

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

Abstract:



Supervisor:

Bent Thomsen

Pages: 39

Appendices: 0

Copies: 2

Finished: 1 June 2015

The content of this report is publicly available, publication with source reference is only allowed with authors' permission.

Contents

1	Introduction	1
I	Problem Analysis	3
2	Error-Prone Areas for Novices	4
2.1	Syntax and Semantics	4
2.2	Pragmas	5
2.3	Programming Paradigms	5
3	Programming Languages and Tools	7
3.1	Text Based Programming Languages	7
3.1.1	Text Based Educational Programming Languages	7
3.1.2	General Purpose Programming Languages	10
3.1.3	Dr. Racket	12
3.2	Visual-based Programming Languages	14
3.2.1	Scratch	14
3.3	Comparison	15
4	Teachers and Programming	16
4.1	Current State	16
4.2	Discussion	17
II	Language Comparison	18
5	Preliminaries	19
6	Criteria	20

6.1	Criteria Evaluation	21
7	Comparative Analysis	22
7.1	Scratch	23
7.1.1	Fibonacci	23
7.1.2	Cups and Ball	24
7.1.3	Hangman	25
7.1.4	Criteria Evaluation	25
7.2	BlueJ	27
7.2.1	Fibonacci	27
7.2.2	Cups and Ball	27
7.2.3	Hangman	28
7.2.4	Iterator	29
7.2.5	Criteria Evaluation	29
7.3	DrRacket	29
7.3.1	Fibonacci	29
7.3.2	Cups and Ball	30
7.3.3	Hangman	31
7.3.4	Criteria Evaluation	32
8	Usability Evaluation	34
9	Results	35
III	Bibliography	36
	Bibliography	37

Chapter 1

Introduction

Traditionally, programming have 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 Java[TODO: 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, the 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 work with teaching the skills for professional programming using the already taught knowledge from primary school. To do this we first need to establish what level people have after working in the educational

language and what knowledge is needed to work professionally with programming. This paper aims to do this as well as suggesting some ideas for ways to bridge that gap.

TODO: JAIS: Should we mention the problem with who is actually teaching the kids in primary school? There is a problem in Denmark, as it is the physics teacher, as far as I know. TODO: JAIS: We need some more stuff here, perhaps chapters or sections. Such as Preliminaries, Previous Work, and a section where we mention the initial problems.

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, which 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, instead 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 its 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 OO features [3]. This leads to a problem of both understanding the very basic concepts of programming, such as control structures (loops and selections), and understanding the OO approach.

Functional programming 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]. **TODO: JAIS: So, what keeps people from using functional programming in stead of imperative?**

It is discussed widely what approach is the most efficient teaching method (**TODO: need some refs here**). For instance, some say it is necessary to learn the concepts of OOP before learning to code, and some say the basic constructs are necessary before learning about different paradigms and advanced structures. The procedural approach is being taught in elementary school in various languages (**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 though this approach, or the classic “algorithms-first approach” [?]. 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 students¹ 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].

¹From 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 updated². 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 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.

