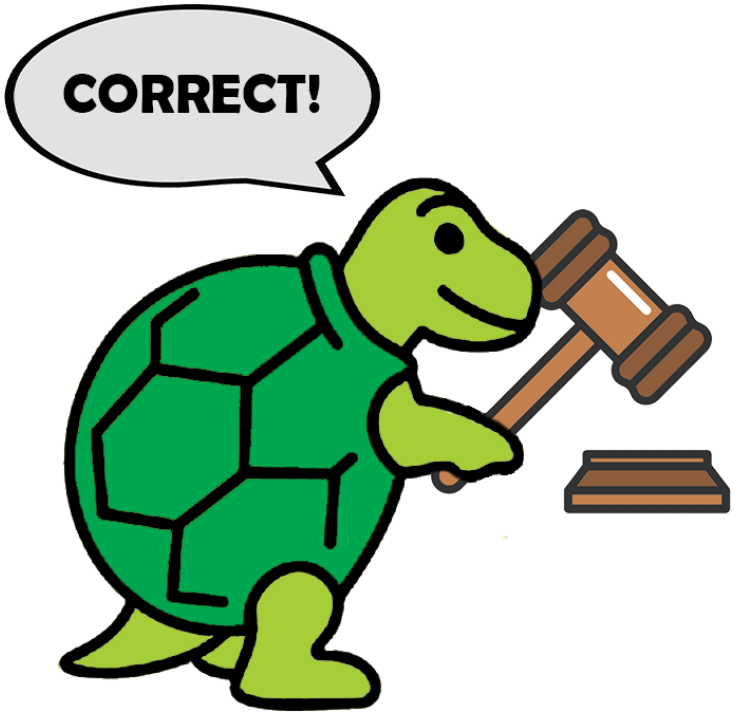
Well done! Keep it up! XLogoOnline Turtle is ready to judge your code



Dominic Weibel

BACHELOR THESIS
17th March 2019

Supervising Professor:
Prof. Dr. Juraj Hromkovič

Supervisor:
Jacqueline Staub

ETH zürich

Abstract

Teachers do not have an easy job preparing lessons and grading their student's work. But now, as programming is a mandatory subject in every child's basic education, it is even harder for them. Holding a programming class is a new subject for most of the teachers and the lack of prior experience makes them feel anxious to teach something utterly new. Teachers need to tutor, praise and support their pupils. But with 25 children, they cannot support everyone at the same time. Weaker children need help while stronger ones want a challenge. The judge presented in this work automatically evaluates programs written by novice programmers and provides them with clear tasks and concise feedback. With this, we want to support teachers and provide them with a tool they can use and rely on.

This judge will be able to decide whether a proposed program is a correct solution to a given problem. Despite there being infinitely many correct solutions to a given problem, the judge is able to distinguish correct from incorrect. The problem to solve is provided as a picture and the proposed solution picture has to match it exactly. We use this as a definition of correctness.

Such a judge also allows the addition of a new component to the XLogoOnline suite: a daily brainteaser. The idea of a daily task is to keep the kids motivated to solve a task at home for themselves and competing with others. With the judge, we can automatically evaluate the submitted solutions to provide feedback, which should on one side further improve the programming skills of children at home, away from the classroom. On the other side, our judge can relieve the teacher during class and provide a place where the kids can prove their skills.

While testing the judge, we received plenty of positive feedback and high acceptance both among the students and the teachers. More than three quarters of them would try out our judge once it is published.

Acknowledgment

When I finished my last but one semester in computer science, I thought about whether I should allow myself one easy semester or whether I should continue on to do my bachelors thesis in the next semester. But when I met Jacqueline following an email-exchange about different projects ready to be realized, I already knew I could not wait a whole semester to start my thesis. The overall subject simply drew my attention and I already wanted to start working on it. This was because the project promised a few things I really cared about when choosing my bachelor thesis' theme: Working in the field of education with children, developing a new part of a programming language IDE and, most important of all, the promise that my work will be put online and used when finished.
I would like to thank everyone who accompanied me on this road: Jaqueline Staub, my supervisor, who often helped me to stay on track and to get motivated again when confronted by unforeseen difficulties. She also got me an office space in the CAB building, which I really appreciated during the whole course of my thesis. For the weekly meetings and especially the feedback I got from her in the last few weeks of writing this thesis I am very thankful.
Anna-Laura John, with whom I shared my office in the CAB: thanks for all the fun moments, the help and simply all the laughter you brought to the office.
Prof. Juraj Hromkovič, I'd like to thank you for your valuable feedback and the spontaneous desert brakes you invited me and the rest of the group to.
Jacqueline Sennhauser, my flatmate from overseas, thanks for proofreading the whole work and for your valuable feedback concerning my syntactic errors.
Thank you Heinz Hofer, Beat Staub and Heidi Staub for letting me sleep at your house the day before the classroom evaluation. It was like being on a relaxing holiday.

Contents

1

Introduction

With this chapter, we introduce the reader to the subject of this work. We provide the readers with the necessary background information about the education of children and the programming language XLogo and one of its online environments, XLogoOnline. Furthermore, we discuss why this work is relevant and necessary, what its goals are and lastly give a brief outline of this paper.

1.1 Background

Talking to computers is a lot like talking to another human: you need a specific language that both you and your opposite can understand. In order to talk to a computer, several different programming languages were invented. Knowing how to program is not only a necessary thing in today's increasingly digital world, but also teaches key concepts in logical thinking and problem solving. Teaching children how to program is not only providing them with a new language they learn, but also providing a different way to think. It involves thinking exactly, logically and unambiguously as well as active explorational, constructive and step by step problem solving. A key learning point in developing complex programs is the modular approach. Additionally, it offers an introduction into the world of computers and computer science [4]. That's why it is important to have programming classes starting as early as primary school.

Establishing programming as a part of each child's basic education in Switzerland was no easy task. For the past 15 years, we, the Chair of Information Technology and Education, have been fighting for computer science to be added as a standalone subject in the mandatory curriculum in primary school. And now finally it is present in the "Lehrplan" of 21 cantons in Switzerland.

Although every canton has its own way to interpret the studies of computer science, each individual child will come in touch with computers and programming.

Children love to discover and try out new things. Take, for example, programming with a computer. This is something new, something, that was not present before, a new way of thinking. Although the first steps are not easy and need lots of practice and patience, as soon as a basic understanding is established, children see the fun in talking to computers and let it do things for them. One way to learn how to program is by learning how to steer a turtle using the programming language XLogo. It is explicitly developed to offer an easy introduction to programming, while not posing many hurdles in the understanding of the syntax of XLogo.

In XLogo, children commandeer a turtle equipped with a pen and make it draw different geometrical shapes. It is perfectly suited for complete beginners due to its graphical nature and therefore ease to spot whether the program does draw what it was supposed to. Also, drawing a concrete shape is possible in many different ways: one can enter the commands to steer the turtle individually or execute a whole bunch of commands at once which we then call a program. There is not only one correct solution but a whole universe of solutions ready to be discovered.

From the perspective of children, programming is a new way to achieve something. With relatively few commands it is possible to draw complex geometrical structures and shapes. There are no boundaries to the child's creative mind. Although, in the end, the goal is to teach the children to independently explore what they can do, they still wish for feedback on their work. From the perspective of the teacher, giving each child individual feedback is a very time consuming aspect. Assessing each individual novice programmers code, see their errors and give valuable feedback uses up the limited time. Oftern it is the case that teachers have never taught programming before and hence lack prior experience. We want teachers to tutor their pupils, praise and support them. But with 25 children, they cannot support everyone at the same time. Weaker children need help while stronger ones want a challenge

1.2 Motivation

The XLogoOnline programming environment is already equipped with a syntax checker so the novice programmers get live feedback while writing code. What is not present is a training area, in which they can solve tasks for themselves and get contextual feedback afterwards. That is what this work proposes: A judge for the XLogo language, that provides tasks to solve and gives feedback for the submitted program.

There are three groups that profit from a judge for XLogoOnline: teachers, students (novice programmers) and us researchers.
Teachers do not only need to teach the children new commands and new ways to think, but also have to keep track of each individual child's progress. So that when a child asks why the pro-

gram does not work correctly, the teacher can asses the problem and provide valuable feedback. The judge supports the teachers in the preparation of exam questions by providing an indication of how easy or difficult a given task is. This indication is based on the comparison of solution paths and time consumed by a large base of students spread across many schools.
Novice programmers get an opportunity to not only test and improve their skills at home with daily changing tasks, but learn to really work autonomously as well. This changes what novice programmers can do to improve their skills - a new platform on which the tasks are given in pictures only and, after a solution is submitted, instant feedback whether the two solutions match graphically. This pool of almost unlimited tasks provides an important resource for training.
Lastly, we as researchers profit from this judge as well. We can evaluate which tasks are easily solved and which pose problems. We can use this knowledge to add features to XLogoOnline to match the demands of the programmers using it, for example by providing special training to common errors (see chapter 6 for an in depth discussion). We get insights on how children solve given tasks. If we collect the data produced by these children who are training with the judge, we improve not only the judge, but also the tasks and ultimately provide even better feedback.

Furthermore, a judge for XLogoOnline opens doors for different follow up projects. We could, for example, provide an separate training area, in which children choose which programming concepts they want to train. A teacher could also use some features of the judge to design a fair and well balanced test in his class. We could, for example, implement a system, in which a teacher activates a task sheet for all of his students and then, in the end, the solutions will be collected and the results visually assembled. The teacher then determines which features he wants to grade and lets the system automatically grade all solutions.

1.3 Goals

In this work, we implement an automatic judge system for the programming language XLogo and integrate it in one of its online development environments, XLogoOnline. An automatic judge is a program that evaluates given solutions without the help of a human, so the end-user (the learners) can use our judge completely autonomously. We implement a judge that evaluates whether a submitted logo program is in the set of possible solutions or not according to a given task.

