From Educational Programming to Professional Programming

Group DPT906E15 - Room X.X.XX

4 February - 1 June

Date	Jais Morten Brohus Christiansen
Date	Henrik Vinther Geertsen
Date	Svetomir Kurtev
Date	Tommy Aagaard Christensen

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Computer Science
Selma Lagerlöfs Vej 300
Telephone 99 40 99 40
Telefax 99 40 97 98
http://cs.aau.dk

Title:
Whist on Mobile Devices

Project period:
4 February - 1 June

Project group:
DPT906E15

Participants:
Jais Morten Brohus Christiansen
Henrik Vinther Geertsen
Svetomir Kurtev
Tommy Aagaard Christensen

Supervisor:
Bent Thomsen

Abstract:

Pages: 48

Appendices: 0

Copies: 2

Finished: 1 June 2015

Contents

Chapter 1

Introduction

Traditionally, programming has been seen as a specialized skill, only relevant for people who work on developing new software. While several attempts are made at making languages more accessible to everyone TODO: reference logo and probably some more, they ultimately failed to get enough widespread use to change this perception. As a result, programming education has been targeted towards college or university students and often had a focus on teaching the student to use professional programming languages like JavaTODO: can probably use source about the widespread use of Java here. The goal here is to create competent software engineers with the ability to work with powerful tools on complicated software.

In recent years, programming is starting to be seen as an essential skill for living in our modern digital society. This has lead to many countries making programming a mandatory subject in primary schools. In this setting, the education is more focused on giving children an idea of what it is like to work on software and to teach more generic skills, such as problem solving and collaborative communication. This leads to this education usually being given in visual educational languages like Scratch [1] for their intuitiveness and simplicity. This means that different levels of education differ in what they teach. First, there is a difference in programming languages, as to between being taught an educational language and a professional language. While the educational language has the advantage of being intuitive, it does not have the expressiveness and robustness for large projects that professional languages have. Second, the computer science education needs to cover a lot of topics to give a sufficient understanding to work professionally with software, where the kids education leaves out a lot of the topics, although this may vary between teachers. Using the list of topics from "What do Teachers Teach in Introductory Programming?" [2] as a reference, we can for example say that topics like algorithm design and debugging are likely to be taught in the kids education. Meanwhile, topics like algorithm efficiency, pointers and object oriented programming are usually left out.

Kids now being taught programming in primary school is great for the general digital literacy of people, but the education can not teach everything necessary to do professional programming. We want to understand the educational value of teaching novice programmers programming in primary school. To do this, we first need to understand how and what novices are taught. The paper therefore aims to understand the areas where novices in primary school have problems as well as the teaching methods used in the education.

TODO: Add initial questions.

The first thing we will address is an analysis of the area as a whole. In Chapter 2, we discuss the various problems that novices are faced with when learning to program. We will then continue with an analysis of different educational languages in Chapter 3. After that, we provide an analysis on the teaching as it is today in Chapter 4.

After an analysis of the area, we will give a language comparison, done as an expert evaluation. The introduction to why and how this is done will be provided in Chapter 5. We will state the criteria evaluated in the comparison in Chapter 6. The comparative analysis itself will be provided in Chapter 7. TODO: Not done yet.

Part I

Problem Analysis

Chapter 2

Error-Prone Areas for Novices

For a person new to programming, different constructs and concepts can be so confusing that this person might give up without much effort. This area is of great interest to many studies, as it can help future generations in learning programming with ease. But to find solutions, the difficulties that can arise when learning programming and the concepts that follow must be known.

This chapter focuses on the different aspects of learning programming that can be difficult to grasp for novices. The different aspects and concepts are found by previous studies as well as subjective speculation.

2.1 Syntax and Semantics

As known, a programming language is based on the syntactical rules and the semantic relations. These concepts can be hard to grasp at first, and can be even harder to understand in relation and when used in a practical solution.

One of the most error prone areas for novice programmers is the basic syntax [3]. This consists of brackets, semicolons, commas, and other such symbols, symbolizing control for the program. This problem might relate to an even greater problem in understanding the strict control that is needed when writing code in general. When writing code, even the smallest mistake or forgotten symbol leads to a compiler error. This error margin isn't seen very often in other lines of work, and might discourage the novices from keep trying.

Understanding what a line of code does in itself might be hard for some new programmers. Understanding the connection of the whole program, and what the single line does for the result is even harder. The semantics can lead to confusion, as the program grows bigger. Some times, the novice programmer is even discouraged from even trying, as the connection between the code and what it results in is not clear.

2.2 Pragmas

TODO: I still need to figure out why it's called pragmas, or if it's just a made-up word. For now, I will leave it out of anything but the title. Also, I haven't been able to find anything on this yet.

A programming language is built on syntax and semantics. To learn how this works can be effective in programming, but programmers often don't think in these terms. Experienced programmers know their way around the basic programming principles and constructs, which is more or less the same in all languages. Programmers often think in patterns, some standardized way for them to program. The logic composition of the elements at hand is often the key for most, working on the idea, in stead of the specific language's behaviour. Even though all programmers have a pattern of programming, good or bad, there is a base line for standard programming patterns. They vary slightly from paradigm to paradigm, but at some level, there is a common thought on the structuring of code. This common coding practice is hard to find and to measure, which leads to no TODO: very little? work around this area. Although, it should be taken into account that this way of finding code patterns might be much more useful than teaching syntax and semantics.

2.3 Programming Paradigms

Different paradigms each have their different difficulties. Many programmers first touch programming through an imperative or procedural approach. Others start out with an object oriented programming language.

Procedural programming has its values in its very straight forward and easily trackable nature. On the other hand, it is hard to see the connection to real world problem solutions, as the very strict text-based structure doesn't resemble these much. Nevertheless, certain tools are used today for teaching, such as Scratch (have we described these yet?), which makes procedural programming a valid learning approach.

Object oriented programming (OOP) has it's values in representing real world problems, and how a solution can be modelled. As OOP is mostly based on classes, being the static description of an object, and objects, being the dynamic model of a real world phenomenon, the concepts of the paradigm can be easily grasped. Of course, this fact demands a teaching method suitable for the novice programmers being taught. On the other hand, OOP is often in relation to procedural programming seen as not being something else, but the same, only with object oriented features [3]. This leads to a problem of both understanding the very basic concepts of programming, such as control structures (loops and selections), along with understanding the object oriented approach.

Functional programming (FP) is a paradigm which uses functions, instead of procedures or objects, as the building blocks of a program. Having its roots in lambda calculus, computations in FP are treated as the evaluation of functions where change of state is avoided and the data is immutable which, in turn, prevents the introduction of side effects in programs [4]. Additionally, many concepts of the imperative approach can be simulated in Functional programming(ex. control structures are expressed in terms of recursion), giving as much of a control or power when building programs. However, the notion of using functions as a natural abstraction instead of objects modelled after the world introduces difficulties for novice programmers in understanding the fundamental concepts of Functional programming. Therefore the paradigm is rarely selected as a choice of teaching programming in introductory courses [5].

It is a wide discussion to determine what approach is the most efficient teaching method. The necessity of learning concepts of OOP before learning to code can be argued, as well as it can be argued that the basic constructs are necessary before learning about different paradigms and advanced structures. The procedural approach is being taught in elementary school in various countries (TODO: assumption, need refs). In OOP, the question is often what teaching methods are used to make students understand the concepts of the paradigm. Studies have shown a better effect when teaching about the concepts before actually coding [6]. Another approach to pursue could be the “object-first approach”. As the name implies, this approach searches to understand the logic behind objects before anything else. This method can both imply teaching concepts or coding before anything else, and can both be used by novices or by students that have already learned the basics of programming. Although, there is still discussion on whether novices should be taught through this approach, or the classic “algorithms-first approach” [**?**] TODO: fix cite. One drawback is the fact that the concepts of OOP might be seen by the novices as the basics for programming itself, instead of the basic concepts, which could lead to a block when exploring other paradigms.

Some modern languages, such as C# and Java, have become what one might call a “multi-paradigm language”. In these examples, they started out being OOP languages, but now they have implemented various features of other paradigms, such as functional and logic programming. These languages could be considered when learning to program, as the drawback from changing to other languages could be limited.

Chapter 3

Programming Languages and Tools

With the growing distribution of computers and mobile devices, e.g. as laptops, smart phones and tablets, the need of programming languages and tools to operate on these machines becomes more and more relevant. Using such tools and languages efficiently, however, often requires much skill and technical aptitude, which in turn takes considerable time and dedication to develop. From the perspective of novice programmers, programming can be extremely hard and overwhelming to get into, especially if they are given no introductory tools and guidance.

This chapter focuses on the distinction between visual and text based programming languages, analysis of both categories as well as some notable examples. Additionally, these categories are further explored by means of how well they fit in an educational setting.

3.1 Text Based Programming Languages

TODO: Watch the Smalltalk talk, and maybe add something about it here

Text-based programming languages have two distinct subcategories; one is the text-based educational programming languages for novices, specifically designed to teach novices for then to be replaced by a more feature rich language, and the other is the general purpose programming languages. Even though general purpose programming languages can be used to teach programming to novices, the two categories are distinguished in where the focus of the design has been. This chapter will explore a set of programming languages in both of these categories.

3.1.1 Text Based Educational Programming Languages

This section will provide a description of some different text based educational programming languages and constructs.

Turtle Programming

One of the first programming languages that added constructs for learning programming was LOGO. It did not have the constructs from the start, but after 12 seventh-grade students[1] worked with LOGO for a year (1968-1969), Seymour Papert, one of the developers of LOGO, proposed the Turtle as a programming domain that could be interesting to people at all ages. He proposed it since the demonstration had confirmed that LOGO was a relatively easy programming language for novices to learn, but he wanted the demonstration extended to lower grades, ultimately preschool children. Constructs for Turtles was then added to LOGO and has since been widely adopted in other programming languages such as SmallTalk and Pascal, and more recently Scratch [7].