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

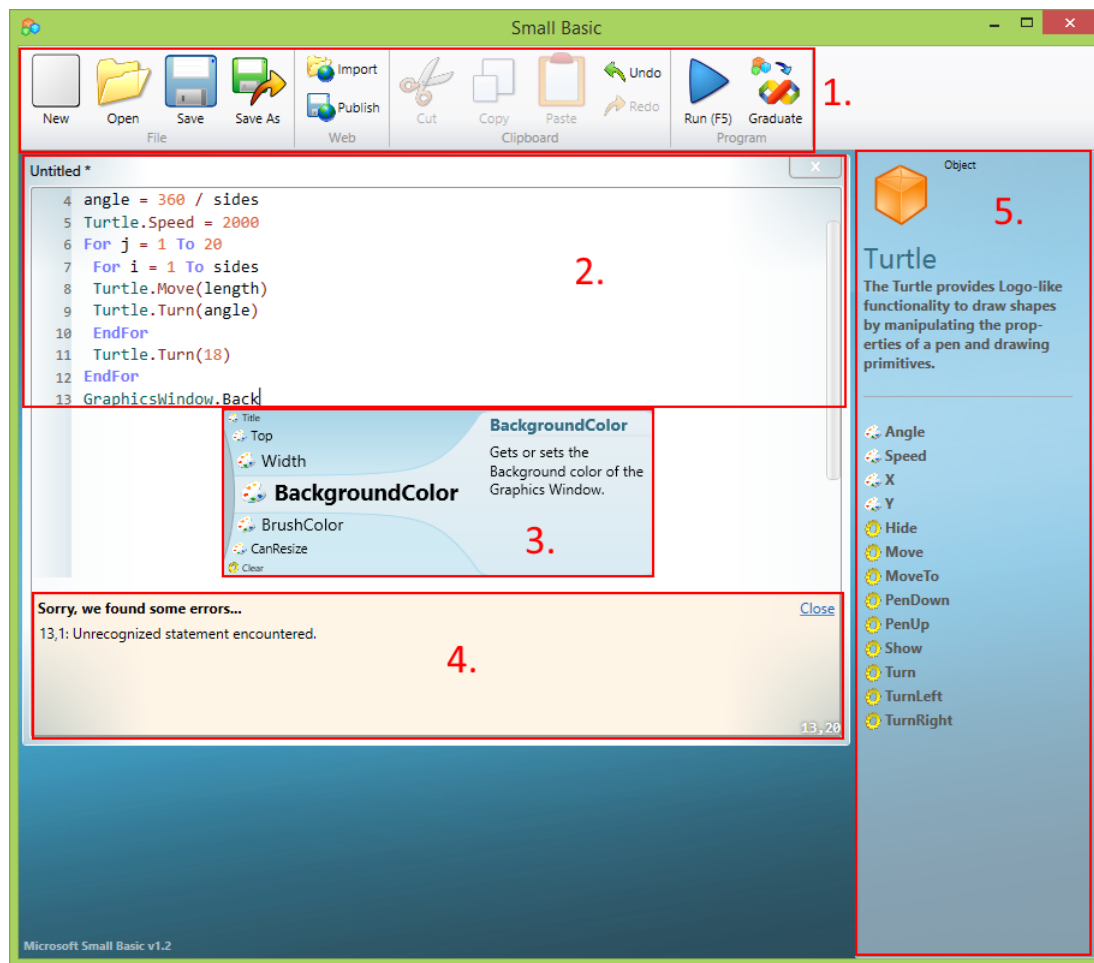


Figure 3.1: Small Basic IDE

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].

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].

3.1.3 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 website⁴.

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.

TODO: Good thing about text based, is terminology, one learns to talk about programming, e.g. what is a statement?