Intrinsically, XLogoOnline is built in a way that students while programming immediately see, whether their solution is correct or not, thanks to the graphical component of XLogo. But novice programmers and especially children are not always used to working independently and direct feedback (i.e. in the form of praise) is an important factor in the learning process. We want the teachers to tutor their pupils, praise and support them. But with 25 children in the classroom they cannot support everyone. With the help of a judge we want to motivate novice programmers to work independently and motivated in their own speed, while the teachers have time to help the ones in need.

Providing valuable feedback is a very important part in learning. One goal of our judge is to find adequate measures to provide the programmer with feedback on how to improve their skills and ultimately how to program well in accordance to present program paradigms.

What's more, when facing the difficult task to create an exam, teachers do not know how to assemble a set of tasks in a way that is fair in its level of difficulties and adapted to the skills of their students. With the judge, we provide well-founded data form which teachers can create fair exams.

Relevance Adding a judge to XlogoOnline opens many doors and offers great opportunities for further work. Teachers need support in teaching how to program XLogo and in how to design fair exams. For the novice programmers it is also important to know a place where they can test and improve their skills.

1.4 Outline of this paper

This paper consists of six chapters. In this first chapter we introduce the reader to the topic, and provide them with relevant background information.
In the second chapter, the reader can find information about **relevant work**. This chapter covers already present research topics. Here the reader finds a section about the programming language XLogo, a section about one of its programming environments, XLogoOnline and a section about different programming judges used today.
The next two chapters, **Design** and **Implementation**, we examine how the judge presented in this work is designed and implemented. In the chapter about the design we examine the minimal expectations of our judge, what functionality a judge for the programming language XLogo has to offer, what metrics we use to judge a program and near the end of this chapter how we integrate this judge into the already existing XLogoOnline environment.
The following chapter concerns the **implementation**. Here we go deeper into the discussed topics of the previous chapter. We first provide the reader with an overview of our judge and then explain the implementation in more detail. In this part we explain the setup necessary for our judge to run properly, how a task is displayed, how the different metrics are implemented and with which other parts our judge interacts.
After these two chapters follows the **Evaluation** chapter. This is the moment of truth: we evaluate how well our judge fulfills our expectations and what other people, in our case, young novice programmers, think of our judge. We describe the setup of our experiment in the classroom and what changes we did to the judge. For this experiment we asked ourselves what the differences between male and female programmers might be. In the end if this chapter, we discuss the results.
The last chapter addresses two topics: Firstly, we talk about the **conclusion** of our work. Here we shortly describe the impact of this work and then move on to the limitations - the places where we got stuck. Secondly, we offer ways to continue our research. Our judge is, at this time, still incomplete. We include different ways of improving our judge and explain what

follow-up projects can be built on top or with parts of our judge.
This work closes with the **bibliography** and the **declaration of originality**.

2

Related Work

In this chapter, we will have a look at the already existing work, on which we rely and build upon. It starts with a short description of the programming language Logo and one of its development environments, XLogoOnline, then outlines what a judge is and finishes with different (online) judges that are already in use.

2.1 Introduction to the language Logo and one of its dialects

Logo is a programming language with the basic idea of providing a canvas, on which one can draw geometric shapes. The drawing is delegated to a turtle, that "holds" a pencil and therefore leaves a trace as it moves around the canvas. To commandeer the turtle, there are several commands available, for example `fd 100` which will make the turtle go forward for 100 steps (or "pixels")or `rt 90` which will turn the turtle clockwise by an angle of 90 degrees.
An interesting concept is that the programmer needs to comprehend a change of perspective. The commands are always executed relative to the perspective of the turtle rather than the programmer's global perspective. Therefore, when the turtle is facing right, the programmer needs to translate their perspective to that of the turtle and then move the turtle accordingly.

2.1.1 XLogo, a dialect of Logo

Like in a spoken language in which different dialects can occur, the programming language Logo also features several slightly different dialects. In this work, we use a dialect called

XLogo. In the following we provide a list of the core commands of XLogo.

- `fd 100` - forward 100: move the turtle 100 steps in the direction it is facing
- `bk 100` - back 100: move the turtle 100 steps in the opposite direction it is facing
- `rt 90` - right turn: rotate the turtle 90 degrees clockwise
- `lt 90` - left turn: rotate the turtle 90 degrees counterclockwise
- `cs` - clear screen: clear the canvas and reset the turtles position
- `setpc red` - set pen color red: set the color of the pen the turtle is holding (to red)
- `pu` - pen up: the turtle wont draw
- `pd` - pen down: the turtle now draws
- `pe` - pen erase: the turtle erases lines it now moves over
- `ppt` - penpaint: the turtle draws
- `repeat 4 [`*`commands`*`]` : repeat *commands* 4 times

With only a handful of the above commands, a programmer can create a first program that looks like this: `repeat 4 [fd 100 rt 90]`. This program executed will result in the image shown in Figure 2.1.

Xlogo was developed to serve a purely educational purpose. Its advantages are threefold: The simplicity of creating a program, the large solution space and the graphical output providing easy error spotting. Simplicity of creating a program means that even a complete beginner can generate graphical output after a few minutes of receiving some basic knowledge. A very large solution space means that for one task, lets say to draw a square, there are a lot of different correct solutions. The last advantage is that the graphical output provides easy error spotting. If the task is to draw a square, it is easy to see whether all the sides have the same length or one side is missing. The possibility to focus on the development of programs to draw pictures from the beginning makes the process of learning very descriptive, motivating and therefore more attractive.[4]

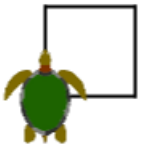


Figure 2.1: *XLogo output*

The Chair of Information Technology and Education at ETH Zurich developed textbooks concerning programming education, as well as a tailor-made programming environment to be used in class: XLogoOnline. We have a closer look at XLogoOnline in the next section.

2.2 XLogoOnline

In this section, we have a look at XLogoOnline, a programming environment for XLogo. We discuss not only the core functionality this environment offers, but also the fundamental details relevant for this work.
XLogoOnline was published in early 2017. It was tailor-made to complement the textbooks published by the chair of information technology and education about XLogo. The motivation of this work was to provide an "intuitive and stable programming environment which is easily accessible from anywhere and which serves the pupils during their learning"[5]. The outcome of this work is a website providing the novice programmers with everything they need to start programming and develop their skills.

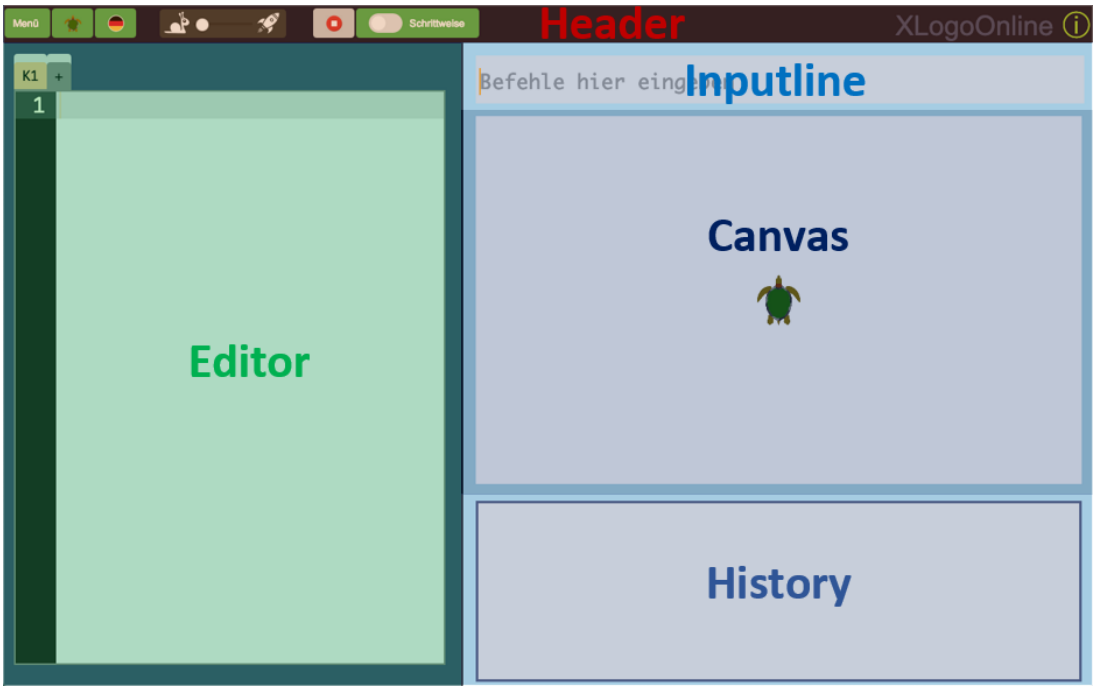


Figure 2.2: *XLogoOnline IDE, v2.8.6, annotated*

Figure 2.2 shows XLogoOnline. It consists of five main parts: Header, Editor, Inputline, Canvas and History. We now briefly explain what these parts do and how they interact with each other.

- **Header** In the Header, the user can access different tools to program with XLogo and the settings of the environment. Here, the user can, for example, find the debugging tool, change the language or save the graphical output.
- **Canvas** In the middle of the canvas is the turtle that executes the commands the user enters in the input line. The turtle leaves a trace in the form of a black line, which can be seen by the user.
- **Inputline** With the commands entered in the Inputline the user controls the turtle on the canvas. When typing commands, the programmers can use any built-in commands or user defined programs that were previously defined in the editor.

- **Editor** To extend the core commands of XLogo, the programmers can define their own commands in the editor.
- **History** At the bottom in the History component, the user can see all previously executed commands as well as errors typed in the Inputline.

For a more thorough explanation of XLogoOnline, we recommend reading the original paper from Jacqueline Staub: xLogo online - a web-based programming IDE for Logo [5].

