software”. In: *Big Data Applications and Services 2017*. Ed. by Wookey Lee and Carson K. Leung. Singapore: Springer Singapore, 2019, pp. 51–60. isbn: 978-981-13-0695-2 (cit. on pp. 2, 5, 16, 28, 31, 38).

[Y.-J. Kim *et al.* 2005] Young-Joo Kim, Mi-Young Park, So-Hee Park, and Yong-Kee Jun. “A practical tool for detecting races in openmp programs”. In: *Parallel Computing Technologies*. Ed. by Victor Malyshkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 321–330. isbn: 978-3-540-31826-2 (cit. on p. 40).

[Lahtinen *et al.* 2005] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. “A study of the difficulties of novice programmers”. In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’05. Caparica, Portugal: ACM, 2005, pp. 14–18. isbn: 1-59593-024-8. doi: 10.1145/1067445.1067453. url: http://doi.acm.org/10.1145/1067445.1067453 (cit. on pp. 1, 8, 9).

[Lappalainen *et al.* 2017] Vesa Lappalainen, Antti-Jussi Lakanen, and Harri Högmander. “Towards computer-based exams in cs1”. In: *Proceedings of the 9th International Conference on Computer Supported Education - Volume 2: CSEDU,* INSTICC. SciTePress, 2017, pp. 125–136. isbn: 978-989-758-240-0. doi: 10.5220/0006323501250136 (cit. on pp. 2, 27).

[S. Li *et al.* 2016] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. “Measuring code behavioral similarity for programming and software engineering education”. In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ICSE ’16. Austin, Texas: ACM, 2016, pp. 501–510. isbn: 978-1-4503-4205-6. doi: 10.1145/2889160.2889204. url: http://doi.acm.org/10.1145/2889160.2889204 (cit. on pp. 2, 26, 27).

[Lopes and Garcia 2002] Anita Lopes and Guto Garcia. *Introdução à Programação: 500 Algoritmos Resolvidos.* Rio de Janeiro: Campus; Edição: Cd, 2002 (cit. on p. 7).

[Lye and Koh 2014] Sze Yee Lye and Joyce Hwee Ling Koh. "Review on teaching and learning of computational thinking through programming: what is next for k-12?" *Computers in Human Behavior* 41 (2014), pp. 51–61. issn: 0747-5632. doi: https://doi.org/10.1016/j.chb.2014.09.012. url: http://www.sciencedirect.com/science/article/pii/S0747563214004634 (cit. on pp. 11, 18).

[Maguire *et al.* 2017] Phil Maguire, Rebecca Maguire, and Robert Kelly. "Using automatic machine assessment to teach computer programming". *Computer Science Education* 27.3-4 (2017), pp. 197–214. doi: 10.1080/08993408.2018.1435113. eprint: https://doi.org/10.1080/08993408.2018.1435113 (cit. on p. 28).

[Maloney *et al.* 2010] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. "The scratch programming language and environment". *Trans. Comput. Educ.* 10.4 (Nov. 2010), 16:1–16:15. issn: 1946-6226. doi: 10.1145/1868358.1868363. url: http://doi.acm.org/10.1145/1868358.1868363 (cit. on p. 15).

[Meerbaum-Salant *et al.* 2010] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. "Learning computer science concepts with scratch". In: *Proceedings of the Sixth International Workshop on Computing Education Research.* ICER '10. Aarhus, Denmark: ACM, 2010, pp. 69–76. isbn: 978-1-4503-0257-9. doi: 10.1145/1839594.1839607. url: http://doi.acm.org/10.1145/1839594.1839607 (cit. on pp. 5, 19, 62–64, 67, 69).

[Milne and Rowe 2002] Iain Milne and Glenn Rowe. "Difficulties in learning and teaching programming—views of students and tutors". *Education and Information Technologies* 7.1 (Mar. 2002), pp. 55–66. issn: 1573-7608. doi: 10.1023/A:1015362608943. url: https://doi.org/10.1023/A:1015362608943 (cit. on pp. 8, 9).

[Nascimento *et al.* 2019] Marcos Devaner do Nascimento *et al.* "Which visual programming language best suits each school level? a look at alice, ivprog, and scratch". In: *2019 IEEE World Engineering Education Conference (EDUNINE).* 2019 (cit. on pp. 11, 14–17).

[Noschang *et al.* 2014] Luiz F. Noschang, Fillipi Pelz, Elieser A. de Jesus, and André L. A. Raabe. "Portugol studio: uma ide para iniciantes em programação". In: *Anais do Congresso Anual da Sociedade Brasileira de Computação.* Workshop sobre Educação em Informática. Vol. 1. 2014 (cit. on p. 35).

[Oliveira Brandão and Isotani 2003] Leônidas de Oliveira Brandão and Seiji Isotani. "Uma ferramenta para ensino de geometria dinâmica na internet: igeom". In: *Anais do Workshop de Informática na Escola - WEI 2003.* 2003, pp. 410–421 (cit. on p. 11).