A Turtle can be a visual element on a screen or a physical robot. In Scratch, the Turtle can be any sprite chosen by the user. In LOGO the Turtle is controlled by a set of commands which are:

- FOWARD X, moves the Turtle X number of Turtle steps in a straight line
- RIGHT X, turns the Turtle X number of degrees in a clockwise direction
- LEFT X, turns the Turtle X number of degrees in a counter clockwise direction
- PENDOWN, makes the Turtle draw
- PENUP, makes the Turtle stop drawing

These commands make up the essence of Turtle programming and the functionality is also present in the other languages which has implemented Turtle programming, maybe using different keywords. Some languages has expanded on these commands, e.g. in Scratch, one can change the color, size and shade of the pen. The commands can be part of user defined functions. Examples could be functions called *SQUARE* or *TRIANGLE* which would draw a square and a triangle respectively using the commands shown. These functions can be part of other functions, e.g. a function called *HOUSE* would use a mix of the commands for correct positioning and then the *SQUARE* and *TRIANGLE* functions to draw the house itself [8].

Turtle programming is not only meant as a tool for learning to program. Seymour Papert states that it is meant as an *Object-to-think-with*. This means that it is supposed to give children a way of relating new topics to something they already know. Alan Kay shows an example of this in a talk, where he uses a car sprite (the turtle in this case) to visualize acceleration. He does this by programming a loop for the car that for each iteration makes a circle showing where the car have been and then moves the car forwards to a new position. In each iteration he also increases the distance that the car moves forwards, meaning that the distance between the circles becomes greater and greater, thus visualizing the car accelerating [9].

[1] From Muzzy Junior High School in Lexington, Massachusetts.

Small Basic

Small Basic is a text based programming language with its whole purpose being teaching novices to program. It is not meant as a language one should keep using, but as a tool for learning programming principles and then “graduating” to learn more advanced languages. In the developers internal trials they have had success with teaching programming to kids in the ages of 10 and 16, but it is intended for novices in general, so it is not specifically made for that age group [10]. It is perhaps one of the last languages with this purpose that is still being updated[2]. It seems as if the focus of teaching programming to novices has changed to visual programming languages, tools for learning existing languages that is meant to be used professionally or languages that are novice friendly, but still meant to be used professionally.

The Small Basic project consist of three pieces: The language itself, an IDE and libraries. This means that one has to use the bundled IDE to program in Small Basic. According to their FAQ [10], the language takes inspiration from an early variant of BASIC but is based on the modern .NET Framework Platform. It is much smaller than Visual Basic, it consists of 14 keywords, and supports a subset of the functionality that Visual Basic .NET supports. It does not have a type system and all variables are global and always initialized as to avoid confusion regarding scopes. It is also imperative and does not use or expose beginners to the concept of object orientation.

To learn programming with Small Basic, its website provides a tutorial for getting familiar with the language and the IDE [11], and a curriculum. The curriculum can be downloaded for offline use, and teaches general programming topics using Small Basic [12]. Through the introduction one will learn that there are two different window types a program can be run in, either the *TextWindow* which is a regular console, or in the *GraphicsWindow* where graphics can be drawn and so on. So it is possible to create e.g. games in Small Basic. Turtle programming is also supported.

The Small Basic IDE aims to help novices as well. The IDE is made up of different parts, which can be seen in Figure 3.1. The first part is the menu bar in the top. It contains a very limited subset of functionality that we are used to see in regular programming IDEs and a Small Basic specific functionality, the Graduate button. The Graduate button can export the code that has been written in Small Basic to Visual Basic so that the developer can keep programming on their project even if they have “graduated” to using Visual Basic instead. The second part is the editor itself, where the code is written. The third part is intelliSense and auto complete. It is works the same as in Visual Studio, it shows what is possible to write with a description of what that functionality does. The fourth part is where the error messages are shown and the fifth part shows the properties of a selected element.

[2] With that being said there was a four year hiatus between version 1.0 and 1.1

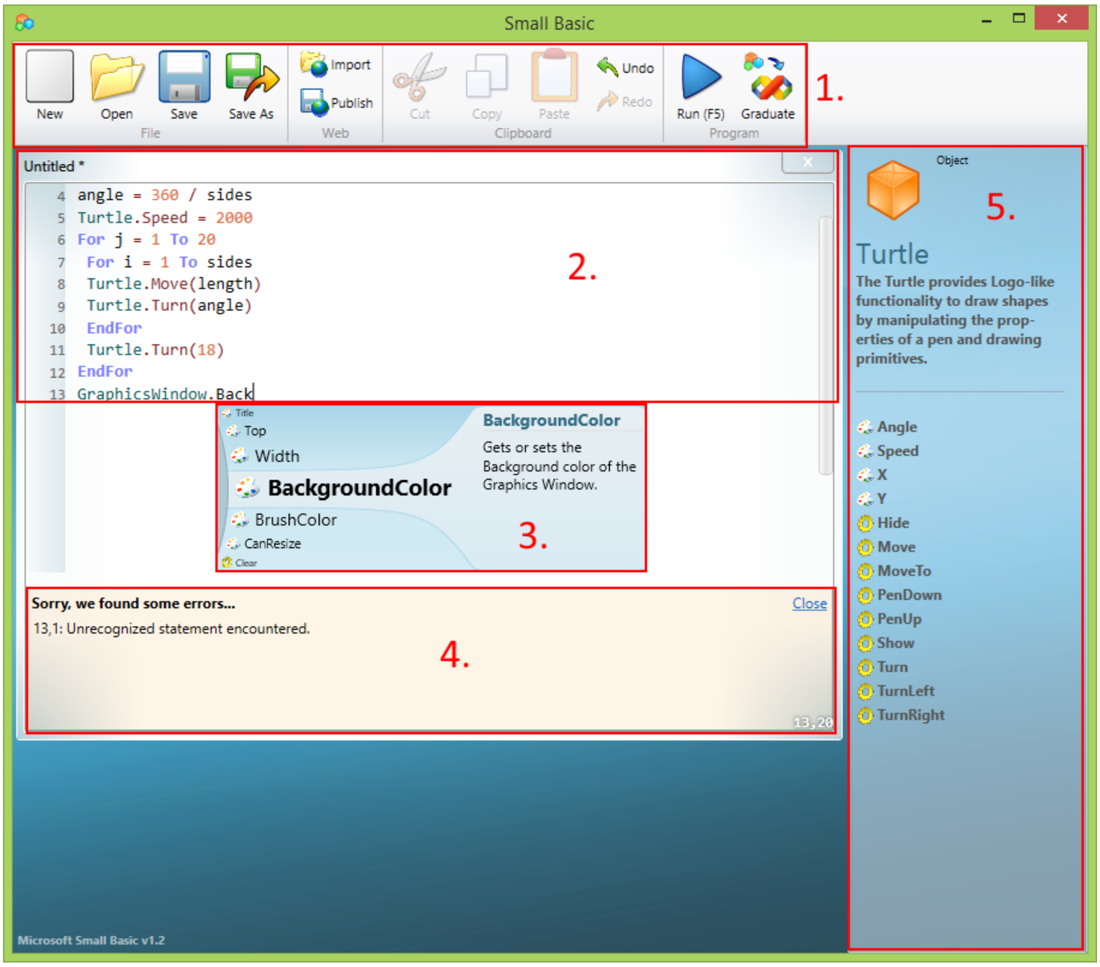


Figure 3.1: Small Basic IDE

3.1.2 General Purpose Programming Languages

This section will provide an overview of some of the general purpose programming languages that has been designed with novices in mind, but still aims to be used as a end-user language. It will also provide an overview of some of the tools used to teach novices to program in languages not specifically designed for novices but as a professional programming language. TODO: We need to discuss terminology further.

Programming Languages

One of the first programming languages designed for novices and to be user friendly is BASIC (Beginner's All-purpose Symbolic Instruction Code). It first appeared in 1964 as a language that would enable students in fields other than science and mathematics to use computers. Since then it became a widely used language and today there are more than 230 different documented dialects of BASIC, among those is Microsoft's Visual Basic.

Another language that was originally designed largely with students in mind was Pascal. It was released in 1970 and aimed to teach students structured programming. However, some early adopters used it far beyond the original intent. This resulted in a lot of work on Pascal and it evolved, both the language itself but also into different dialects, which ended up being designed more towards end-user use rather than an educational tool. Although this meant that the language and dialects became very popular and widely used. The language itself and the dialect called Delphi/Object Pascal is still used today [13].

A third language that was designed and is still maintained as an easy-to-use general purpose programming language is Quorum. Quorum is an evidence-based programming language. This means that it is updated and changed according to current research on how an easy-to-use programming language should be designed. Some of the team members behind the language host an annual workshop called The Experience Programming in Quorum (EPIQ), which is "an international professional development workshop for educators to learn the foundational skills necessary to teach students computer science using the Quorum programming language" [14].

3.1.3 BlueJ

BlueJ is a development environment will allows the creation of Java programs quickly and easily. BlueJ has its roots back in the nineties when Michael Kölling developed a pedagogical language and environment called Blue [15]. Essentially, BlueJ was initially released as a port of Blue to Java in 1999 and its support continues to this day, thanks to Sun Microsystems and Oracle.

First and foremost, the focus of BlueJ is that its primarily designed for teaching, with good pedagogy in mind. Therefore, many of its characteristics are centered around that notion such as simplicity,interactivity, maturity and innovation. BlueJ has a smaller and simpler interface compared to professional programming environments such as NetBeans or Eclipse with the deliberate intention of not to overwhelm beginners. Additionally, it allows a great deal of interaction with objects e.g. inspecting their values, calling methods on them, invocation of Java expressions without compilation etc. Given that BlueJ is fifteen years old with a solid foundation and full-time team working on it, beginners in programming can easily make use of its technical support. Being a well established environment, BlueJ also has several original features not present in other IDEs such as object bench code, code pad and scope colouring.

In order to address better the pedagogical side of BlueJ, the BlueJ development team is constantly looking for ways to improve the learning process of programming and make it easier,simpler and more enjoyable. However, this is often not an easy thing to do since there is no real way to measure how good a design decision is [16]. Having a way to obtain data from its usage will give the team more control over the environment and will benefit the community of BlueJ users as well. Even more, the collected data could be of interest to the wider researcher community and generally people who might be interested in how BlueJ is being used, to much greater benefit. This idea of expanding the amount and type of data collected, while keeping it anonymous, gave fruition to the Blackbox project.