2.3 The world of programming judges

Feedback is one of the most powerful influences on learning and achievement [3]. In the context of programming, that means getting feedback about a written program. Broadly said, is a judge program that gives feedback about another program. In this section we discuss what different programming judges are present today and discuss reasons for a judge for XLogoOnline.
Judging a program can concern several different aspects: one can reason about correctness, performance, functionality or syntax. All these statements can also be made by an automatic program which we call judge [2].

2.3.1 Each judge has its own purpose

Each judge has its own purpose. Depending on what we want to evaluate in a program, we design a judge differently. Whether we want to test the skills of programmers in competitive programming like the Hackathon in Zurich [1], provide formative feedback for a solved task or even grade a program. For competitive programming, judges typically evaluate correctness, functionality and performance.

Judges for competitive programming: the DOMjudge An example of a known system here is the DOMjudge [2], which is an automated system to run programming contests. It's focus is on usability and security. With the DOMjudge live contests can be held. It has been used in many live contests and has also been adapted for use in numerous online contests or teaching environments.
Although the DOMjudge is open source, for the purpose of our work it has relevant limitations. We can't use parts of the DOMjudge for our own judge due to two main reasons. Firstly, one of the main parts of the DOMjudge is the graphical interface it offers for the contest. Since we already have a graphical interface, we do not need this feature and we do not want to try to decouple the interface from the rest. Secondly, the DOMjudge analyzes programs according to their written output. Because XLogo produces graphical output, we can't use this part of the DOMjudge.

[1] `https://digitalfestival.ch/en/HACK`
[2] `https://www.domjudge.org/`

Judges for exams Another application of a judge is during exams. This is particularly helpful in situations with a lot of people solving the same programming task, since a judge is able to run these programs against testcases and produce feedback. Testcases are necessary for a program to test its functionality. Testcases denote a collection of tests the program needs to pass i.e. give the correct solution to. Having a judge for exams allows for instantaneous evaluation of the performance of a program to each individual student. Not only here, but also when grading the taken exams afterwards, a judge reduces the workload for teachers drastically.
At the ETH, there is a judge present for many computer-based exams, which gives feedback on whether a program compiles, takes too long to excute or how many testcases it passes. The same kind of judge is used in online competitions.

Judges for autonomous training A third category of judges is the autonomous training of programming skills. Websites such as the sphere online judge [3] offer many programming tasks and problems and motivates the users to submit solutions until a given problem is solved. Here, the sole purpose of the judge is to decide whether a submitted solution is correct or not. This website has over 650'000 registered users and over 6'500 problems waiting to be solved. This is the kind of judge we want to add to the XLogoOnline environment.

Each of these three judges serves a different purpose. All three of these judges summatively asses the program, that means they use the judging part as a test in order to give feedback. We want our judge to output a formative assessment of a program and we want to use the judge as an intermediate check to give feedback about a program.

2.3.2 A judge for XlogoOnline

Since its launch in early 2017, the XlogoOnline environment has been constantly improved and new features have been added. Currently, there is a real time syntax checker for input, which provides an informative error message and there is a debugger, which allows program to be executed step by step i.e. one command at a time. At the moment, however, the only way for a novice programmer to train for themselves is be provided with exercises from external sources like teachers or parents. With the proposed judge for XLogoOnline, we add our part to offer an autonomous training opportunity. Our judge adds another feature to the XLogoOnline environment, so that it ultimately provides novice programmers with a way to autonomously program in their own pace.
In the next chapter, we show more information about the interior working of our proposed solution.

[3] `https://www.spoj.com/`

3

Design

In this chapter, we look at the basic design ideas of our judge. We begin by having a look at the design process, here the three steps we highlight are: the high-level overview of our judge, the basic functionality our judge should offer and the more detailed overview. Here we explain what steps the judge goes trough when the user wants feedback for a submitted task. We then present the metrics we want our judge to be able to evaluate and in the end explain how we integrate the designed judge into XLogoOnline.

3.1 Designing a judge for XLogo

With the judge presented in this work we want to tackle the problem that teachers have a hard time assessing every student individually and give adequate feedback. Our judge should support teachers and let them shift their time spent more towards individual support for those children that need it the most. Our judge provides the pupils with concise feedback and thereby fosters their autonomy and capability to resolve issues in their code individually.
In this section we have a look at how to design a judge for the programming language Logo. We first have a look at a high-level overview of the interaction between our judge and the user. Then we explain the basic functionality our judge offers and close the section with a more detailed overview of the steps our judge goes through when evaluating a program submitted by the user.

3.1.1 Interaction partners

The first step in the design process is to identify the environment our judge will end up working with. Here we have two main interactions with our judge: On the one hand, our judge interacts

with the programmers using XLogoOnline and on the other hand, our judge needs to interact with the already existing XLogoOnline environment.

Interaction with the user Our judge needs to interact with the user at least twice: Firstly hen a user wants to solve a task and lets it grade by our judge. The judge then needs to display the task to solve. Secondly, when finished programming, the user submits the created program. The judge now evaluates the program and provides feedback about the program. In figure 3.1, we show this exchange schematically.

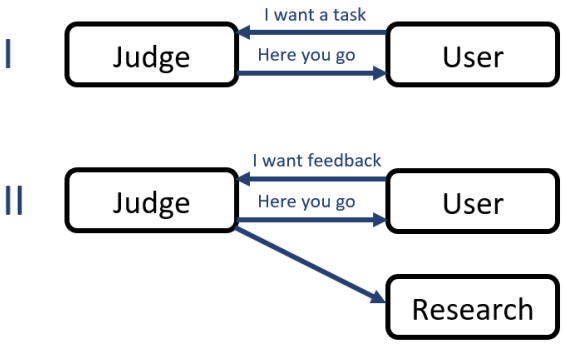


Figure 3.1: *Two interactions between judge and user*

Displaying the task We want to provide the user with an image of the task to solve. This image should be in the same size as the solution the user will generate to make it easier to spot whether the task was solved correctly. That's why we decided to display the task next to the canvas (either vertically or horizontally, depending on the images orientation). Figure 3.2 shows the two possible ways to display the task. In the background of both the reference solution and the drawing area we display a grid with the length of 100 turtle steps in order to make it easier to spot the length of the lines in the reference solution.

Data sent to us researchers One more thing that can be seen in the figure 3.1 is the connection to us researchers. We decided to add this connection in order to be able to collect data about how the tasks were solved and what the difficulties were. We analyze this data to generate new tasks and update the already existing tasks.

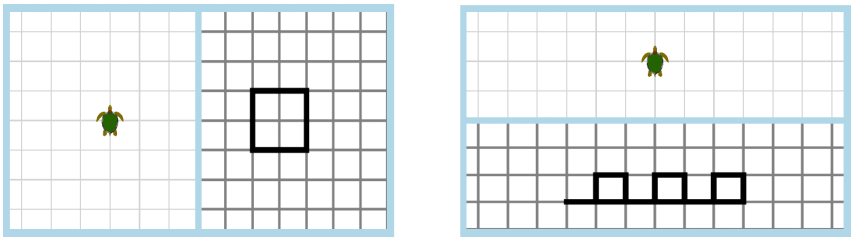


Figure 3.2: *We can display the task either vertically (left) or horizontally (right)*

Interaction with XLogoOnline The other interaction with our judge is through XLogoOnline, where we distinguish two main UI interactions: The first one occurs after the user requests a task, which the judge needs to display on the *Canvas* of XLogoOnline. The second UI interaction occurs after the users have submitted their program, owing to the fact that the judge needs to get not only the whole program generated by the user, but also the graphical output. When our judge finishes evaluating the program, it delivers a feedback for the user via XLogoOnline. We illustrated this exchange in figure 3.3

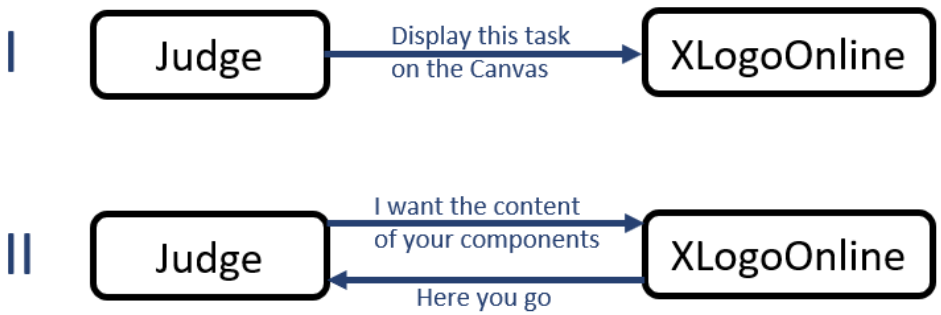


Figure 3.3: *Two interactions between judge and XLogoOnline*

3.1.2 Basic functionality our judge offers

Now that we have identified the basic interaction partners of our judge, we can have a look at the basic functionality we want our judge to offer. We consider the following three aspects of core functionality provided by our judge: a minimalistic interface, correctness and concise feedback. In the following, we will explain these aspects individually.

Minimalistic interface Our judge must not distract the user with its interface. We decided to restrict the ways the user can interact with the judge to two buttons: One for activating the judge and displaying the task and the other to submit a solution and get feedback.

Correctness Since programming is a creative process, each problem statement allows for multiple distinct correct solutions. Although we have not yet explained which metrics we want to use, our judge needs to be able to distinguish whether a submitted program belongs to the infinite space of correct solutions or whether it is part of the infinite space of incorrect solutions. Lets have a closer look at the following example: the task is to draw the stairs shown in subfigure (i) in figure 3.4. Subfigures (ii), (iii) and (iv) belong to the space of incorrect solutions while (v) is one of may correct solutions.