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

⁴<http://docs.racket-lang.org/>

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

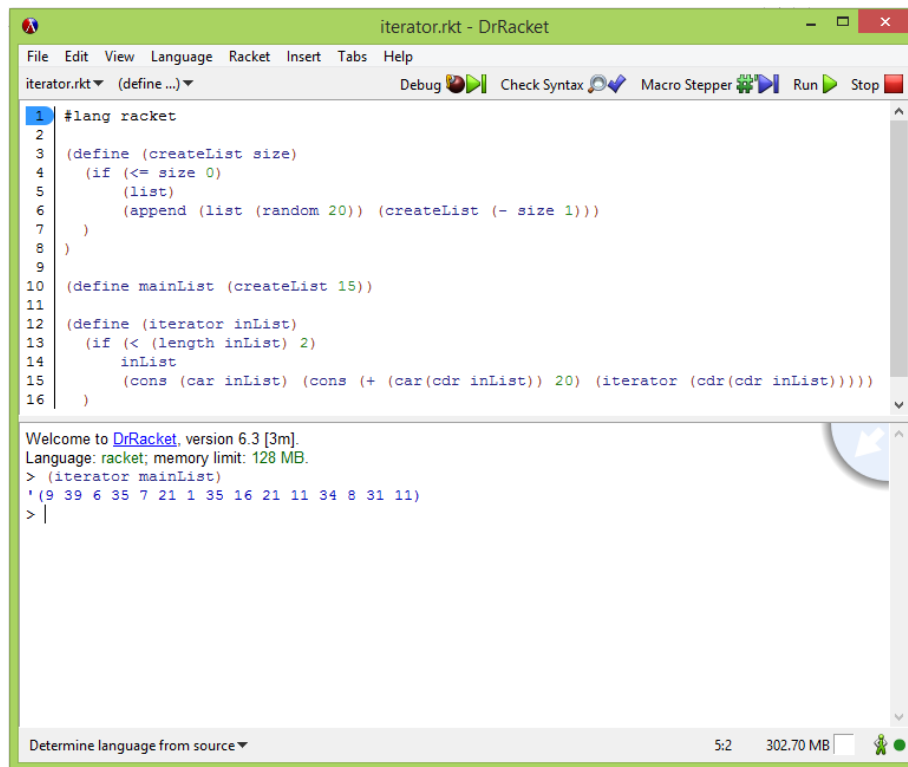


Figure 3.2: Dr. Racket IDE

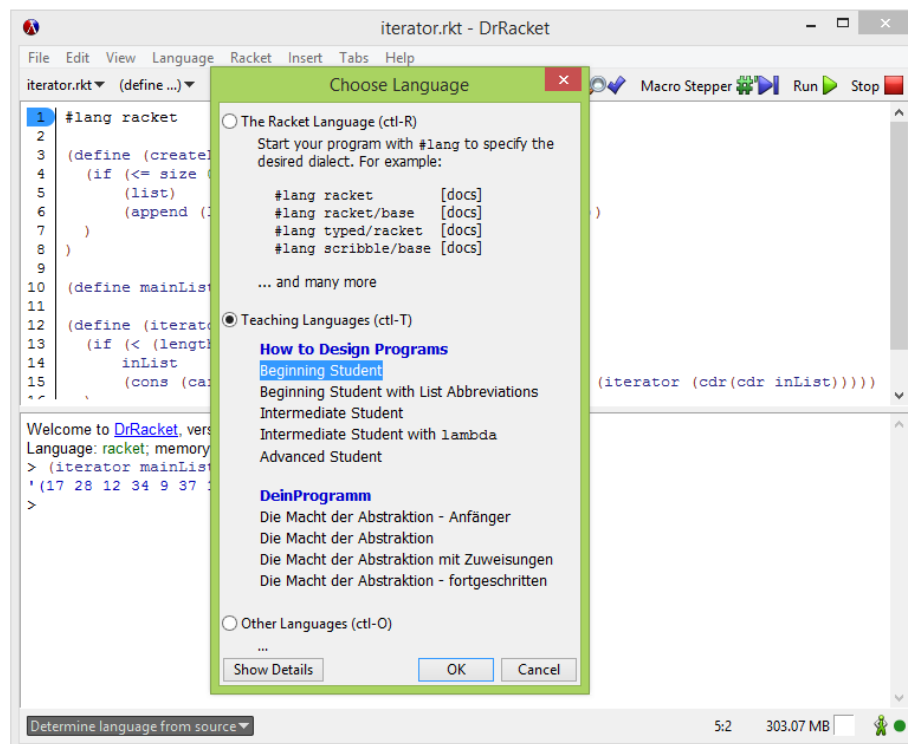


Figure 3.3: Dr. Racket, Choose Language Window

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 and 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

TODO: A general comparison on text-based and visual programming languages. Not on the specific languages chosen, but on the concepts and their fit for novices and teaching.

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 Portal¹ 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 course²:

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>

²This 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 related³. Given these possible problems, it might be interesting to make research in this area to determine if these end up being actual problems.

³These 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 to 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