The Blackbox data collection project was announced at the 2012 SIGSE conference in Raleigh,North

Carolina,USA. [16] Initially, with the feedback form attendees, the data collection method was finalised along with some technical details for the whole process. As already mentioned, one of the key features of the project is keeping the data anonymous in order to avoid any ethical and legal complications. The data collection continues to date, as the only condition is to have BlueJ 3.1.0 or newer in order to participate in this research [17].

3.1.4 Dr. Racket

Dr. Racket is and IDE for the language Racket, which is a functional programming language and has its roots in Lisp and Scheme but with added features. The IDE is created for novices that are learning the Racket language, and programming in general, in combination with the book "How to Design Programs"[3]. There is also some "getting started" tutorials on their website and the language is fully documented, which can also be found on their website[4].

Even though the IDE is for novices, it still has features similar to a fully fledged IDE and the assumption for having those are so that the novices can stay with the IDE even when they have finished learning the basic elements. Although most of these features are hidden away in the menu and a novice is only required to start the IDE to get started with the programming. Some of the features for novices can be seen in Figure 3.2, where one can see the editor window (upper part of the window) and the interactive window (lower part of the window). In the editor a bunch of code can be written and then run using the "Run" button in the upper right corner. Code can also be written in the interactive window, where it will be executed immediately after the completion of an expression. After the execution the value that the expression has been evaluated to is written as well, so that a novice always can follow what they are doing. The two windows work together, so if a user has written a definition in the editor window and presses run, that definition is available in the interactive window.

Another feature for novices can be seen in Figure 3.3, in which the "Choose Language" window is shown. In this window the user can select which language should be support by the IDE, e.g. subsets of the Racket language. This feature is there so that a novice can choose e.g. a subset of racket which removes access to advanced features so that they can focus on what is relevant. An example of this is if the "Beginning Student" is chosen, procedures must take at least one argument. This limitation is there because procedures in the languages have no side-effects meaning that procedures without arguments are not useful, so it can help avoiding confusing syntactic mistakes [18][5]. So in a sense it makes sure that the novice follows the programming guide lines for the language and does not end up with a work around more appropriate for an advanced user.

[3]It is a free book and can be found here: http://www.htdp.org/

[4]http://docs.racket-lang.org/

[5]A list of the features modified by choosing the different languages can be found here: [18]

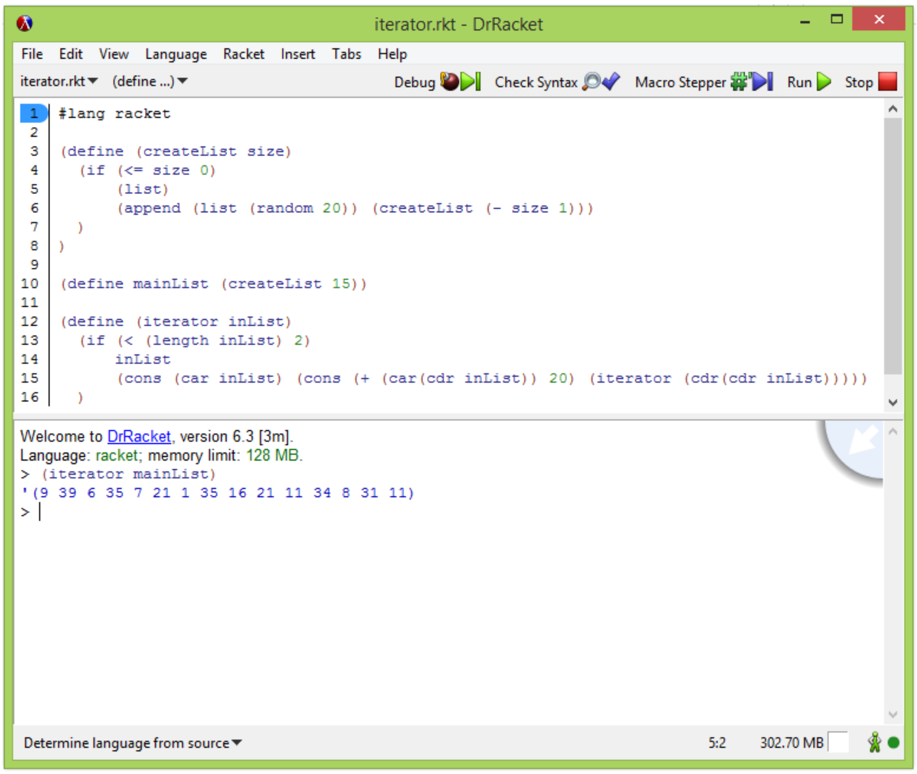


Figure 3.2: Dr. Racket IDE

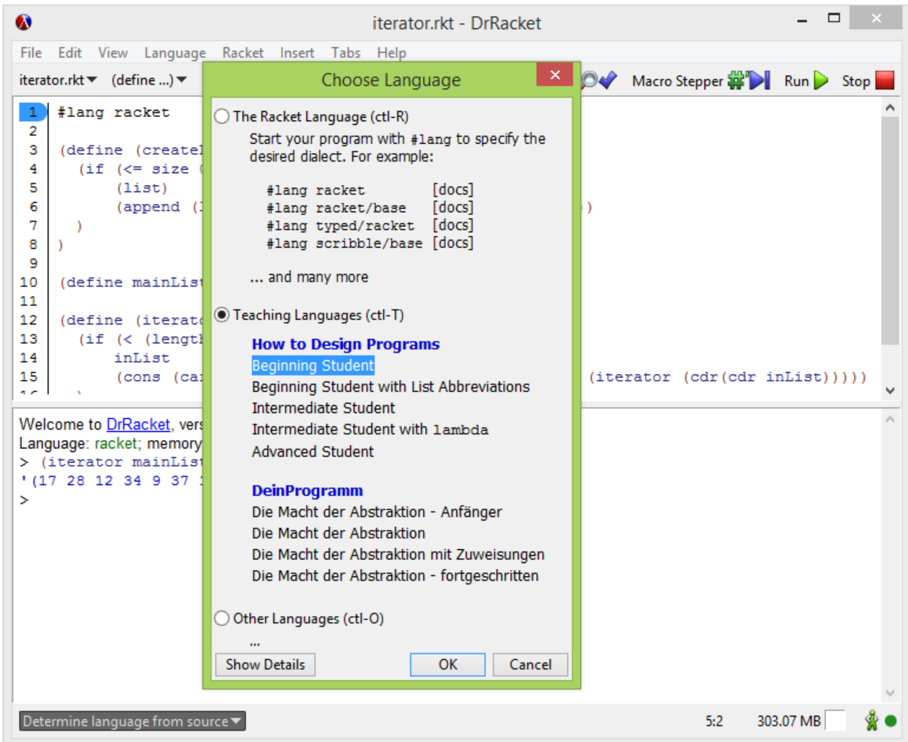


Figure 3.3: Dr. Racket, Choose Language Window

3.2 Visual-based Programming Languages

Traditionally, most programming languages are categorized as text-based because of the way the program logic is written, by making use of a syntax, specific to every language. Therefore, it is often difficult to learn and use a programming language since it requires one to familiarize oneself with the syntax and available constructs first in order to use the language effectively and that takes skill many people lack.

In order to address the difficulties in learning programming, for the past 25 years TODO: JAIS: Isn't it longer than that? Either way, a cite is needed, research has been done on the so called "Visual Programming" or "Graphical Programming", and dozens of visual-based programming languages have been created. This approach, reserved and used in the past primarily for systems design, allows the use of spatial representations in two or more dimensions in the form of blocks and different structures and shapes. Compared to text-based programming where lines of code are used, graphical programming replaces these with visual objects, essentially replacing the textual representation of language components with a graphical one, more suitable for visual learners and intuitive for people with no prior knowledge in programming. The creation of programs in such languages is defined by placement and connection between visual objects where the syntax is encoded within the objects' shapes. The main aim of visual programming languages (VPL) and environments, as stated by Koitz ans Slany [19], is "*diminishing the syntactical burden and enabling a focus on the semantic aspects of coding*." VPL try to facilitate end-user programming, both kids and adult novice programming, empowering the creation of new programs, not just their consumption, effectively minimising the distance between the cognitive and computational model.

Currently, there is a wide variety of visual programming languages with varying popularity such as Alice, Greenfoot, Tynker, Scratch, Raptor and many more. From these, Scratch and Alice will be described and analyzed further.

3.2.1 Scratch

Scratch is a visual-based programming environment which allows users to create visually-rich, interactive projects. Since its inception in 2003, the main goal of its creators has been to address the needs and interests of young people (primarily ages 8 to 16) and make a soft introduction to the world of programming for them. Publicly released in 2007, the project has grown in size and scope, with a dedicated site hosting all its 11 million projects and with a user base of 8 million [20]. Given its targeted audience, one of the main design goals of Scratch is the focus on self-directed learning and exploration through tinkering with the different constructs of the language and environment. This combined with the steady increase of its popularity has prompted hundreds of schools and educational organizations to adopt and integrate it into their curriculum [1].

What makes Scratch a sensible choice for people with no prior programming experience is the fact that it has less emphasis on direct instruction than other programming languages. Instead, it focuses on the aspect of learning through self exploration and peer sharing, which breaks the norm of a traditional educational approach.

3.3 Comparison

As shown in this chapter both text and visual based programming languages has parts of their domain dedicated to learning programming. This is done mostly through tools, be it environments for general purpose programming languages or visual programming languages where the environment can be seen as a part of the language. Both approaches has pros and cons, and these will be explored in this section.

Interface Layout

One of the apparent problems with visual programming environments is that a lot of them has multiple elements, e.g. code, available blocks, etc., where each of them requires a slice of the available screen space. This means that a lot more thought should be put into the interface design, where as environments for text-based programming languages can put less focus on it.

Statement Categories

One of the pros of visual programming environments is that they often require a way of dragging e.g. blocks to create functionality. This means that all of the available blocks, or commands in a text based programming language, is always shown and available to the user (it might be split into categories), which gives the user an overview of the possibilities in the language. Text based programming languages lacks in this aspect, as the user often is presented with an empty editor or some tutorial code, which still does not tell the user much about the possibilities. Small Basic is an example where the possibilities are shown through its auto completion, but the user is just presented with a long list and not all of the commands are descriptive in their naming, possibly making it hard to get an overview for a novice.