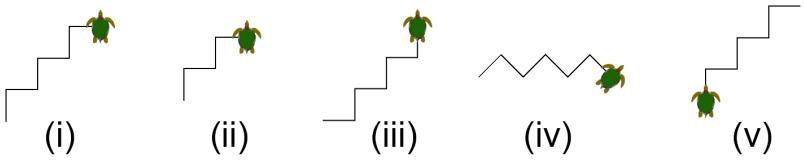


Figure 3.4: *(i): Reference solution, (ii) - (iv): Wrong solutions, (v): Correct solution*

Concise Feedback Lastly, the judge has to give meaningful and relevant feedback. This is important, because since with our feedback we actively influence how novice programmers perceive their skills. For instance, when deducting points for longer solutions, we might intimidate very young programmers who just learned programming. As this is not our intention, we decided to reduce the feedback provided to the user to only whether the submitted solution belongs to the correct solutions space.

Keeping these three functionalities in mind for the rest of this chapter, we can now take a look at the overview of our judge.

3.1.3 Overview of our judge

In this subsection we focus on the main steps that our judge goes through when evaluating a program submitted by the user. First, the user **interacts** with the judge by submitting a solution program, then our judge **preprocesses** the collected data, **evaluates the metrics** and **collects and assembles their feedback**. In the end it **presents the feedback** to the user.

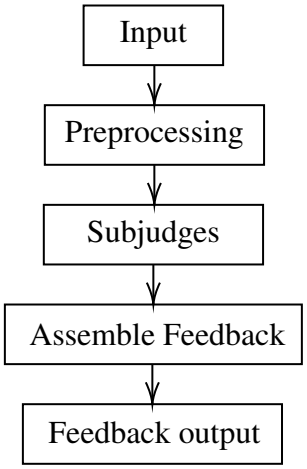


Figure 3.5: *Overview judge*

Interaction with user The user always initiates the execution of the judge. As soon as the user wants to get feedback, the judge gets activated. In order to evaluate a program, our judge first collects the relevant information including the curront content of*Editor*, *History* and *Canvas* (providing all information required to assess how the current picture was created).

Preprocessing The content of *Editor*, *History* and *Canvas* first needs to be preprocessed in order to be used by the subjudges. In this step unnecessary spaces are removed from the *Editor* content and the user submitted graphical output is trimmed to remove white space.

Subjudges A subjudge is one part of the judge that evaluates exactly one metric (all of which will be introduced in the next section). We show an overview of our subjudges' pipeline in figure 3.6. We now explain briefly the individual subjudges and how they are connected. First, a subjudge checks whether the syntax is ok (1). This is followed by a subjudge checking the graphical correctness of the provided visual output: whether the reference solution and the submitted solution match on a pixel basis (2). When the two images do not match, a subjudge (2a) evaluates and highlights the difference. Otherwise, the remaining subjudges evaluate the metrics for the code. These are *Time to develop*, *Length of solution*, *Usage of editor* and *Semantics*. The evaluation of the metrics for code are optional since the judge we want to publish should not restrict the creativity of the children with its feedback. The output of all these subjudges is collected in the next step.

Assemble feedback from subjudges In this step, the evaluation of each subjudge is collected and assembled to be presented to the user. Here we decide what we want to present to the user. At the moment, just the feedback from the subjudge 2 is displayed to the user, providing feedback whether there are graphical differences between the provided solution and the corresponding task.

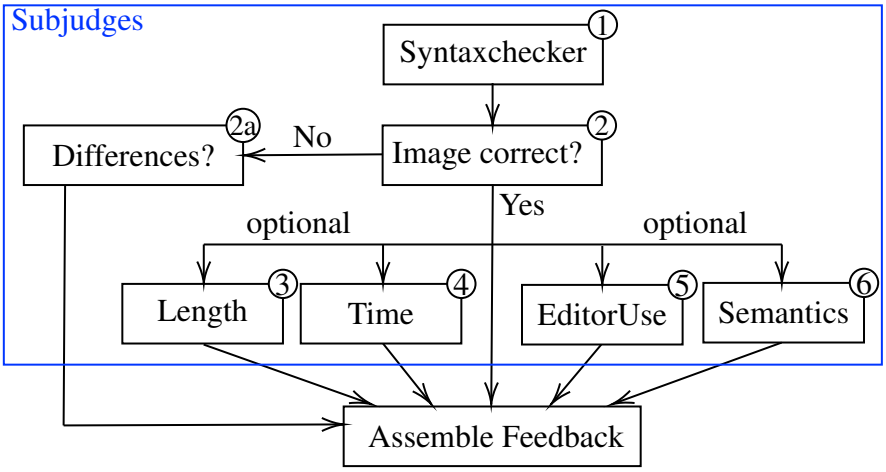


Figure 3.6: *The different subjudges (1-6)*

Present feedback The last step of our judge is to present the feedback to the user. We decided to only give feedback concerning the graphical output: whether the submitted image matches the reference solution. We also added a random praise, something like *"Well done - we knew you would be able to do it"*.

3.2 Metrics

In the main steps the judge goes through, we mentioned different metrics to asses properties of a program. In this section, we have a look at these different metrics, how feasible they are in terms of implementation and producing feedback and whether we want to include them in our judge.

While programming XLogo a programmer generates a program that can be translated into graphical output. Both the written program and graphical output have advantages when evaluating different metrics. Lets take for example the task to produce a square and lets assume the submitted program looks like this: `fd 100 rt 90 fd 100 rt 90 fd 100 rt 90 fd 100 rt 90` and the corresponding graphical output is shown in Figure 3.7. With the graphical output we can definitely decide whether the task was solved correctly what here is the case. This is not possible to decide only by looking at the code and without generating part of the graphical output. Likewise by only evaluating the graphical output we can not determine whether the programmer used the `repeat` intruction, for this to evaluate we need to look at the written code (the programmer here did not use the `repeat` instruction).

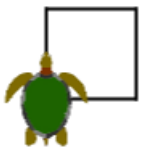


Figure 3.7: *Graphical output*

We now have a look at ways not only to evaluate the written code but also how to assess the generated image.

3.2.1 Metrics for code

There are a lot of things we can evaluate in the written code. Let us again have a look at the previous example where the task was to draw a square, the submitted program was `fd 100 rt 90 fd 100 rt 90 fd 100 rt 90 fd 100 rt 90` what in turn generates the image shown in Figure 3.7. We can have a look at how long it took to fabricate the code (we need more information collected by the judge), we can asses the number of executed instructions (in this case eight) or we can analyze the semantics of a program (no use of `repeat` and modular design). We now list the different metrics for program code and reason about feasibility and whether we want to include them in our assessment strategy.

- **Time to develop:** We want to know, how long the development of the solution took. Here we can either measure the time in minutes or the overall commands used to get to the solution. The idea behind this is to have a comparison between solutions that took a long time to develop and others with less time used. Offering feedback about the time used to develop a solution allows the users to compare to themselves and can awaken the competition spirit (I want to be faster then myself from yesterday). It also offers the users a way to reflect about their own programming style, i.e. when we give feedback about the overall commands used a programmer that normally tries out a lot can decide to think more about the commands before entered.
- **Length of solution:** With this metric, we want to measure the length of the program that generates the final (and correct) output. The difference to the previous metric is that here we only look at the instructions needed to generate the final result and not at all the instructions that were executed. The XLogo command `cs` acts as a resetter for the canvas and with this metric we only count the instructions after the last canvas reset. Feedback about the length of the solution allows us to award the programmers that write shorter code and use for example repeat statements.
  We can either use this metric relatively or absolute. Relative means we compare the length to the length of other solutions submitted, absolute means we award more points for a shorter solution. We decided to use it relatively, because we want the programmer to be compared to other programmers that solved the same task. Therefore we need to store the length of the shortest program submitted by anyone that solved this task globally. We will not include this metric in our final assessment strategy because we need to carefully evaluate if the feedback is actually helping the novice programmers.
- **Usage of editor:** In XLogoOnline, one can enter the commands one at a time in the input line or define a program in the editor or a combination of both. This metric measures whether a pupil made use of the editor. Since one of the core ideas of the programming language XLogo is to use modular design, we include this metric in our assessment strategy.
- **Semantics:** Evaluating a programs semantics offers the most individual feedback but is at the same time the hardest to implement. The difficulty is that we want novice programmers to algorithmically solve problems but still teach them about programming paradigms like how to use `repeat`. In example, we want programmers to rather solve the task to draw a square like this: `repeat 4 [fd 100 rt 90]` than like this: `fd 100 rt`

`90 fd 100 rt 90 fd 100 rt 90 fd 100 rt 90`. But distinguishing good from bad coding style is not easy, because there are many different solutions for one problem using different programming paradigms. That's why we do not include this metric in our final assessment strategy. We discuss this issues more thoroughly in the chapter Conclusions and further work.

3.2.2 Metrics for images

Having discussed the different metrics for written code, we next look at metrics for evaluating the graphical output. This is necessary since, as already mentioned before, it is not possible to check whether the submitted program translates into the correct image or what the differences are without the submitted image. Our judge presents the task to solve as a picture so that the programmers can compare whether their solution matches the given task. Basically, there are two things we can look at when comparing the reference solution and the submitted image. Whether they are identical (if the submitted image is correct) and what the differences are, if any.

- **Correct solution:** While comparing two pictures and decide whether they match is quite easy to do as a human, the computer needs some constraints to be able to correctly decide whether the submitted image matches the reference solution. The basic idea here is to compare the two pictures on a pixel basis. One thing to point out is that we have to make sure that for example these two programs both count as correct solutions to the task to generate a square. Solution 1: `repeat 4 [fd 100 rt 90]`. Solution 2: `repeat 4 [bk 100 lt 90]`. Although both programs generate a square, the first one will draw upwards and the second one draws downwards. Nevertheless both solutions are correct. We will have a closer look at this in the chapter about the implementation. We add this metric to our assessment strategy because we want to provide the user with this fundamental feedback.
- **Differences:** If two images do not exactly match, this metric highlights the places in the picture that are not the same. Since the user already knows whether the reference solution and the submitted image match with the previous metric, this metric is added to support the user to find errors. The idea is to subtract the pictures from each other (mathematically speaking) and then highlight the places where the two pictures are not the same.