Comparative 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

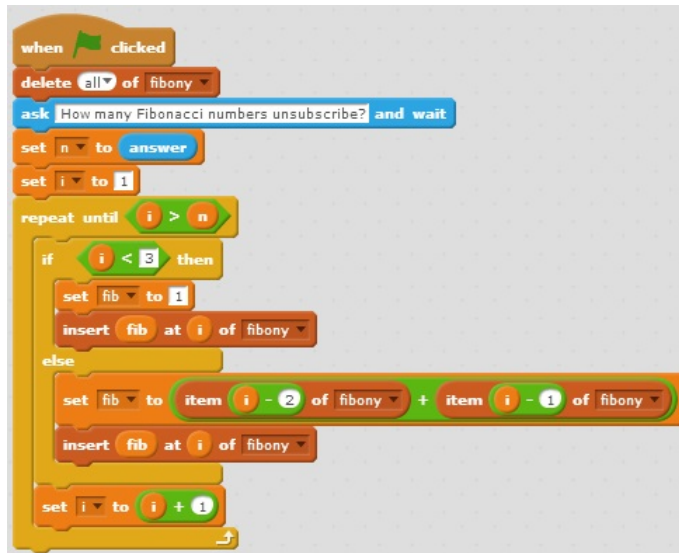
This task is to create a hangman game, where the user is supposed to guess a word with only a set amount of wrong guesses. The purpose of this task is to have a larger piece of software as the precious tasks are quite small in a sense. This will hypothetically favor Scratch and BlueJ over Dr. Racket, 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 Fibonacci

The fibonacci sequence is done through simple iteration in Scratch. An example can be seen in Figure 7.1a. 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 7.1b.



(a) Scratch fibonacci code.



(b) Scratch fibonacci output.

Figure 7.1: Code and output for fibonacci numbers.

7.1.2 Cups and Ball

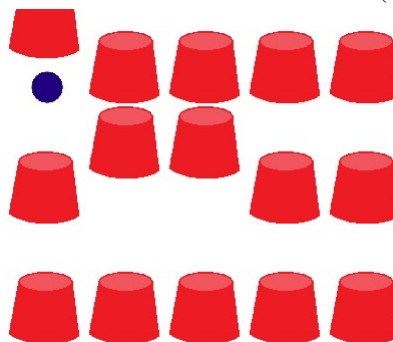
The game of guessing the position of the ball amongst the cups is made with events, as Scratch is able to handle these with blocks. Events happen e.g. when a cup is clicked, cups are cloned, and when the ball is clicked. Code is also attached different sprites, as these work individually to the events. The code blocks for the cups can be seen in Figure 7.2a, and the code blocks for the ball can be seen in Figure 7.2b. A screenshot of the game screen while in a game can be seen in Figure 7.2c.



(a) Scratch cup code.



(b) Scratch ball code.