Writing Speed

One of the pros of text based programming languages is that it is possible to produce code faster compared to visual programming environments. The reason for this is that visual programming environments often are dependent on moving code blocks using the mouse, and if an user wants to make changes to e.g. a function made up of multiple blocks, they have to pull it apart to get to the block they wish to exchange. In text based environments the user has the possibility of using both the keyboard and the mouse when they want to select something, giving the possibility of using the preferred peripheral of the user.

TODO: HENRIK: What do you guys think? TODO: HENRIK: How do i finish this section, what is the goal of it?

Chapter 4

Teachers and Programming

Around the world, learning computational thinking is starting to appear in the school curriculum for students in the age range of 5-16 years old, depending on the country. Because of this it is relevant to study which resources are available to the teachers who are going to teach the children. In this chapter we will describe the current state of the teachers and discuss possible problems regarding that state. The chapter will focus mainly on Denmark and the United Kingdom.

4.1 Current State

As of 2015 the danish government has added programming to the school curriculum, although this might be a wrong way of phrasing it as, according to the Danish Learning Portal[1] it is not only programming that should be taught but computational thinking in general. An example of this can be seen in this quote on how to incorporate "programming" into the mathematics course[2]:

> Programming activities can support that the students work with algorithms, meant as systematic descriptions of issues, solution strategies and events. A recipe is a good example of an algorithm. (1) Mix the dry ingredients together, (2) stir. (3) Add 2/3 of the water and stir. (4) If the dough is smooth, stir for 2 minutes. Else go to step (3) and add more water. Algorithmic thinking is about setting up and making machines execute such algorithms... [**?**]

The problem with this addition to the curriculum is that the danish government expect the individual teacher to teach themselves the subject, to then teach the students. The government has allocated $1 * 10^9$ Danish Crowns to educate current teachers in the new subjects added to the curriculum for 2015, but programming is only one subject among many. This means that it is uncertain how much of it goes to educating teachers in the ability to teach programming. One of the things the government has suggested is the programming environments which can be used, here among "Scratch", "Tynker" and websites like

[1] http://www.emu.dk

[2] This is translated from danish, so the wording might be different in the original text, but the meaning is the same

Code.org.

Another part that is uncertain is how the educations regarding teaching will implement the new curriculum. It is the individual educational institution's job to make certain that the teachers who graduate are equipped to the goals of the curriculum [**?**].

It is a different story regarding the U.K. They implemented programming and computational thinking in the curriculum in 2013, which took effect September 2014 [**?**]. To prepare the teachers for this they allocated $1.1*10^6$ British Pounds in funding specifically to train school teachers who are new to teaching computing [**?**]. This was announced in December 2013 and in February 2014 another $500,000$ British Pounds was allocated in funding to attract businesses to help train teachers [**?**].

4.2 Discussion

The United Kingdoms seem to have a good grasp on how to educate teachers with no knowledge about programming. Denmark on the other hand seems to struggle which can possibly lead to problems in the future. To start with, if the teachers are not experienced enough or not enthusiastic about the subject, as it, in the case of Denmark, is the physics/chemistry teacher who mainly has to teach programming, can possibly end up demotivating students from programming. Luckily governments suggests languages such as Scratch to use in lessons, which has a good range of online tutorials and self learning material.

Another possible problem is that if the students are taught in a way that gives them a wrong understanding of aspects in programming or makes them develop bad habits, they can struggle with those problems later in their education, given that they choose an education that is programming related[3]. Given these possible problems, it might be interesting to make research in this area to determine if these end up being actual problems.

[3] These problems are speculative and anecdotal and have no roots in the literature.

Part II

Language Comparison

Chapter 5

Preliminaries

To get a better idea of the landscape of educational programming languages, we are going to compare three popular educational languages. The three languages are: Scratch, BlueJ and Dr. Racket. These three languages are chosen as the most popular languages of each their paradigms TODO: probably need to find a source for such a claim. Scratch represents an imperative paradigm, BlueJ an object-oriented paradigm and Dr. Racket a functional paradigm. To do this we will first define some criteria to compare the languages on, after which we will subjectively analyse the languages to get a comparison.

Chapter 6

Criteria

A set of criteria have been set up in order to measure the differences between languages with focus on novice learning. These criteria are the base from the evaluation, and are a mixture of measurable and subjective criteria.

This chapter will present the criteria for the evaluation, the reason for the criteria being chosen, and how to measure them. Several criteria have been considered, where most are taken from known criteria for comparison [21] [22], and others are found through discussion of the research questions.

Readability
: The language expresses itself through syntax as readability for the programmer. Readable code gives a greater understanding of the semantics as well of the nature of the code. For this project, readability is a vital criteria, as interest often follows understanding. Readability is hard to measure, as it is a subjective matter, mostly depending on the person writing code. The measurement of this quality will therefore be based on using code conventions and skilful coding.

Writability
: Writability is the ability to translate thoughts into code. It describes the expressivity of the code and the ease of writing, in a combination of quality and quantity. Writability can be measured in the level of abstraction. This can be done through lines of code as well as looking into the different language constructs that support abstraction.

Observability
: Observability is the level of feedback gained for a better understanding of the input. It is to what extend you can observe reactions to what you make. To measure the observability is lo look at this level of feedback.

Trialability
: The level of possibility of trial and error through coding is measured through trialability. This feature is measured by the level of feedback when an error occurs, how often feedback is given, and a discussion on how easily a novice programmer can recover from a mistake.

Learnability

As a language is learned, there are helping and hindering factors. These are measured through learnability, which is done by measuring the cost of learning the language and its environment.

Reusability

The level of possibility for reusing code, through abstraction. This is measured both in quantity and quality, depending on the abstraction level and layer depth.

Pedagogic Value

A programming language in itself can be easy to learn, but if it doesn't help the programmer in learning the basic concepts of programming, then there is no pedagogic value. This means the programming language should support the general programming concepts that are typical for common languages and coding conventions.

Environment

The development environment plays an important part for novices, and it should provide help for the programmer in a simple, clear and manageable way. The evaluation of the environment will be done according to how interactive systems are evaluated by David Benyon [23, p. 225-250].

Documentation

The amount of documentation, as well as the informative value of this, is important for a novice, as a help for solving problems they cannot solve themselves.

Uniformity

The consistency of appearance and behavior of language constructs. If the code doesn't look like any conventional language, the programmer will not learn the general approach to programming, and will have trouble moving on from this language. This is measured in terms of constructs being similar to known syntaxes, or preferably written in the exact same way.

Miscellaneous

There is a possibility that other points of interest will be discovered during the comparison. These will be described in this category.

TODO: A conclusion on how to measure criteria.

6.1 Criteria Evaluation

As the criteria are hard to measure in an objective way, the evaluation will be through a subjective discussion. Each of the environments are meant for different purposes, as they are based on different paradigms. Taking this into account, we will create solutions to problems that are fit for each of the environments, and then compare these solutions in how all the environments handle and solve this problem.

Chapter 7

Language Analysis

In this chapter a subjective comparative analysis is made of the three programming environments/languages; Scratch, BlueJ and Dr. Racket as stated in Chapter 5. The procedure used for the analysis is as follows: First, some tasks have been made which are implemented in the different environments/languages. Then each language is analyzed individually using the implementations of the tasks, where the analysis is based on the criteria presented in Chapter 6. Finally, the analyses are used to compare the environments/languages in Chapter 9. TODO: Depends on where we make the final comparison. The goal of this analysis is to get an insight into what each paradigm has to offer regarding educational programming and their environment and how they compare. TODO: We all need to agree that this is the case.

The tasks are made to get as good an overview of the languages as possible. They will sometimes hypothetically favor a certain paradigm over the other, but this is done on purpose to try and find possible problems with the languages. The tasks that are implemented in each of the languages are:

Task 1: Iterator
: This task is to create a list of a certain size, which is filled with random numbers between 0 and 19, and to create a function which adds a number to every other element in the list. The purpose of this task is to get an example of how lists are made and iterated through in the different languages, which also will give an idea of how the languages are structured in general.

Task 2: Fibonacci Sequence
: This task is to implement an algorithm which calculates numbers of the Fibonacci sequence limited by an input by a user. The purpose of this task is to see how an "introductory" algorithm can be implemented in each language. It is also to have a task that somewhat favors the functional paradigm, as the functional paradigm is meant to be away to express math in a programmatic way.

Task 3: Cups and Ball
: This task is to create a game where a ball is hidden under a cup with 15 cups to choose from, and the goal is to locate the ball. The purpose of this task is two fold, where the first part is

to show how the interactive window work in Scratch and the second part is to have a task that hypothetically favors object oriented languages, and to see how this is coped with in a non-object oriented language. Two of the three environments/languages does not support graphics as such, so there will be a difference in how this task is handled in between the languages.

Task 4: Hangman
This task is to create a hangman game, where the user is supposed to guess a word with only a set amount of wrong guesses. The purpose of this task is to have a larger piece of software as the precious tasks are quite small in a sense. This will hypothetically favor Scratch and BlueJ over Dr. Racket, as the functional paradigm is not really meant for this sort of task.

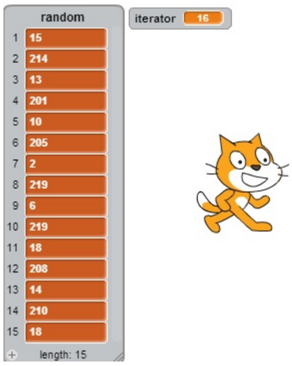
7.1 Scratch

Scratch was chosen blabla TODO: what to write here

7.1.1 Iterator

The iterator is made in a standard iterative way, through a simple loop. The debugging tool when clicking a piece of code is used to show a visual representation of the output in the game window. The control structure is a *repeat* loop, which is the same as the well known *for* loop, only with another name. The code for the program can be seen in Figure 7.1a, and the output is seen in Figure 7.1b.

(a) Scratch Iterator code.