It is important to point out, that the following metrics apply to a general judge for the XLogoOnline environment. The problem this work tackles in the end is to support teachers and help novice programmers - the judge resulting from analyzing this problem will not give feedback to any of these metrics except if the solution is correct. We want novice programmers to experiment and have fun while the program paradigms we teach them in our schoolbooks should be taken into account secondly. As we discuss in the last chapter: Conclusion and Future Work, we need to adjust the assessment strategy according to the prior knowledge of the programmer using the judge. Therefore we propose that the programmers can choose for themselves what kind of feedback they want and include in the final assessment strategy only whether the task was solved correctly, based on the graphical output.

3.3 Integration into XLogoOnline

In this section, we explain how we integrated the proposed judge into XLogoOnline programming environment. First, we will have a look at the structure of the XLogoOnline environment, before we explain how we integrated our judge into XLogoOnline.

3.3.1 Overview design XLogoOnline

The XLogoOnline environment consists of four user interface components. The editor, the inputline, the canvas and the history window. Each of these windows contains a seperate part of the overall programming environment and suits a different functionality. We now explain these four components, why they are there, what their functionality is and in which way the respective component is relevant to our judge.

- **Editor:** The *Editor* contains user-defined programs. Here they can define new custom commands and thereby enrich the XLogo programming language and thus their individual toolset to solve problems.
- **Inputline:** The *Inputline* is used to send the commands or programs to the turtle, which then will be interpreted and cause a visual output on the *Canvas*. The users can use either the core commands of XLogo or the commands they defined for themselves in the *editor*.
- **Rendering:** The *Rendering* contains a rendering of the drawing of the turtle. The *Rendering* changes when the user enters a command in the *Inputline*. This is the output of the program the users programmed and therefor also where the users see whether they programmed correctly.
- **History / Error:** This component serves two purposes: First it shows all previously executed commands, thereby offering a *History* of the current session. Secondly it shows error messages to syntactically incorrect programs when the user tries to execute an unrecognized command in the input line.

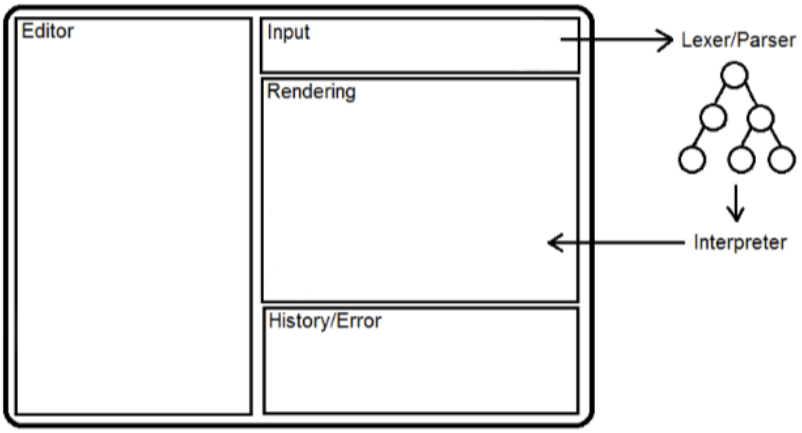


Figure 3.8: *Overview of XLogoOnline*

3.3.2 Adding the judge to XLogoOnline

The judge interacts twice with components of XLogoOnline: Once to display the task when requested and once to assess a submitted program. Since both times the user triggers this event, we added the judge as an additional component to XLogoOnline and let it react to user events. In the first event, the judge responds with a picture of the task to draw. When the users submit their solution, we bundle all relevant information, namely the content of the *editor*, the *rendering* and the *history*, and pass it on to the preprocessing part of our judge. In figure 3.9, we see such a bundle: in green all text-related content is shown, while all picture-based content is colored in blue

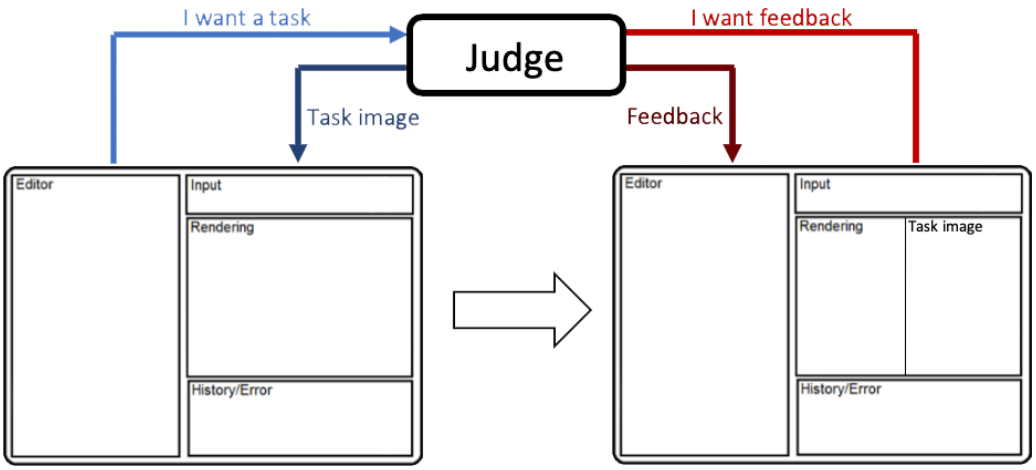


Figure 3.9: *Overview of XLogoOnline with the Judge included*

In this chapter we looked at the basic design of our judge. We looked at the judge itself, which steps it goes through when evaluating submitted program and its basic functionality. Then we discussed the different metrics we use to asses a submitted program and in the end explained how we integrated the judge into XLogoOnline. In the next chapter we are going to to look at how we implement the design in detail.

4

Implementation

In the last chapter we saw the general design idea behind our judge. In this chapter we are going to explain how we implemented these design choices. We first give an overview of how the functionality is provided by the judge when it setups and then interacts with the user and the XLogoOnline base system. Afterwards, we have deeper look at the individual parts and how they function. Here we explain the timing issues when drawing a task, the metrics, reasoning about correctness and in the end how we made sure the judge displays the tasks correctly independent of the screen resolution.

4.1 Overview

In this section we have a closer look at when our judge interacts with the fundamental XLogoOnline base system and the user. Before these interactions, the judge needs to setup (i). Afterwards these interactions happen in two ways: (ii) When the user requests a task and (iii) to receive feedback. In figure 4.1 we can see the general overview of our judge. Let's have a look at the setup as well as the two use cases.

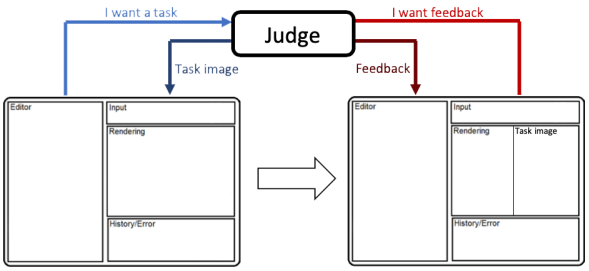


Figure 4.1: *Overview of XLogoOnline with the Judge included*

i. **Setup** The setup consists of asking the server to provide a task. This task is the same for everyone at this day using XLogoOnline and changes daily, giving the users every day a new challenge to solve. Our judge receives the task as a logo program of the form `to`

`solution \n repeat 4 [fd 100 rt 90] \n end` *('\n' denotes the newline character).*

ii. **Get a task** Upon starting the contest mode, two steps are executed in a row. First, all counters measurements to be taken for the different metrics are initialized (e.g. starting a timer to be able to give feedback about the development time). Then the judge draws the task on the *Canvas*. In the section about timing and drawing the graphical output of a taskwe will discuss in more detail how this is achieved.

iii. **Receive Feedback:** As soon as a users submit a solution, they request feedback. This triggers the steps described in chapter three about overview, is shown in figure [figure V] and results in feedback to be displayed to the user.
We want to stress here that when working with children, we want to encourage them to try out different ways to solve a task. That's why the judge for these young children will not output any feedback concerning their code except whether they solved the task correctly and the image they drew matches the image of the reference solution.

4.2 Implementation of the judge

In this section, we have a look at how we implemented the different mechanisms provided by our judge. This includes the setup of the judge, how to draw the image of the task, how the different metrics work and it's interaction with the *Rendering*.

4.2.1 Timing and drawing the graphical output of a task

To draw a task, a sequence of steps is necessary. The problematic part here is timing. Drawing on the *Canvas* takes time. This is since we parse and interpret code that may produce thousands of lines on the *Canvas*. And that parser is executing the code in a step by step fashion doesn't inform us when it is done. In order to solve this issue, the parser must be redesigned - we have a closer look at this in the last chapter of this work: Conclusion and Future work.
We need to wait until the parser has finished executing code and the picture is finished drawing, because we want to store the drawn picture to get a screen specific reference solution. We discuss the reason for this in the subsection about independence from the screen. At this moment, we use hardcoded timeouts, that allows a program to be executed not earlier than a specified number of seconds. We follow with a list of the steps that are executed. In order to get the desired images, it is crucial to only continue to the next step if the previous step has completely finished. In Figure 4.2 we illustrate the following process showing only the *Canvas* component of XLogoOnline.

1. In the setup of the judge, we get a logo program representing the task to solve. In the first step, we let XLogoOnline draw the solution. This is the reference solution we will use to compare the submitted image to.

2. We store the picture as an array of pixels, so that we later can compare it to the submitted solution. We then clear the *Canvas* (2a) and draw the solution again , this time with the

reference grid the children need to infer the length of the lines on the reference solution (2b).

3. We temporarily store the picture of the reference solution, this time with a background grid and again reset the *Canvas*.
4. In the last step we draw the temporarily stored picture from the previous step on the canvas, depending on its orientation either horizontally or vertically (in figure 4.2 vertically).