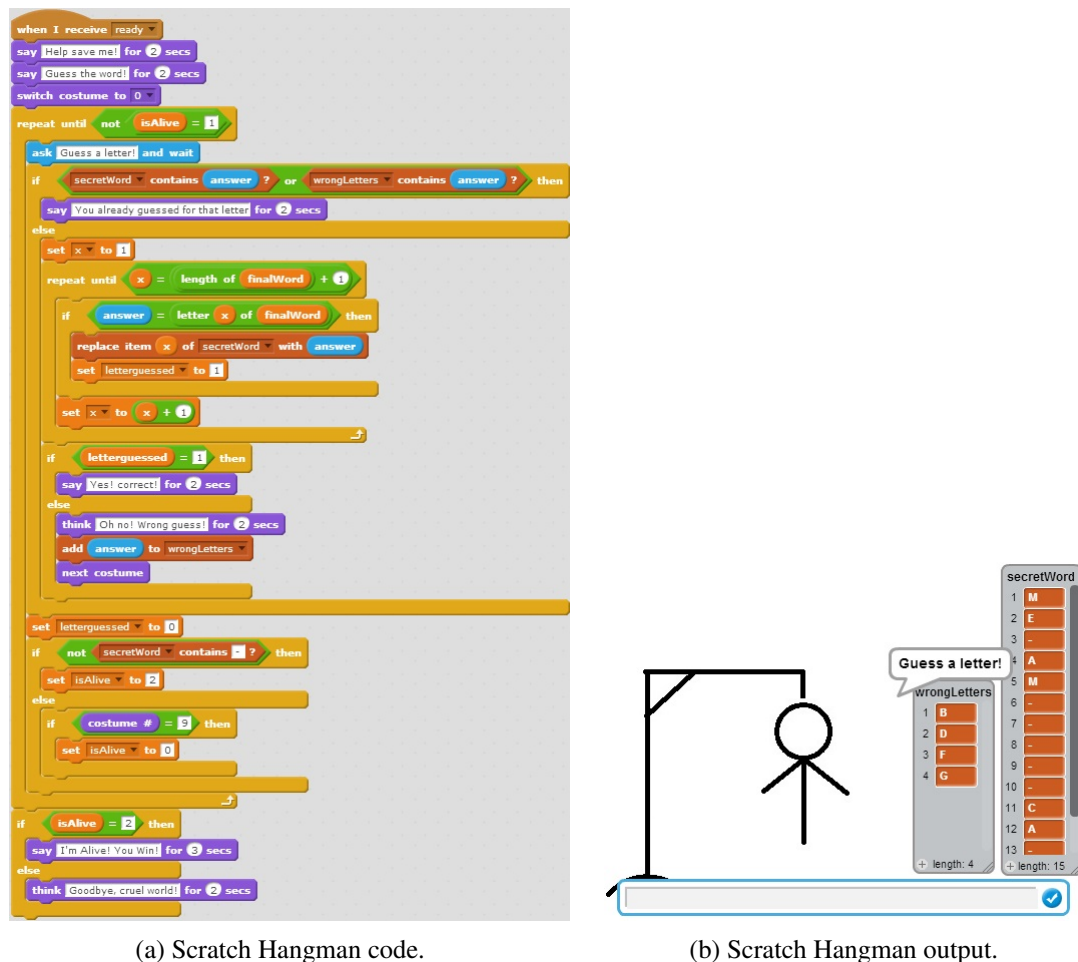


(c) Scratch Cups and Ball output.

Figure 7.2: Code and output for Cups and Ball.

7.1.3 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 7.3a. 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 7.3b.



(a) Scratch Hangman code.

(b) Scratch Hangman output.

Figure 7.3: Code and output for Hangman.

7.1.4 Criteria Evaluation

Readability

Scratch is known for its great readability, and this shows clearly when reading it. The colored structures clearly show what the different building blocks are doing, and variables are easy to identify. We believe this is a great advantage for novices

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, the programmer can see the output whenever wanted.

Trialability

The visual environment for 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.

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. The lack of proper 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. As these can be cloned, the code written serves as a blueprint for all instances of the sprite to use. Sadly, it is the only way for Scratch to reuse code.

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 missing syntax found in text based programming language 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 disconnection 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.

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.

7.2 BlueJ

TODO: Results and discussion.

7.2.1 Fibonacci

The Fibonacci sequence in BlueJ can be expressed both by using an iterative and recursive approaches. An example could be seen in Figure 7.4b, where the user is prompted to give a number and the respective result is shown.

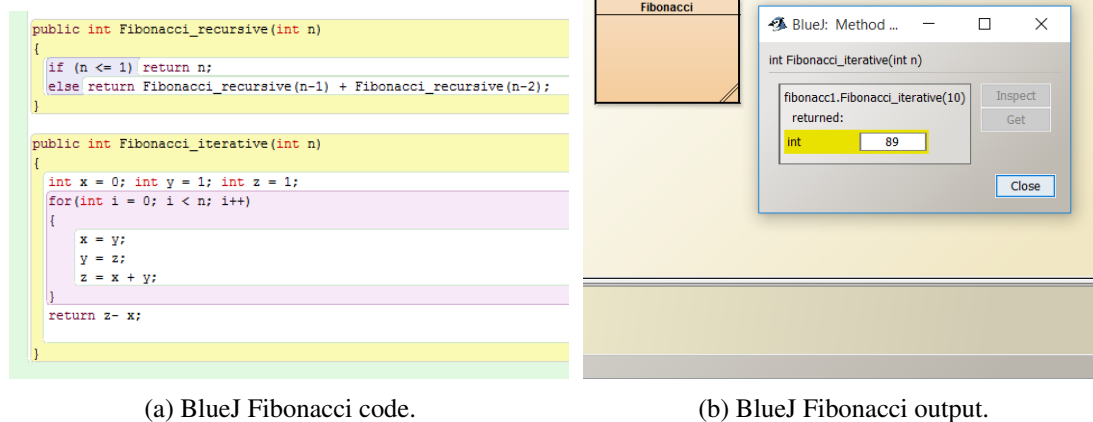


Figure 7.4: Code and output for Fibonacci numbers.