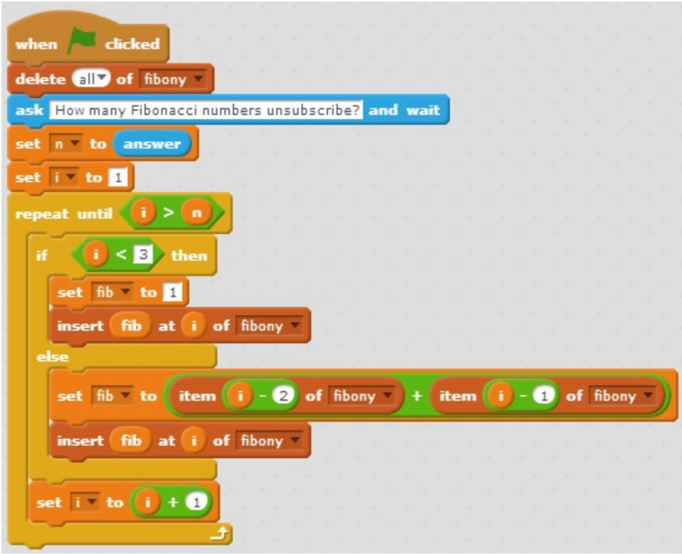


(b) Scratch Iterator output.

Figure 7.1: Code and output for Hangman.

7.1.2 Fibonacci

The fibonacci sequence is done through simple iteration in Scratch. An example can be seen in Figure 7.2a. The user is asked to input how many numbers of the sequence are wanted. With a single loop and a selection, a list is presented, as seen in Figure 7.2b.



(a) Scratch fibonacci code.

(b) Scratch fibonacci output.

Figure 7.2: Code and output for fibonacci numbers.

7.1.3 Cups and Ball

The game of guessing the position of the ball amongst the cups is made with events, as Scratch is able to handle these with blocks. Events happen e.g. when a cup is clicked, cups are cloned, and when the ball is clicked. Code is also attached different sprites, as these work individually to the events. The code blocks for the cups can be seen in Figure 7.3a, and the code blocks for the ball can be seen in Figure 7.3b. A screenshot of the game screen while in a game can be seen in Figure 7.3c.

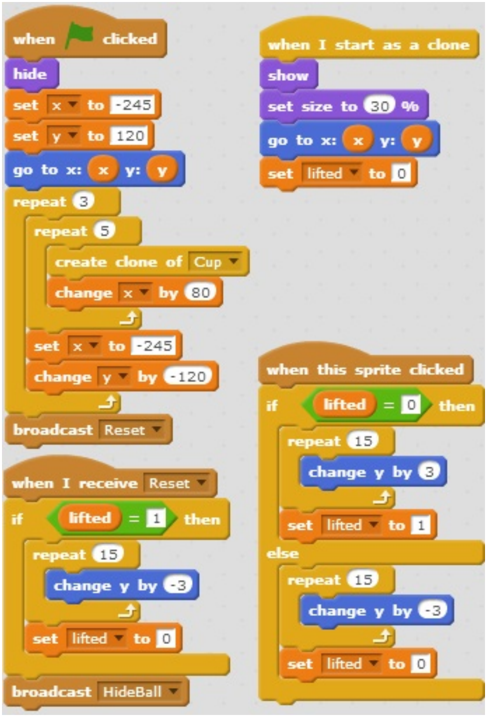
7.1.4 Hangman

The Hangman game is made in an imperative manner. As there are many conditions to take into account, the code is rather long, and there is a lot of control structures. The guessing part itself is a big loop, which can be seen in Figure 7.4a. On the game screen, a list holds the letters for the word to guess, a list holds all the wrong guesses, and a sprite changes for each wrong guess. An input field is provided for guessing. The game screen can be seen in Figure 7.4b.

7.1.5 Criteria Evaluation

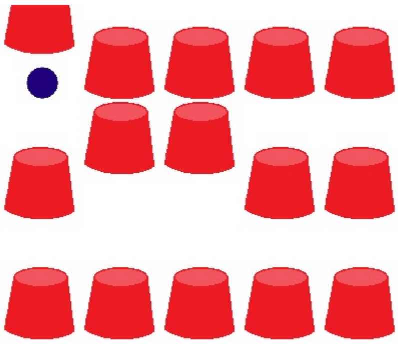
Readability

Scratch is known for its great readability, and this shows when reading it. The colored structures clearly show what the different building blocks are doing. This could lead to a problem with color blind people, as it is not possible to change the colors. The language is very verbose in its statements and declarations, leading to a better understanding of what happens in a block. A problem is the fact that a project becomes very big very fast. An example of this can be seen in



(a) Scratch cup code.

(b) Scratch ball code.



(c) Scratch Cups and Ball output.

Figure 7.3: Code and output for Cups and Ball.

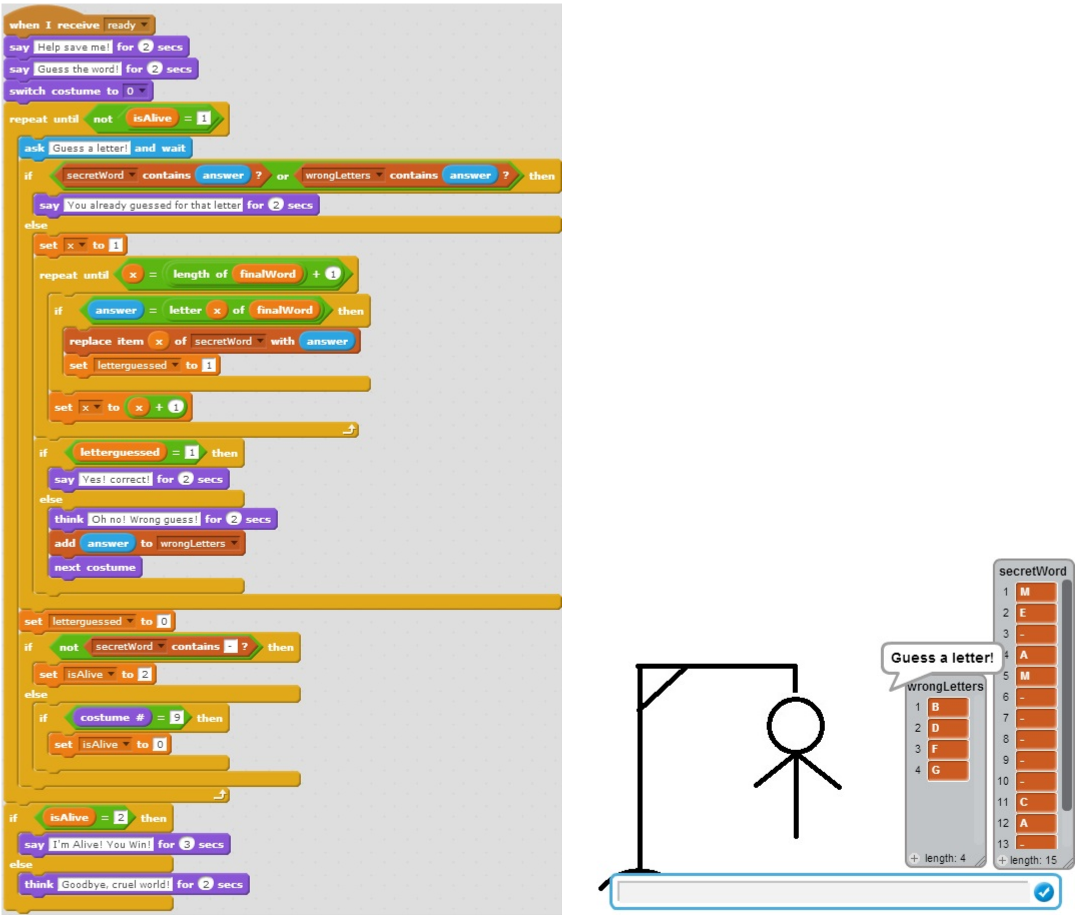
Figure 7.4a, where the collection of blocks seems hard to read at first glimpse, due to its sheer size.

Writability

The visual programming style has its pros and cons. It is easy to create simple structures, but it takes too long for an advanced programmer. It is extremely easy to understand how to use Scratch, due to the fact that it uses building blocks as arguments.

Observability

Scratch has a live game window, where output is shown when compiling code. Combined with the possibility of double-clicking sets of blocks to compile that specific piece of code, the programmer can see the output whenever wanted.



(a) Scratch Hangman code. (b) Scratch Hangman output.

Figure 7.4: Code and output for Hangman.

Trialability

The visual environment in Scratch allows close to no syntax errors. Combined with the level of observability, Scratch has great possibility of recovering from errors, as the error is easily found in the game window. In bigger programs, however, it can be hard to find the error, as one cannot follow the stack. Closing in on a small error that affects the whole program can be hard.

Learnability

Making a game is a good way to capture the attention of children. But to keep them occupied, the process of coding should be intensive in a playful way. Building blocks from the visual style is a way to do it, as building blocks has been proven successful in the context of playing (Lego is an example of this). The lack of conventional syntax, however, can be hindering further in the programmer's career. That means the language is great for novice programming, but it comes to a stop.

Reusability

As Scratch is also a minor game engine, it uses 2D sprites for visualization. Each sprite can contain code local to that sprite. The code shown in Figure 7.3a is the code local to the Cup sprite, whereas the code shown in Figure 7.3b is local to the Ball sprite. As these can be cloned, the code written serves as a blueprint for all instances of the sprite to use. Furthermore, Scratch has the possibility of defining custom blocks. These blocks can bring a collection of blocks down to the size of one, and they can easily be reused. These custom blocks work as functional procedures for the language.

Pedagogic Value

Scratch makes use of the most basic of concepts, such as variables and control structures. That said, there is a huge leap when moving from Scratch to a non-visual programming language. The lack of conventional syntax, which is found in text based programming languages, can confuse a novice when moving on.

Environment

Scratch has a very friendly environment, with great visibility in the color coding. As novices are not familiar with anything code-related, the naming of structures is meaningful and easy to understand. Navigation can be a bit confusing at first, as it can be hard to know what category a needed block is found in. The control of the blocks can be a bit annoying, as disconnecting interlocked blocks does not always go as expected. The game window combined with the block compilation, as mentioned, is a great way of obtaining feedback for trial-and-error.