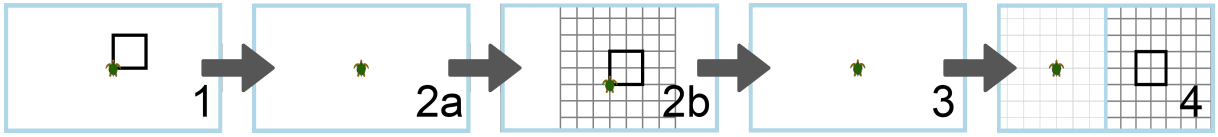


Figure 4.2: *Steps of drawing the reference solution*

Orientation and grid Depending on the orientation of the task, we draw the task differently. There are two possibilities: vertically or horizontally. Either way, we split the *Canvas* window in half and draw the task in one half with the other half offering space to solve the task and draw. In Figure 4.3 We also adjust the position of the turtle to be in the middle of the now smaller drawing window. To help the programmer to draw the provided task, we add a grid to both the solution and drawing area. The length of one of these grid-cells is 100 turtle steps.

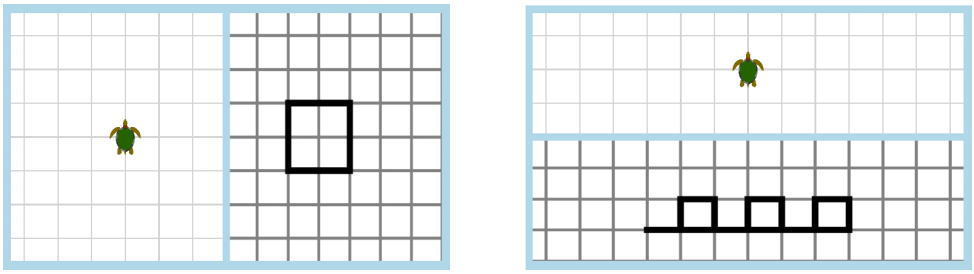


Figure 4.3: *Vertical and horizontal orientation*

4.2.2 Metrics

In this subsection we have a look at the discussed metrics. We list their mechanism, their functional components and connection to other parts of the XLogoOnline environment. We include only the metrics found feasible in the previous chapter and explain the metrics for the program code and for the picture that need more details about the implementation.

- **Time to develop:** In this metric, we are looking at both the time taken to develop a solution in seconds or the total number of instructions used to generate a final image. For the number of instructions used to generate a solution, we do not count the individual statements but the number of times the user pressed enter. This corresponds to the number of lines in the *History* component.

- **Length of solution:** With the length of the solution metric we want to analyze how many instructions were used to draw the final image. We use the content of the *History* component, but only count the instructions after the last `cs` i.e. all instructions that were actually issued so far and contribute to the current drawing. To explain how we deal with the use of user-defined programs, we calculate the length of the solution for the example shown in Figure 4.4. The first thing we do is identify the last written `cs` instruction and delete all issued commands before it. This leaves us with `repeat 3 [rt 90 fd 100 lt 90 square rt 90 fd 100 lt 90]` as the content of the history and `to square \n repeat 4 [fd 100 rt 90] \n end`. Now we identify the names and length of all programs in the *Editor* in this case there is only one, its name is `square` and its length eight. To unfold a `repeat` we multiply the number after `repeat` and multiply this by the number of instructions in the square brackets, in this case two. When in such a user-defined command there is again a call to a user-defined command, we deal with it the same way we did for the commands in the *History*. At last, we need to unfold the `repeat` instructions in the *history*, what we already explained. In the example, this results in 3 times (6 + 8), what evaluates to 42.

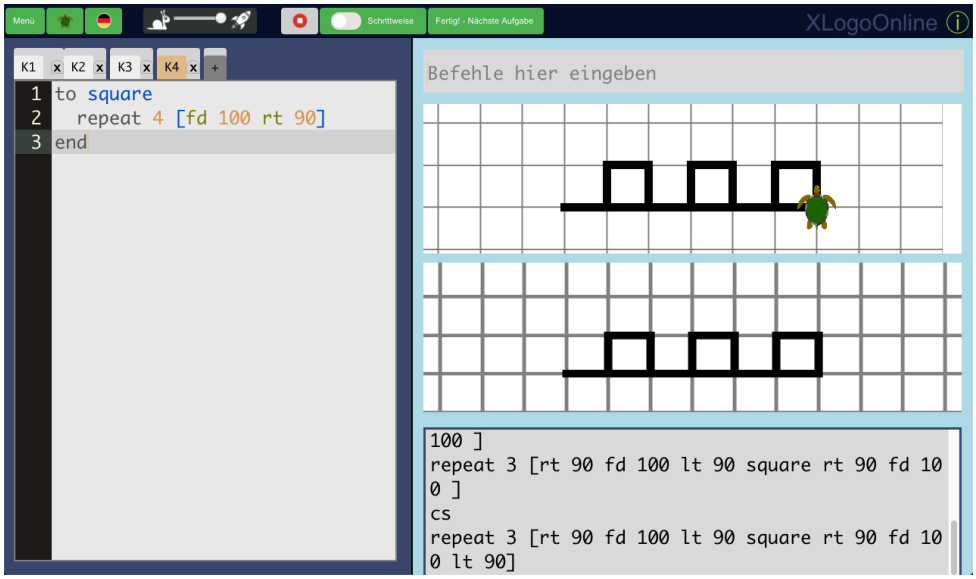


Figure 4.4: *Example of length of solution*

- **Correct solution:** We have a more detailed look at this metric in the next subsection: Correctness.
- **Differences:** This metric highlights the differences in two images. In order to find the differences in two pictures, we use the mechanism already described in the previous metric: the correct solution. Then we subtract the two pictures from each other on a pixel basis. We do this the following way: If two pixels match, we set the corresponding pixel in the submitted picture to 50 percent black. If however two pixels do not match, we color the corresponding pixel in the submitted image red. This results in a picture, that still contains the original image (in gray) and at the same time highlights the parts that do not match. We then show this image to the programmer. Figure 4.5 shows the resulting difference image (right) from the reference solution (left) and the submitted

solution (middle).

Figure 4.5: *Left: reference solution, Middle: submitted solution, right: Differences highlighted in red*

4.2.3 Correctness

In this subsection we discuss the correctness property of our judge. Programming is a creative process of finding a solution to a given problem. There are a lot of different programs that correctly solve a given problem. Our judge has to be able to decide definitely, whether a submitted solution belongs to the space of correct solutions for the reference task or whether it belongs to the space of incorrect solutions. This decision is based on the graphical output of the program. Due to this several problems arisen during implementing this metric:
To decide whether a task was solved correctly, both of the images need to exactly match. This is implemented by comparing the two images on a pixel base. In Figure 4.6 we show the code that compare two pictures (im1 and im2), given as pixel arrays, exactly match.

```
private picComp(im1: ImageData, im2: ImageData): boolean {
  //Compares two given pictures (given as Arrays with 4 values per pixel)
  //return true if they match, false otherwise
  if(im1.height != im2.height || im1.width != im2.width){
    //check if height / width matches, return false otherwise
    return false;
  }
  let imDat1 = im1.data;
  let imDat2 = im2.data;
  for(var i = 0; i < imDat1.length; i++){
    if(imDat1[i] != imDat2[i]){
      return false;
    }
  }
  return true;
}
```

Figure 4.6: *Code example: Do two images match on a pixel basis?*

Since we compare the two images need to match on a pixel basis, we have to make sure the are stored the same way. In example, the position of the image in the *Canvas* must not matter. To counter this issue, we save both images by looking for the pixels in the canvas that are not colored the same as the background and then crop the image to the points (minX, minY), (minX,maxY), (maxX, minY) and (maxX,maxY), where minX is the pixel different to the background with the lowest x coordinate. MinY, maxX and maxY are recovered the same way.

The last issue with correctness is, that since we compare the images on a pixel basis and most of computer screens have different resolutions, we can not use a already generated reference solution. That's why we draw the reference solution on every screen - i.e. every time the user requests a tasks to solve.

4.2.4 Independence from screen

In this subsection we explain how we made sure that our judge works for every screen resolution. As we already have discussed the generation of the reference solution image, we here have a look at the displaying of the reference solution and the reference grid. We first have a look at how to generate a fitting reference grid, before we explain how to place the image of the reference solution in the middle of it.
To create a fitting reference grid, we first calculate how many grid cells fit in the resolution dependent *Canvas* window horizontally and vertically. After checking the orientation of the reference solution (which dimension, width or height, is larger) we then generate a grid that is either half the number of grid-cells fitting vertically and the number of grid-cells fitting horizontally or vice versa. For a perfect square image we display it vertically. We then also place the turtle in the middle of the remaining *Canvas*. See Figure 4.7 for an illustration of the three possibilities. As the last step, we now place the reference solution in the middle of this grid.

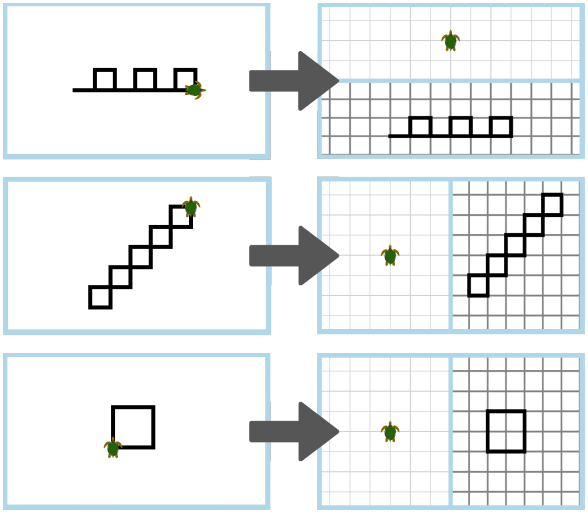


Figure 4.7: *The different ways to display a task*

We explained all the details necessary to reproduce the code for a judge. We had a look at an overview of the judge and how it interacts with the already existing system. Then we had a look at how we met the requirements followed by the implementation of the mechanisms in the judge itself. This included its initialization, the drawing of an image of a task the metrics used for the judge and finished with the interaction with the *Rendering* part.
In the next section, we look at how we evaluated the judge.