[Osherove 2009] Roy Osherove. *The Art of Unit Testing: With Examples in .Net.* 1st. Greenwich, CT, USA: Manning Publications Co., 2009. isbn: 1933988274, 9781933988276 (cit. on pp. 21, 23).

[Oualline 1997] Steve Oualline. *Practical C Programming: Why Does 2+2 = 5986?* third. Nutshell Handbooks. O'Reilly Media, 1997 (cit. on p. 7).

[Papert 1980] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas.* New York, NY, USA: Basic Books, Inc., 1980. isbn: 0-465-04627-4 (cit. on pp. 1, 8, 10).

[Papert 1996] Seymour Papert. "Computers in the classroom: agents of change". *The washington post education review* 27 (1996) (cit. on pp. 8, 10).

[Pears *et al.* 2007] Arnold Pears *et al.* "A survey of literature on the teaching of introductory programming". In: *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education.* ITiCSE-WGR '07. Dundee, Scotland: ACM, 2007, pp. 204–223. doi: 10.1145/1345443.1345441. url: http://doi.acm.org/10.1145/1345443.1345441 (cit. on p. 21).

[Peffers *et al.* 2007] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. "A design science research methodology for information systems research". *Journal of Management Information Systems* 24.3 (2007), pp. 45–77. doi: 10.2753/MIS0742-1222240302. eprint: https://doi.org/10.2753/MIS0742-1222240302. url: https://doi.org/10.2753/MIS0742-1222240302 (cit. on pp. 2–4).

[Pimentel *et al.* 2019] Mariano Pimentel, Denise Filippo, and Flávia Maria Santoro. "Design science research: fazendo pesquisas científicas rigorosas atreladas ao desenvolvimento de artefatos computacionais projetados para a educação". In: *Metodologia de Pesquisa em Informática na Educação: Concepção da Pesquisa.* Vol. 1. Série Metodologia de Pesquisa em Informática na Educação. Porto Alegre: SBC, 2019 (cit. on p. 3).

[C. K. Poon *et al.* 2016] C. K. Poon, T. Wong, Y. T. Yu, V. C. S. Lee, and C. M. Tang. "Toward more robust automatic analysis of student program outputs for assessment and learning". In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC).* Vol. 1. June 2016, pp. 780–785. doi: 10.1109/COMPSAC.2016.208 (cit. on pp. 5, 25, 27, 37, 62).

[Chung Keung Poon *et al.* 2018] Chung Keung Poon *et al.* "Automatic assessment via intelligent analysis of students' program output patterns". In: *Blended Learning. Enhancing Learning Success.* Ed. by Simon K.S. Cheung, Lam-for Kwok, Kenichi Kubota, Lap-Kei Lee, and Jumpei Tokito. Cham: Springer International Publishing, 2018, pp. 238–250 (cit. on pp. 25, 27, 37).

[Prather *et al.* 2018] James Prather *et al.* “Metacognitive difficulties faced by novice programmers in automated assessment tools”. In: *Proceedings of the 2018 ACM Conference on International Computing Education Research.* ICER ’18. Espoo, Finland: ACM, 2018, pp. 41–50. isbn: 978-1-4503-5628-2. doi: 10.1145/3230977.3230981. url: http://doi.acm.org/10.1145/3230977.3230981 (cit. on pp. 27, 28).

[Rapkiewicz *et al.* 2006] Clevi Elena Rapkiewicz *et al.* “Estratégias pedagógicas no ensino de algoritmos e programação associadas ao uso de jogos educacionais”. *RENOTE* 4.2 (2006) (cit. on pp. 5, 9).

[Reguera and Leiva 2017] J. L. Reguera and Y. F. Leiva. “A learning methodology for object oriented programming with effective support from the pa3p automatic evaluation platform”. In: *2017 36th International Conference of the Chilean Computer Science Society (SCCC).* Oct. 2017, pp. 1–8. doi: 10.1109/SCCC.2017.8405111 (cit. on pp. 2, 26).

[Resnick *et al.* 2009] Mitchel Resnick *et al.* “Scratch: programming for all”. *Communications of the ACM* 52.11 (Nov. 2009), pp. 60–67. issn: 0001-0782. doi: 10.1145/1592761.1592779. url: http://doi.acm.org/10.1145/1592761.1592779 (cit. on pp. 1, 15).

[Sáez-López *et al.* 2016] José-Manuel Sáez-López, Marcos Román-González, and Esteban Vázquez-Cano. “Visual programming languages integrated across the curriculum in elementary school: a two year case study using “scratch” in five schools”. *Computers & Education* 97 (2016), pp. 129–141. issn: 0360-1315. doi: https://doi.org/10.1016/j.compedu.2016.03.003. url: http://www.sciencedirect.com/science/article/pii/S0360131516300549 (cit. on pp. 1, 19, 63, 67).

[Savi *et al.* 2011] Rafael Savi, Christiane Gresse von Wangenheim, and Adriano Borgatto. “A model for the evaluation of educational games for teaching software engineering”. In: Sept. 2011, pp. 194–203. doi: 10.1109/SBES.2011.27 (cit. on pp. 6, 56, 57).