Documentation

Scratch has an integrated introduction tutorial, which is a great way for novices to learn the language. Furthermore, each building block can be right-clicked to find a documentation page for that specific block. This documentation is easy to read, but is unfortunately only available in English. Furthermore, it is possible to find and share projects via Scratch's homepage.

Uniformity

The basic concepts used in Scratch do the same as in other languages. However, the naming of the expressions and structures are often different, e.g. Scratch has a *repeat* structure, instead of a *for* structure.

Miscellaneous

Scratch uses sprites, but it is very hard to make different sprites work together, as the language itself is not object-oriented. As an example, you cannot reference to a specific sprite or one of its clones. Furthermore, the language has great accessibility in the fact that it is programmed in a browser.

7.2 BlueJ

TODO: Results and discussion.

Since BlueJ uses Java as its underlying language, all the selected tasks will be implemented in imperative, object-oriented manner and there also might be some similarities to their counterparts in Scratch in terms of code structure and use of constructs.

7.2.1 Iterator

The Iterator in BlueJ is implemented using simple iteration through a list with a certain size, specified by the user. Then a *for* loop is used to add a number,in that case 20, to every other element from that list and then the result is printed, as shown in Figure 7.5

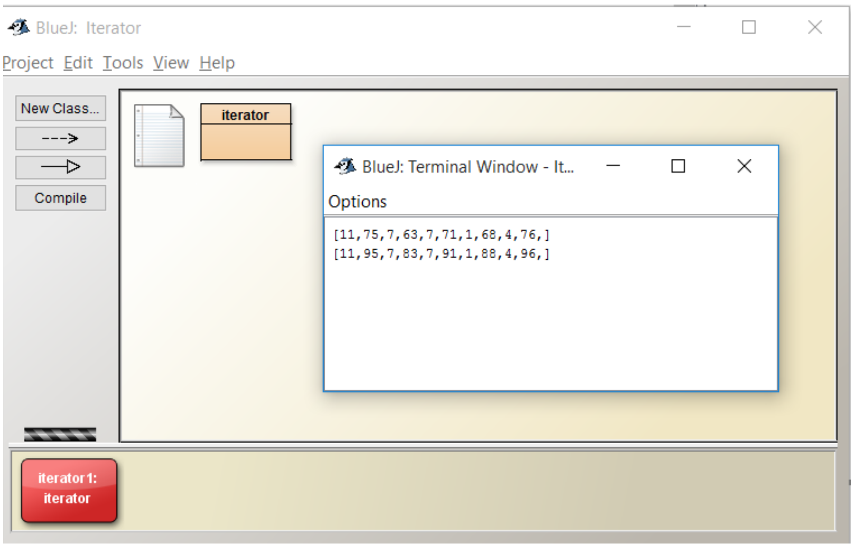


Figure 7.5: Code and output for Iterator

7.2.2 Fibonacci

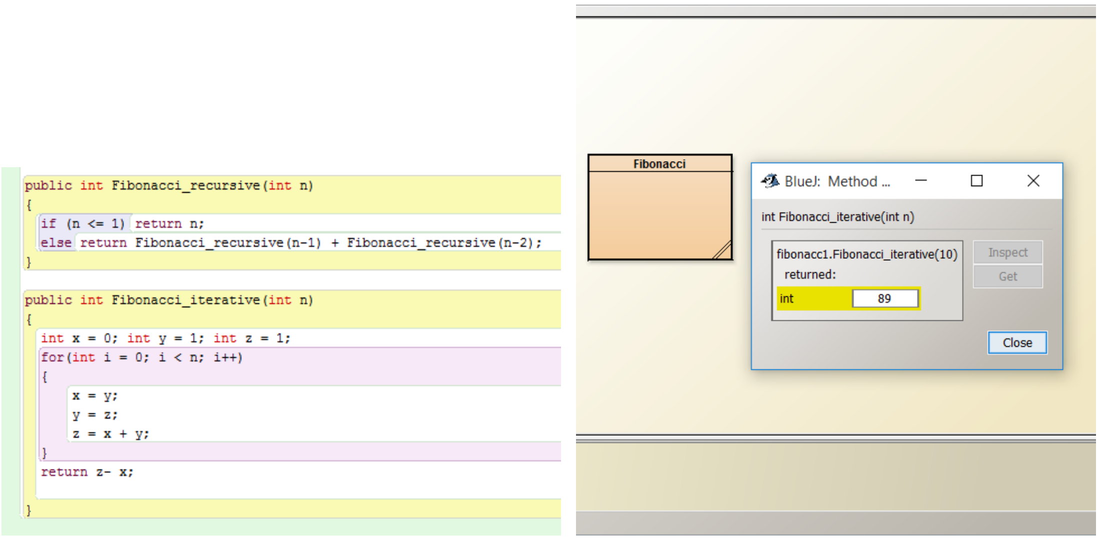
The Fibonacci sequence in BlueJ can be expressed both by using an iterative and recursive approaches. An example could be seen in Figure 7.6b, where the user is prompted to give a number and the respective result is shown.

7.2.3 Cups and Ball

Similarly to how this game was implemented in Scratch, it gives the player the chance to guess where a ball might be among 15 identical cups, with the difference that it feels less intuitive since there is no visual feedback given but rather textual one - that the player has either successfully guessed the position of the ball or not. The example can be seen in Figure 7.7

7.2.4 Hangman

The Hangman is also a guessing game where the player has to guess a particular word by providing one letter at the time. In BlueJ implementation, if the letter is part of the word, the its added to a guessing



(a) BlueJ Fibonacci code. (b) BlueJ Fibonacci output.

Figure 7.6: Code and output for Fibonacci numbers.

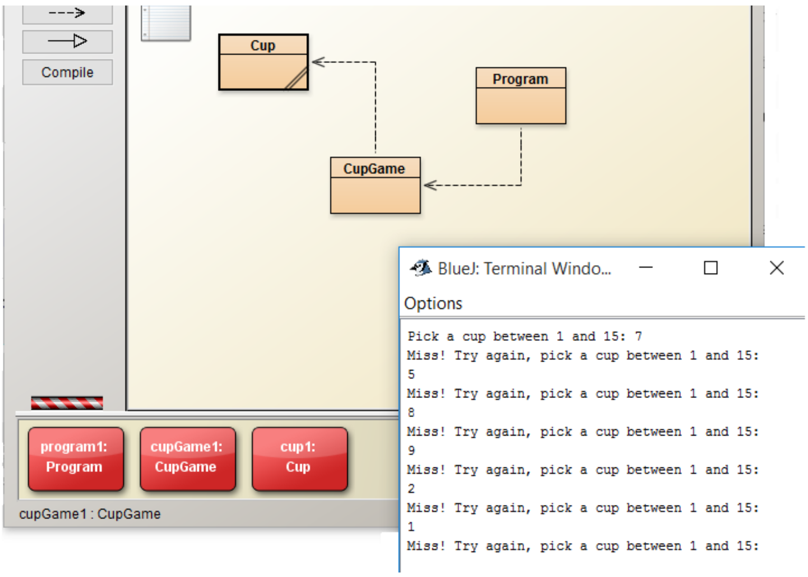


Figure 7.7: Code and output for Cups and Ball

list, otherwise if it not, it is added to a wrong list instead. Every time the player gives a wrong letter, one of his lives is lost, thus the "total lives" the counter decrements by one, and if he is to lose all of them eventually, the game is lost and he has to start all over. The code follows an imperative style of programming with conditional statements and loops, some of which are illustrated on Figure 7.8

```
boolean isDone = false;
boolean isCorrect = false;

while(!isDone)
{
    System.out.println("Guess a letter:");
    String guess = key.nextLine();
    if(guess.length() > 1)
    {
        System.out.println("Please write only a single letter!");
    }
    else
    {
        isCorrect = words.Guess(guess);
    }

    if(!isCorrect)
    {
        hangman.LoseLife();
        System.out.println("You now have " + Integer.toString(hangman.GetLives()) + " left");
    }

    if(!words.GetGuessed().contains(" _ "))
    {
         System.out.println("You Win! The word was " + words.GetGuessedString());
         isDone = true;
         break;
    }
    else if(hangman.GetLives() <= 0)
    {
        System.out.println("You Lose!");
        isDone = true;
        break;
    }
```

Figure 7.8: Code for Hangman game

7.2.5 Criteria Evaluation

Readability

BlueJ has sufficiently good readability both by its textual and visual representations. The Main panel provides an overview of the hierarchy and interactions between different classes and objects giving novice programmers a greater understanding of how different pieces of the code are connected. Additionally, the textual representation provides good indentation and different colour schemes for every method so it is easier to understand the structure of the code.

Writability

As already mentioned in Section 7.2, BlueJ uses Java as its underlying language, therefore it inherits all its pros and cons along. Writing code follows an object-oriented fashion and additionally keywords, such as data types, return statements etc., are highlighted in order to be differentiated by the programmer more easily.

Observability

BlueJ has fairy good observability as it usually manages to quickly and efficiently compile the code giving more time to test different scenarios since it is mainly suitable for small educational projects. Its visual representation provides the option to create instances of objects as visual elements and call different methods specific to these objects in an easy, fast and understandable way.

Trialability

In terms of trialability, BlueJ has a good handling of syntactical and semantic errors, where the need be exceptions are thrown and the code cannot be compiled before those are fixed. Additionally, the problematic sections are highlighted in red with an appropriate message for the give error.

Learnability
Since BlueJ is first and foremost an educational platform, it provides a good way for novice programmers to jump into the object-oriented style of programming with a steady learning curve and sufficiently good documentation.

Reusability
BlueJ provides a great deal of reusability as it is fairly simple to drag and drop classes from other projects and integrate them into the existing one with small modifications.

Pedagogic Value
BlueJ is a powerful educational tool which provides a gentle introduction for novice programmers to the world of object-oriented programming. Given its thorough documentation and exercises with steadily increasing difficulty, BlueJ can give a solid start for everyone with the aptitude for learning.

Environment
Usually, the Integrated Development Environment (IDE) of a language is the only direct way of communication with that language. In the case of BlueJ, it provides the bare essentials needed for learning object orientation, specifically targeting novice programmers without the unnecessary clutter present in more advanced IDEs such as Eclipse or NetBeans. Furthermore, it does not require much effort to install it and create projects in it.