5

Evaluation

In this chapter we present our findings from an evaluation of our judge with the children . Firstly, we discuss briefly whether it provides the basic functionality as seen in chapter three. Secondly, we look at an evaluation we performed with 74 children from four classes. Here we posed the question, whether we would find differences between boys and girls when competing in programming tasks. The questions we will answer cover the following comparing points: solved tasks, length of history, number of tries and hard tasks. Before that, we have a close look at what the setup was for the children and our judge, which tasks we let them solve, how the children performed and how they liked it.

5.1 Basic functionality

In this section we evaluate whether our judge meets our expectations discussed in chapter three. To recap, there are three core requirements: firstly, our judge needs to provide a minimal interface. Secondly our judge needs, at the very least, to be able to decide whether a submitted program belongs to the space of correct solutions for the given problem or to the space of incorrect ones. And thirdly, our judge needs to give concise feedback to the user.
We now have a look at these three functionalities separately.

Minimalistic interface This functionality insures that the interface of the judge to the user is as minimal as possible. We implemented this functionality with two buttons: The user clicks one button to get a task displayed and the other to submit a solution and receive feedback.

Correctness By providing this basic correctness functionality, the judge always accepts a correct solution. The metric we chose is whether the image provided and the user-drawn image are identical on a pixel basis. We encountered a few cases, while testing with children, in which correct solutions were not accepted. We are not sure, why this issue popped up and we will have a discussion about it in chapter six: Conclusion and future work.

Concise feedback Our judge need to be able to provide concise feedback while testing and evaluating the different metrics. In the setup used for the following experiment, we choose to only give feedback about whether the task was solved correctly. This feedback is concise and gives the users helpful feedback whether they solved the task correctly.

To sum it up, our judge provides two of the three required basic functionalities. We will continue to work on the judge after this work is finished.

5.2 Evaluation in the classroom

We had the chance to test our judge with an in-class experiment. We prepared our judge to be able to catch information about the following metrics: The solved tasks, the length of the history, the number of tries and the hardness of tasks. All these metrics serve our first goal to find out what differences boys and girls pose in programming. We had the luck to be able to test our judge with almost 100 children aged between 10 and 12. The second goal was to find out whether our judge is working correctly and what its impact can be when it is used by children. In this section, we first explain the setup of this experiment, then what changes we made to our judge. Then follows a detailed discussion about the evaluation of the metrics we let our judge assess.

5.2.1 Setup

In order to test our judge, we made several changes to it and the XLogoOnline environment. We prepared a contest with set of 14 tasks that we let the children solve in half an hour. The order of these tasks was the same for everyone. To be able to get to the next task, the current task needs to be solved correctly. The judge then decides, whether a task was solved correctly or not. We collected all submitted data in the background anonymized. We used a questionnaire to correlate the stored data and basic information about the programmers.
During the contest, we would not answer any questions to have fair conditions for everyone.

The children In total, 74 children took part in our experiment. 36 of them are girls, 38 are boys and one of them didn't provide us with an answer. The children are aged between 10 and 12, with the most (about three quarters) at the age of 11 years. The children are split up in four classes: three five grade classes and one sixth grade (in which there are ten boys and ten girls). They all at least once programmed with XlogoOnline. Before we started the contest, we refreshed their memories and gave them a short lesson about the core commands of XLogo.

5.2.2 Judge used for testing:

We changed our judge used in this setup to the following setup: Our judge had a set of tasks present that it would display in a fixed order. We didn't want the children to run out of tasks, so we arranged 14 tasks and ordered them by difficulty. To advance to the next task, the children needed to solve the previous task correctly. Whether a task was solved correctly or not was decided by our judge based on the correctness metrics. After half an hour, the contest was over and the judge displayed the number of solved tasks per child. To let the children finish their thoughts when the 30 minutes were over, our judge would let them have one more try to submit a solution after the clock ran out.
In the background, our judge collected and stored all data concerning the metrics we now explain further:

- *Picture match*: Is the task correctly solved?
- *Number of solved tasks*: How many tasks were solved in the given time.
- *Time of development*: For this, we used the time in minutes as well as all issued instructions present in the history.
- *Times tried to submit*: How many times was a wrong solution submitted before a correct one?
- *Used Editor:* Were new user-built commands used to solve the task?

5.2.3 The tasksheet and questionnaire

We wanted the children to never run out of tasks. That's why we increased the difficulty with each task and included as many as 14 tasks for the children to solve. In Figure 5.1 we show the tasks. They are numbered so that we can correlate them to the plots later in this chapter.
After the experiment in the first class, we had to make some minor adjustments to the order of the task. Now task number ten was beforehand task number seven. This task was On the questionnaire, the children first had to fill out some basic information about them (Age, Gender, whether they program at home) and then they had to mark the tasks they found the most difficult (of the ones they solved). To correlate these questionnaires and the data collected by the judge, we generated for each child a unique four digit code they had to write down on the questionnaire.

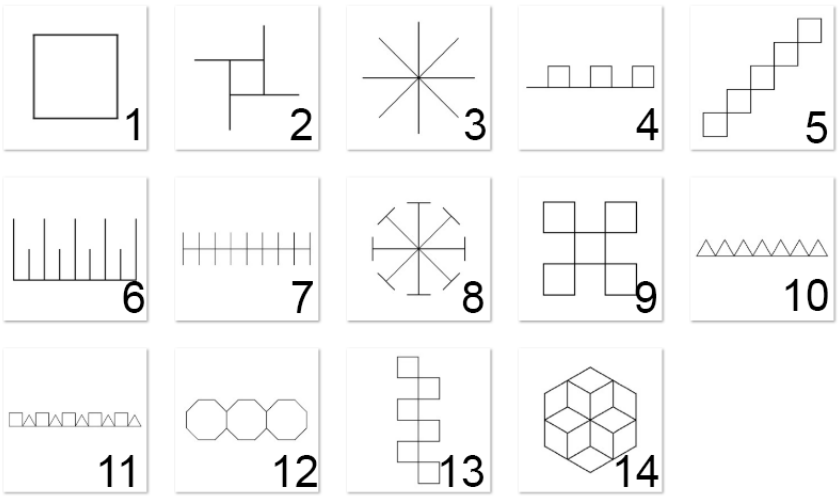


Figure 5.1: *The taskset the children had to solve*

5.2.4 Differences in programs between boys and girls

What is the difference between programs of boys and girls? Do they approach a problem differently? Is the structure of their work different? How do they cope with failures and how do they react when facing difficulties?
To answer these questions, we collected data in the background as the pupils submitted their solutions and we let them fill out a questionnaire on which we would get basic demographic information like gender, whether they program at home or not and which tasks they experienced especially difficult. To be able to match the data with the questionnaire and to anonymize their data, we used a anonymous 4 digit code.

Where our questions come from Whenever Prof. Dr. Juraj Hromkovič is talking about programming with children, he mentions clear differences between how boys and girls do program. Boys solve a task more exploratively (learning by doing) while girls avoid situations the are not confident (think before acting).With the metrics we presented beforehand, we want to further support this hypothesis and gather additional information, how we can recognize this in programs.

5.2.5 Exploration vs planning

In this subsection, we analyze our findings and show that there is a difference in how boys and girls program. We explain, how we spot that boys love to explore and try out different things and that girls take time to plan before doing anything. For each metric, we look at the results and discuss what that could mean and what the implications are.

Solved tasks To begin with, we have a look at how many tasks were solved. Both boys and girls were able to solve nearly the same number of tasks in the same time. Although we changed

the order of tasks, this does falsify the following graphs since in the class we used a different order of the tasks there were as many boys as girls. The two leftmost figures in figure 5.2 show how many boys and girls could solve the task indicated on the x axis. When we compare the two plots we can see no big differences. The figure on the right confirms this: it shows how many tasks on average were solved. Since there are almost as many girls as boys (36 girls and 38 boys), we can stress the importance of this first finding: If boys and girls have a different strategy to solve tasks, neither strategy is better or worse.

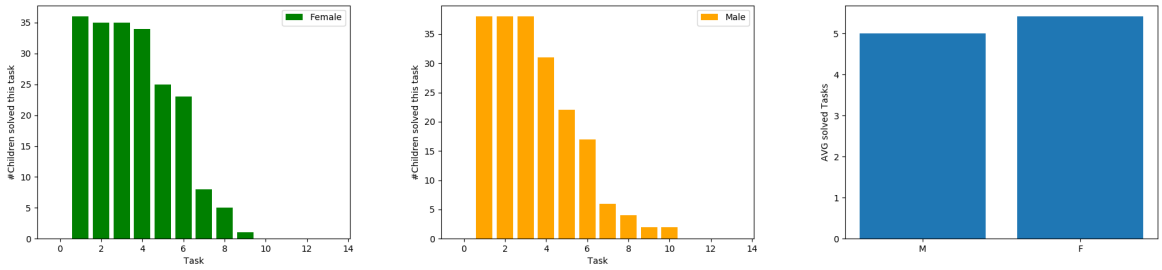


Figure 5.2: *Left and middle: How many solved a task, Right: The average of solved tasks for boys and girls*

Length of history This second metric is about behavior: How do the children work and how do they solve a task. This metric measures the length of the history. This includes all the commands that were executed for one task. Our hypotheses here is that since boys explore more by trying out and girls more think about the commands they issue, we expect the boys to have more commands in the history and the girls having overall longer commands issued in the inputline. What we found out is that there is almost no difference between the boys and girls when comparing the average length of the history of issued commands.
On the right side in figure 5.3 we see the average length of one command in the history per task. One command is the collection of instructions entered at one time into the inputline. The length denotes the number of different instructions, such as `fd 100` or `rt 90` (both counting as one command). We again see, that there is no big difference between boys and girls. We conclude, that there are no significant differences between how many commands boys and girls issue while programming.