[Shu 1986] Nan C. Shu. “Visual programming languages: a perspective and a dimensional analysis”. In: *Visual Languages.* Springer, 1986, pp. 11–34. isbn: 9781461318057 (cit. on pp. 7, 12).

[D. M. D. Souza *et al.* 2015] Draylson Micael De Souza, Seiji Isotani, and Ellen Francine Barbosa. “Teaching novice programmers using progtest”. *International Journal of Knowledge and Learning* 10.1 (2015), pp. 60–77. doi: 10.1504/IJKL.2015.071054. eprint: https://www.inderscienceonline.com/doi/pdf/10.1504/IJKL.2015.071054. url: https://www.inderscienceonline.com/doi/abs/10.1504/IJKL.2015.071054 (cit. on p. 27).

[L. d. Souza *et al.* 2021] Lucas de Souza, Igor Felix, Bernardo Ferreira, Anarosa Brandão, and Leônidas Brandão. "I know what you coded last summer". In: *Anais do XXXII Simpósio Brasileiro de Informática na Educação*. Online: SBC, 2021, pp. 909–920. doi: 10.5753/sbie.2021.218673. url: https://sol.sbc.org.br/index.php/sbie/article/view/18117 (cit. on p. 59).

[Sykes 2007] Edward R. Sykes. "Determining the effectiveness of the 3d alice programming environment at the computer science i level". *Journal of Educational Computing Research* 36.2 (2007), pp. 223–244. doi: 10.2190/J175-Q735-1345-270M. eprint: https://doi.org/10.2190/J175-Q735-1345-270M. url: https://doi.org/10.2190/J175-Q735-1345-270M (cit. on pp. 18, 62–64).

[Tan *et al.* 2009] P. Tan, C. Ting, and S. Ling. "Learning difficulties in programming courses: undergraduates' perspective and perception". In: *2009 International Conference on Computer Technology and Development*. Vol. 1. Nov. 2009, pp. 42–46. doi: 10.1109/ICCTD.2009.188 (cit. on pp. 8, 9).

[Taylor 1982] Robert P Taylor. *Programming primer : a graphic introduction to computer programming with BASIC and Pascal*. English. Includes index. Reading, MA : Addison-Wesley, 1982. isbn: 0201074001 (cit. on p. 7).

[Tennent 1976] R. D. Tennent. "The denotational semantics of programming languages". *Commun. ACM* 19.8 (Aug. 1976), pp. 437–453. issn: 0001-0782. doi: 10.1145/360303.360308. url: http://doi.acm.org/10.1145/360303.360308 (cit. on pp. 7, 29).

[Vainio and Sajaniemi 2007] Vesa Vainio and Jorma Sajaniemi. "Factors in novice programmers' poor tracing skills". In: *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '07. Dundee, Scotland: ACM, 2007, pp. 236–240. isbn: 978-1-59593-610-3. doi: 10.1145/1268784.1268853. url: http://doi.acm.org/10.1145/1268784.1268853 (cit. on pp. 8, 9, 64).

[Vee 2013] Annette Vee. "Understanding computer programming as a literacy". *Literacy in Composition Studies* 1.2 (2013). issn: 2326-5620. url: http://licsjournal.org/OJS/index.php/LiCS/article/view/24 (cit. on pp. 1, 8, 10).

[Watson and F. W. Li 2014] Christopher Watson and Frederick W.B. Li. "C". In: *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. ITiCSE '14. Uppsala, Sweden: ACM, 2014, pp. 39–44. isbn: 978-1-4503-2833-3. doi: 10.1145/2591708.2591749. url: http://doi.acm.org/10.1145/2591708.2591749 (cit. on pp. 5, 9).

[Whitley 1997] K. N. Whitley. "Visual programming languages and the empirical evidence for and against". *Journal of Visual Languages & Computing* 8.1 (1997), pp. 109–142. issn: 1045-926X. doi: https://doi.org/10.1006/jvlc.1996.0030. url: http://www.sciencedirect.com/science/article/pii/S1045926X96900300 (cit. on p. 18).

[Wilcox 2015] Chris Wilcox. “The role of automation in undergraduate computer science education”. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE ’15. Kansas City, Missouri, USA: ACM, 2015, pp. 90–95. isbn: 978-1-4503-2966-8. doi: 10.1145/2676723.2677226. url: http://doi.acm.org/10.1145/2676723.2677226 (cit. on p. 28).

[Wing 2006] Jeannette M. Wing. “Computational thinking”. *Commun. ACM* 49.3 (Mar. 2006), pp. 33–35. issn: 0001-0782. doi: 10.1145/1118178.1118215. url: http://doi.acm.org/10.1145/1118178.1118215 (cit. on pp. 1, 10).

[Yu *et al.* 2017] Y. T. Yu, C. M. Tang, and C. K. Poon. “Enhancing an automated system for assessment of student programs using the token pattern approach”. In: *2017 IEEE 6th International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. Dec. 2017, pp. 406–413. doi: 10.1109/TALE.2017.8252370 (cit. on pp. 5, 25, 27, 37).