Documentation
As already mentioned, BlueJ has a solid documentation with a lot of reference materials and additional exercises for those who are interested to delve deeper in its features. Also it is fairly easy to get access to the documentation on the BlueJ homepage without the need for any additional books.

Uniformity
BlueJ has a high uniformity compared to other languages since Java is a widespread language and it uses well established language constructs. Its language constructs have mostly predictable behaviour.

7.3 DrRacket

DrRacket is an environment used to learn to write Racket code. Racket is a functional programming language, and therefore DrRacket is our representative for an educational programming environment for the functional paradigm. Worth noting is that all of us have learned to program in an imperative paradigm first and do not have much experience working with functional languages. This will likely impact our code examples and opinion on the criteria evaluation.

7.3.1 Iterator

The first code example is the iterator example. Usually when one wants to apply a function through a list one would use the map function, but since that applies the function to every element in the list, and we want to apply it to every other element the code looks like this instead:

```
(define (iterator inList)
  (if (< (length inList) 2)
      inList
      (cons (car inList) (cons (+ (car(cdr inList)) 20) (iterator (cdr(cdr
         inList)))))))
```

Listing 7.1: The iterator function in DrRacket

The function is a function that takes a list `inLinst` in and returns a list. In the trivial case where the list is shorter than two, the function does not need to be applied to anything and the list is simply returned. Otherwise the second element in the list should be modified and the function recursively called on the remainder. To do this three functions are used: `car` which returns the first element of the list called the head, `cdr` which returns the list minus the first element called the tail, and `cons` which combines a head and a tail to create the bigger list. Firstly we want to combine the unmodified head of the list with the modified tail to construct the new list. We then construct the modified tail by modifying the head of the tail, which modifies the second element in the list, and calling the function recursively on the tail of the tail to go through the rest of the list, and combining these.

7.3.2 Fibonacci

The second example is the Fibonacci implementation:

```
(define (fibonacci n)
  (if (or (= n 0) (= n 1))
      1
      (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))
```

Listing 7.2: The Fibonacci function in DrRacket

This is a simple recursive implementation of Fibonacci with no memory optimizations. Recursion is second nature to functional programming languages, so this is an intuitive implementation. The function takes in a number to find the Fibonacci number of and then calls itself recursively on the two preceding numbers to get the two numbers it needs to sum up. Eventually a trivial case of the called number being one or zero in which case it simply returns one.

7.3.3 Cups and Ball

The next code example is the cups and ball example. This example is intuitively solved in an object-oriented way and since Racket has objects and classes, we do it like that. We define the class like so:

```
(define Cup%
  (class object%
    (define holdsBall 0)

    (super-new)

    (define/public (AddBall)
      (set! holdsBall 1)
      )

    (define/public (HasBall)
      holdsBall)
))
```

Listing 7.3: The Cup class definition in DrRacket

Each object of the class cup has a variable `holdsBall`, which is used to store whether this cup has a ball, were 1 means it has a ball. They also have two functions: `AddBall` which sets `holdsBall` to 1, and `HasBall` which returns `holdsBall`. The code then creates a list of 15 balls and calls `AddBall` on one of them chosen randomly. The main game loop is facilitated with a recursive function `AskUser`:

```
(define (AskUser)
   (define guess (read))
  (if (= (send (list-ref cups (- guess 1)) HasBall) 1)
      (println "Congratulation, you found it!")
     (begin
       (println "Miss! Try again, pick a cup between 1 and 15: ")
           (AskUser))))
```

Listing 7.4: The AskUser function in DrRacket

Here the user is prompted for a number between 1 and 15. The cup on that position on the list then has its `HasBall` function called. If it is 1 the user is congratulated and the game ends, otherwise the user is prompted to guess again and the `AskUser` function is called to repeat the cycle.

7.3.4 Hangman

The final example is the game of hangman. Here we have a list of 11 words all 15 letters long represented by strings. The initial values are then defined:

- `finalWord` is assigned a random string from our list of words and represent the word that should be guessed
- `wrongLetters` represent the list of letters guessed on that were wrong and is assigned to the empty list
- `knownLetters` is the list of correctly guessed letters and their position. This is initialized to a mutable string of 15 underscores.
- `hangman` is the number of lives left and it is initialized to 8.

The user can guess on a letter by calling the guess function with a string. If the string is one char long the function `checkLetter` is called with parameters 0 and the guess string. The function looks like this:

```
(define (checkLetter st l)
  (cond
    [(> (+ st 1) (string-length finalWord))
     (if (and (equal? hasFound 0) (not (member l wrongLetters)))
         (loseLife l)
         #f)]
    [(equal? l (substring finalWord st (+ st 1)))
     (printf "correct guess! on place ~a\n" (+ st 1))
     (set! hasFound 1)
     (string-set! knownLetters st (string-ref l 0))
     (checkLetter (+ st 1) l)
     #t]
    [else (checkLetter (+ st 1) l)]))
```

Listing 7.5: The checkLetter function in DrRacket

The `st` parameter is the index of the char in the `finalWord` string we are looking at and `l` is the guess string. This function checks for three conditions:

- If the index is larger than the length of the string, we are done going through the string. `hasFound` is initialized in `guess` to 0 and if it hasn't changed it means the letter was not found in `finalWord`. If `l` also was not in the `wrongLetters` list, it means the guess was a new wrong guess and `loseLife` is called. `loseLife` reduces `hangman` by one and adds the letter to `wrongList`.
- If `l` is equal to the substring of `finalWord` at index `st` then the guess is correct. The user is notified, `hasFound` is set to 1, the letter at index `st` in `knownLetters` is changed to `l` and the function is called recursively on the next index to go through the rest of the string.
- Otherwise the function is called with the next index to keep looking through the string.

In the end the function `guess` checks for `hangman` being 0 to report a loss and if underscore is still a part of `knownLetters` as otherwise a win is reported.

7.3.5 Criteria Evaluation

In this section we will discuss DrRacket in relation to our criteria. Again since we all are used to another programming paradigm these opinions might be biased.

Readability

DrRacket is fairly readable, with the mandatory use of parentheses giving it a certain structure. However the lack of infix operators hurts the readability since it means that common mathematical operators like plus, minus and greater-than need to be before the two arguments. This breaks the common mathematical notation, which makes it harder to understand intuitively. The heavy use of parentheses however does mean that they sometimes can blend together making it harder to distinguish the operations from each other as can be seen in Listing 7.1. The environment does provide an automatic highlight of the area a pair of parentheses cover when the cursor is next to one, but it does not fix the immediate readability. In general the functional paradigm also has some readability issues since the code usually requires a better overview of the function to understand it. In a more imperative paradigm it is easier to build up an understanding from partial understandings of the sequences.

Writability

DrRacket has good writability, often being able to express things a little more concisely and having a consistent syntax for all sorts of function calls. There is good support for many levels of abstraction with functions being an integral part of the language and it offering constructs for object-oriented programming. Its only problems are a the higher need for overview like in readability, and that the syntax especially around missing parentheses can get hard to keep track of and often requires some debugging. The parentheses highlight tool does help with that quite well though.

Observability

DrRacket has good observability, as it offers an immediate console on runtime, where one can call all the functions and variables from your code and immediately see the return value.

Trialability

DrRacket has good trialability, as the code compiles quickly and easily and whenever an error is encountered it highlights the code it found the error as well as giving the error message. It does have the usual issues with the error message not always being helpful and sometimes reported at other places than where the actual problem is, and you still need to compile every time you want to test out any changes to the code.

Learnability

DrRacket has good learnability with the possibility of using sublanguages to restrict the possibilities available to novice programmers. This allows for slowly opening up the higher level sublanguages to build towards the full language. It also has easily available tutorials to learn the language in this manner. It does however have the problem of not showing the legal constructs to a novice, which means they have to look up the documentation to find examples of how the code should look.

Reusability

DrRacket has great reusability with a strong focus on using functions on many levels and support for objects and classes.

Pedagogic Value

DrRacket has some great pedagogical values in the way that it encourages learning and using recursion, which is a powerful programming tool. Its consistency with function call conventions also help convey the ubiquitous use of function calls. Notably the choice of sticking to the function call syntax instead of allowing infix operators in mathematical expressions, help convey the fact that these are coded as functions in the code.

Environment

DrRacket is a reasonably good environment for Racket. It provides easy setup of compilation to an easily accessible console to evaluate the code with. Its tools for highlighting the parantheses areas and and marking the definition with a line to it when hovering over a variable, both help keep track of the structure. The environment itself however does not provide any assistance for getting started.

Documentation

DrRacket has a lot of good documentation both in the form of easily accessible online tutorials and an online manual with a search function. There are also several books on the subject as well as a decent amount of online forum discussions on Racket and Scheme.

Uniformity

DrRacket has low uniformity relative to the other languages we know like Java, C# and F#. Its function calls has the parentheses around the whole call instead of only the parameters, and the lack of infix operators makes basic mathematical operations look vastly different.

Chapter 8

Comparative Analysis

8.1 Readability

All the three environments we have selected have a varying degree of readability. This is directly tied to the target groups each one tries to address and their respective age. For this reason, we consider Scratch to have the highest readability of the three since it mainly targets the youngest group of kids and it is the only environment of the three which supports a fully visual programming language. That helps immensely with how programs are structured and represented through the use of simple, modular blocks and how easy is to understand different parts of the code. Furthermore, great efforts were made during the development of Scratch so syntactical errors could be avoided for the majority of the cases and in terms of semantics, there is intentionally a limited set of possible data types and blocks which could be given as arguments at any given time which reinforces the intuitiveness of the language.

On second place we have BlueJ which uses Java as its underlying implementation language and it's mainly used in CS1 courses or pre-college courses, involving people of that respected age. This naturally means that object-orientation practices are used for creating programs and there is a heavy emphasis on classes and objects which are modelled after their real world counterparts and highschoolers and students have easier time understanding them. Although not anywhere near the level of Scratch, BlueJ has a visual representation of classes and instances of classes and objects which gives a better overview and understanding of the hierarchical structure of programs than traditional IDEs used with Java.