7.2.2 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, but 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 7.5

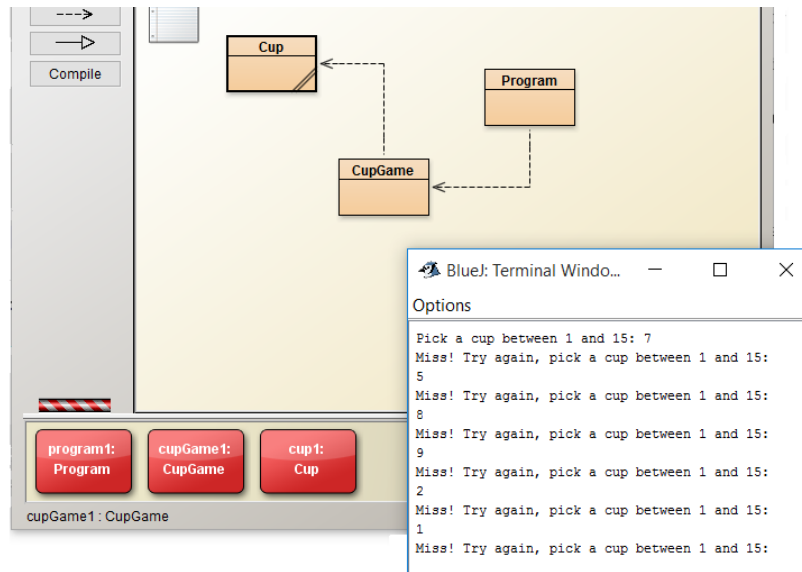


Figure 7.5: Code and output for Cups and Ball..

7.2.3 Hangman

The Hangman is also a guessing game where the player has to guess a particular word by providing one letter at the time. If the letter is part of the word, the its added to a guessing list, otherwise if it not, it is added to the wrong list instead. Every time the player gives a wrong letter, one of his lifes is long, and thus he loses the game if he loses all of them. Since BlueJ uses Java as the underlying language, the code follows an imperative style of programming with conditional statements and loops, some of which are illustrated on Figure 7.6

```
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 7.6: Code for Hangman game

7.2.4 Iterator

7.2.5 Criteria Evaluation

Readability

Writability

Observability

Trialability

Learnability

Reusability

Pedagogic Value

Environment

Documentation

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 Fibonacci

The first code example we will show in DrRacket is the Fibonacci implementation:

```
1 (define (fibonacci n)
2   (if (or (= n 0) (= n 1))
3       1
4       (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))
```

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.

7.3.2 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:

```
1 (define Cup%
2   (class object%
3     (define holdsBall 0)
4
5     (super-new)
6
7     (define/public (AddBall)
8       (set! holdsBall 1)
9     )
10
11    (define/public (HasBall)
12      holdsBall)
13  ))
```

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`:

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

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.

7.3.3 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:

```
1 (define (checkLetter st l)
2   (cond
3     [(> (+ st 1) (string-length finalWord))
4       (if (and (equal? hasFound 0) (not (member l wrongLetters)))
5         (loseLife l)
6         #f)]
7     [(equal? l (substring finalWord st (+ st 1)))
8       (printf "correct guess! on place ~a\n" (+ st 1))
9       (set! hasFound 1)
10      (string-set! knownLetters st (string-ref l 0))
11      (checkLetter (+ st 1) l)
12      #t]
13    [else (checkLetter (+ st 1) l)]))
```

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.

7.3.4 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 hurt the readability since it means that common mathematical operators like plus, minus and greater-than need to be before the two arguments breaking 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. 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, where a more imperative paradigm is easier to build a partial understand from the sequence.

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

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

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

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]. I ain't got time for that!

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.

Miscellaneous

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

Chapter 8

Usability Evaluation

TODO: Do the experiment, then write setup and results here.

Chapter 9

Results

TODO: Results and discussion.

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-functionanl 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. 3.1.2
- [16] M. Kölling, “Bluej blackbox data collection project.” <http://www.bluej.org/blackbox.html>. Used: 01/11/15. 3.1.2
- [17] B. team, “Bluej blackbox data collection for researchers.” <https://blackboxdc.wordpress.com/>. Used: 01/11/15. 3.1.2
- [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. 3.1.3, 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, 7.3.4