We place DrRacket last in terms of readability since its entirely textual approach with combination of unconventional use of mathematical operations, brackets and lack of infix operators makes it the hardest to understand environment for novice programmers of varying age, in terms of its underlying programming language adhering to the functional paradigm.

8.2 Writeability

Similarly to how the three platforms were evaluated according to their readability, we consider Scratch the environment with the highest writeability, followed by BlueJ and lastly - DrRacket. Programs in Scratch are not only easy to understand but also relatively easy to make. It is actually the only one from the three environments which does not require any programming knowledge but rather intuition, intended results and trial-and-error since it is specifically designed for young kids. Compared to BlueJ and DrRacket which have a compilation step, Scratch extends its writeability through its reactive nature, giving immediate results from an executed code and faster time in correcting mistakes. Keywords in the language are named in such a way that are more descriptive of the action they convey and more intuitive in terms of understanding compared to the traditional approach used both by BlueJ and DrRacket.

BlueJ takes the second place since it does not provide much more beyond the standard in terms of writability established by Java, with the exception of the option to colour code, where different parts of the code take whatever colours the user specifies, taking into account colour blind people as well.

DrRacket takes the last place since the style of programming defined by the functional paradigm seems the least intuitive to work with. Additionally, as already mentioned in 7.3.5, this is compounded by the fact that the language lacks infix functions, thus common mathematical operations have different syntactical structure.

8.3 Observability

In terms of observability Scratch is the best of the three languages. The visual feedback allows for a greater range of feedback, that can still be intuitively read. The ability to simply click on a code block to run it and see the results also makes the code to feedback time really small. On the second place we have DrRacket. It only requires a click for compilation and sometimes a function call, where BlueJ requires instantiation of an object as well.

8.4 Trialability

In terms of trialability Scratch wins again albeit not as much. Its quick feedback-loop and the decomposable execution are great factors for trialability, but its lack of a slower runtime evaluation mode hurts it a bit. DrRacket and BlueJ are very close, with both of them highlighting the error code on encounter amongst other features. DrRacket only wins here on account of being an instantiation of objects shorter than BlueJ in its feedback-loop.

8.5 Learnability

Scratch is the clear winner in learnability. The list of operations both serve to give some suggestions for actions, and helps give an overview of the possible actions. This is something both DrRacket and BlueJ lack, as the textual coding interface requires the user to have an idea of how the syntax and semantics work before they can make anything. This means that they both rely on some additional instruction, either through an instructor or a tutorial, before the user can start working. Neither of the two provide an in-environment tutorial unlike Scratch.

8.6 Reusability

Both BlueJ and DrRacket has good features supporting this criteria, both supporting classes and objects. Although, BlueJ is a little ahead of DrRacket in this criteria as it is using an object-oriented programming language which posses a great deal of features for reusing code through abstraction. This fits well with the definition of the criteria shown in Section 6. Even as DrRacket has support for classes and objects, it is not the main focus of the language which places it a bit behind BlueJ. Scratch has limited features supporting this criteria with its cloning of sprites and function definitions, lacking features for abstraction.

8.7 Pedagogic Value

Each of the languages/environments provides pedagogic values for different aspects of programming. Scratch provides it for basic low level concepts, but has a possible shortcoming in the sense that a user do not have access to some of the functionality Scratch provides, when moving on to a text based programming language which might confuse them. BlueJ does not provide a lot regarding basic concepts of programming, but provide a lot of tools to get an understanding of how classes and objects works. DrRacket provide pedagogic value in the sense that it can provide an understanding of the underlying functionalities in programming languages. An example of this is that in a lot of programming languages, the user will use the expression $2 + 2$ to get the value 4, possibly assuming that it “just works”, where DrRacket shows that using the $+$ operator is a function call in itself.

8.8 Environment

As scratch is a visual programming language, its environment differs greatly from both BlueJ and DrRacket. With this said, BlueJ also offers a visual representation of the structure. DrRacket has no graphical advantages that would help novices in learning to code, other than libraries that provide high order functions for creating shapes as an output. This results in Scratch and BlueJ being better for a younger targeting group. As an addition to this, Scratch also functions as a small game engine, with a lot of visual representations of its mechanics and output. This, along with the playful nature of the environment, makes Scratch the environment of the three best fitting children in primary school.

BlueJ is used for programming Java, and its environment is excellent in learning novices the concepts of objects and classes. It provides a lot of helping functionality, which is not found in other development environments for the language. Scratch is good for teaching younger novices the basic concepts of imperative coding. DrRacket helps in understanding declarative programming, which is much different from most well-known paradigms. They each have their merits, but as an educational tool for teaching novices, Scratch has the most helpful environment.

8.9 Documentation

Each of the language environments have different ways of providing documentation. BlueJ has an advantage in being an environment for programming Java, which leads to a lot of documentation and helping forums. Furthermore, it has documentation on how to use the environment itself. Scratch as a language is very dependent on its environment, as it is a visual language, which leads to a mixture of environment and programming documentation. DrRacket has little documentation on how to use the environment as a whole, but more about how to use it to code. This means most of the documentation is about Racket as a language.

There is a possibility of finding online tutorials for all the languages. Scratch is very educationally friendly in that area, as it provides a tutorial as a part of the programming window. DrRacket also offers a very thorough guide and reference work on their homepage, and BlueJ offers a paper, serving as a tutorial and guide for the environment. The documentation for BlueJ is unfortunately not very novice friendly, as it assumes the programmer is known in Java.

8.10 Uniformity

The three languages have little in common overall, which means a lower rating of internal uniformity. As BlueJ is an environment of a well known programming language, its uniformity is unquestionable when it comes to object-oriented programming. Scratch is very simplistic, and its basic constructs are similar to those in widely used languages, although they are differently named. It also has an advantage for novices, as the declarations often don't require knowledge of coding conventions for that type of expression. As an example, the block setting the value of a variable shows "set [NAME] to [VAL]", instead of the traditional "[NAME] = [VAL]". In Scratch, [NAME] is even a drop-down list of possible variables, and [VAL] can both be written directly, or be another variable block. DrRacket is very different from well known languages, which leads to a low score in uniformity. The lack of infix operators and other convention in parentheses makes it hard to transition from this language to other languages.

Chapter 9

Results

TODO: Results and discussion.

Chapter 10

Discussion

Chapter 11

Conclusion

Chapter 12

Further Works

Part III

Bibliography

Bibliography

[1] R. N. S. B. E. E. Maloney John, Resnick Mitchel, “The scratch programming language and environment,” *Journal Name*, vol. 10, no. 16, 2010. 1, 3.2.1

[2] C. Schulte and J. Bennedsen, “What do teachers teach in introductory programming,” *ICER ’06 Proceedings of the second international workshop on Computing education research*, pp. 17–28, 2006. 1

[3] S. Garner, P. Haden, and A. Robins, “My program is correct but it doesn’t run: A preliminary investigation of novice programmers’ problems,” *ACE ’05 Proceedings of the 7th Australasian conference on Computing*, vol. 42, 2005. 2.1, 2.3

[4] Cunningham & Cunningham, Inc., “Functional programming.” `http://c2.com/cgi/wiki?FunctionalProgramming`, 2011. Used: 15/11/15. 2.3

[5] M. McMillan, “Teaching functional programming techniques in non-functioanl programming languages,” *Conference Tutorial*, vol. 27, pp. 66–67, 2012. 2.3

[6] S. Xinogalos, “Object-oriented design and programming: An investigation of novices’ conceptions on objects and classes,” *ACM Transactions on Computing Education*, vol. 15, no. 3, 2015. 2.3

[7] S. Papert, *Mindstorms: Children, Computers, And Powerful Ideas*, p. 218. Basic Books, Inc., 1 ed., 1980. 3.1.1

[8] S. Papert, *Mindstorms: Children, Computers, And Powerful Ideas*, pp. 11–15. Basic Books, Inc., 1 ed., 1980. 3.1.1

[9] A. Kay, “Alan kay shares a powerful idea about teaching ideas.” `https://www.youtube.com/watch?v=JDpsXWuedVc`, 2014. Used: 26/09/15. 3.1.1

[10] Microsoft, “Small basic faq.” `http://smallbasic.com/faq.aspx`. Used: 26/09/15. 3.1.1

[11] Microsoft, “Microsoft small basic, an introduction to programming.” `http://download.microsoft.com/download/9/0/6/90616372-C4BF-4628-BC82-BD709635220D/Introducing%20Small%20Basic.pdf`. Used: 26/09/15. 3.1.1

[12] Microsoft, "Small basic curriculum: Online." http://social.technet.microsoft.com/wiki/contents/articles/16982.small-basic-curriculum-online.aspx. Used: 26/09/15. 3.1.1

[13] T. Software, "Tiobe index for october 2015." http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Used: 26/09/15. 3.1.2

[14] Qourum, "Epiq 2016." http://quorumlanguage.com/epiq.php. Used: 26/09/15. 3.1.2

[15] M. Kölling, "Bluej features overview." http://www.bluej.org/about.html. Used: 01/11/15. 3.1.3

[16] M. Kölling, "Bluej blackbox data collection project." http://www.bluej.org/blackbox.html. Used: 01/11/15. 3.1.3

[17] B. team, "Bluej blackbox data collection for researchers." https://blackboxdc.wordpress.com/. Used: 01/11/15. 3.1.3

[18] racket lang.org, "2.3 how to design programs teaching languages." http://docs.racket-lang.org/drracket/htdp-langs.html. Used: 16/12/15. 3.1.4, 5

[19] S. W. Koitz Roxane, "Empirical comparison of visual to hybrid formula manipulation in educational programming languages for teenagers," *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, pp. 21–30, 2014. 3.2

[20] MIT Media lab, "Scratch statistics." https://scratch.mit.edu/statistics/. Used: 28/09/15. 3.2.1

[21] L. Rhodes, "Design criteria for programming languages." http://jcsites.juniata.edu/faculty/rhodes/lt/plcriteria.htm, 2015. Used: 23/11/15. 6

[22] University of Washington, "Evaluating programming languages." http://courses.cs.washington.edu/courses/cse341/02sp/concepts/evaluating-languages.html, 2015. Used: 23/11/15. 6

[23] D. Benyon, *Designing Interactive Systems*. Addison Wesley, 2 ed., 2010. 6