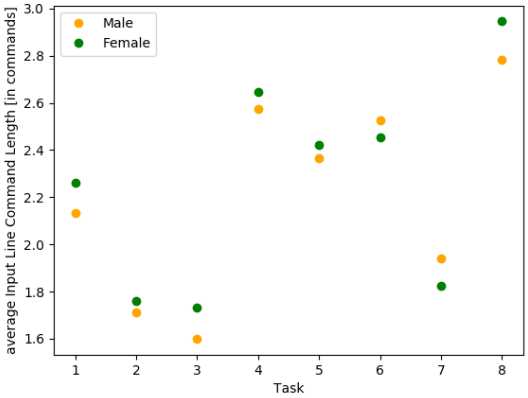


Figure 5.3: *The average length of one command in the history*

Tries The next metric we look at is the number of times a child pressed the submit button with an incorrect solution. According to our hypotheses, we expect boys to have a significantly higher average of submit button presses, because they more often explore and try out. In figure 5.4 we see, that the boys on average pressed the submit button one time more than the girls. This corresponds to our hypotheses that boys more often want explore to see what happens. Girls, on the other hand, think and plan more about their actions.

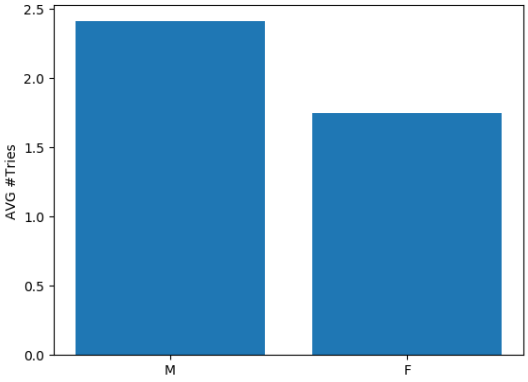


Figure 5.4: *How many times a programmer tried to submit a solution per task on average*

CS-Score This metric analyzes the number of `cs` instructions used. The `cs` instruction is used to reset the canvas and inserted after a wrong instruction. The cs-Score counts how often the children started over and issued `cs`. In relation to our hypotheses and the previous metric, we expect boys to more often use this instruction than girls. In figure 5.5 we see the average number of `cs` instructions used for a task in blue. Here we see that boys (orange) and girls (green) use almost the same amount of `cs` instructions, except in the tasks 4, 8 and 9, in which boys use more `cs` instructions. This shows, that for some tasks, boys try out more, but not for all tasks. Therefore we can not really confirm out hypotheses with this metric.

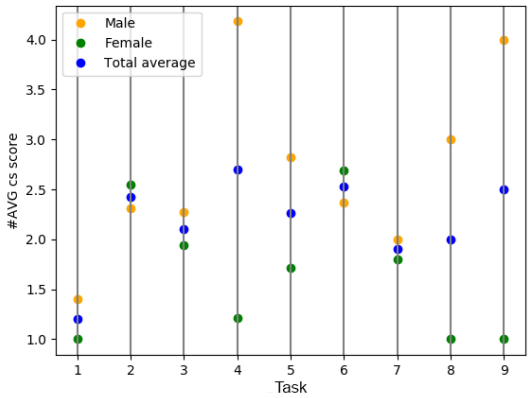


Figure 5.5: *The average used cs per task*

The difficult tasks Each child was given a questionnaire on which they should indicate which tasks they judged the most difficult out of the tasks they solved. While the children programmed, we stopped for each task the time they used to solve the task. Our hypotheses in this case is, that (a) there should be a correlation between the difficulty of the task indicated by the children and the time it took them to solve the task. For example, when a task is mentioned as hard by most of the children, we should also see a high number in the graph showing the time used on average. In figure 5.6 we see horizontal pairs of graphs: on the left we show how many percent of the children that solved this task indicated that this was a difficult one and on the right we show the average time it took the children to solve this task (on the bottom of the bars we show the standard deviation of this bar). For task number five, this hypotheses concludes: more than 80 percent of all girls that solved the task indicated it as hard and in turn it took them on average almost a minute longer to solve it. On the other hand, although only 20 percent of the boys that solved task number seven indicated it was difficult, it took them almost the same amount of the time the girls needed. (More than 50 percent of the girls find the task difficult). Does that mean girls better estimate their abilities and boys sometimes overestimate themselves? It could be an indication, but there is only weak evidence. The last pair of graphs is similar to the first one, only that now both girls and boys estimated their skills correctly (we want to stress here the low deviation).
Summarizing the findings only leads us to the conclusion, that we need to further evaluate the data we have to validate the claims made here.

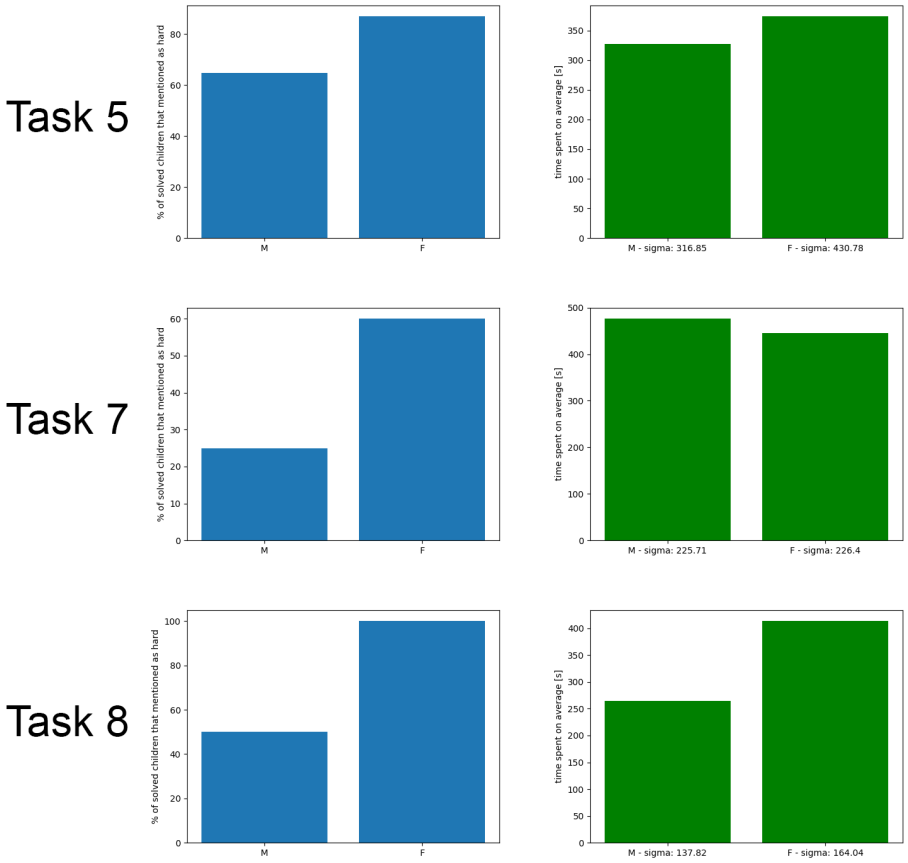


Figure 5.6: *Left: How many programmers indicated this task hard, right: How much time was spent on average on the task*

5.3 Conclusion

5.3.1 Discussion of findings

Research is not easy. In the best case, we deploy a hypotheses which then gets validated with data from experiments. In the worst case, the opposite happens and our hypotheses gets teared apart. But with the findings presented in this work, neither is the case. Our hypotheses stands a bit shaky in between these two extremes. There was some evidence, but also there was no evidence at places there should have been. The question now is, what does that mean? In our opinion, this is actually the best case that can happen in research, because it motivates further research. It is now not possible to jump to wrong conclusions, because there is only slight evidence. This in terms mean, that we have to carefully check whether the connection of two metrics is statistically stable or just coincidence. Also, we definitely need more data than just four classes.
We are sure, there is a difference in the way boys and girls approach and solve problems. We saw some evidence, but not enough to make a clear statement. We will continue research in this direction and hopfefully find out more to support our hypotheses.

5.3.2 Bugs, performance and feedback about the judge

While testing the judge in class, we encountered some minor bugs and one major bug. The minor bugs included render errors, internal program errors and internet connection issues that mostly could be solved by restarting the program. The major bug we encountered was that in a few cases the judge would not accept correct solutions. This happened most of the time when the children submitted their solutions for task number four (see figure 5.1). Even when we tried out the master solution, the judge would refuse to accept the solution at correct. This bug happened only on one of the two laptop models present in this class, so our fix was to hand out new laptops, on which we solved the previous tasks so that the children could continue. Strangely, on laptops that previously would show this bug, later on the judge would function without errors. We were not able to reproduce this bug on our own computers.
Concerning the performance, we did not have any issues. Collecting the data and displaying the tasks was done smoothly and without any problems by our judge.
The feedback was well understood by the children. We also had no problems in how the children understood the feedback, except the most of them were surprised when time was up, although there was a pop-up stating exactly this.

5.3.3 A collection of all the cool things

During the experiment, we experienced a lot of awesome things. In this subsection, we want to share our highlights with the reader.
Over the course of the whole experiment, all children were so excited. We could see and feel pure emotions and energy. When we told them that they are the first children in Switzerland that have the opportunity to test our judge, we experienced a small riot full uf excitement and anticipation. Almost the same happened when we asked them if they spot something new in the XLogoOnline environment (there was a button indicating a contest-mode). And when we a few days later evaluated the questionnaires we saw a lot of cute remarks on them. Most of the children said thank you and a few even praised us for our work.
After the individual experiments we asked a lot of teachers and pupils whether they would use the judge if it was available online. The following pie charts in Figure 5.7 show the results of this question. All teachers we asked answered yes and approximately four fifth of the pupils told us they would use the judge, if it is present online, at least once.

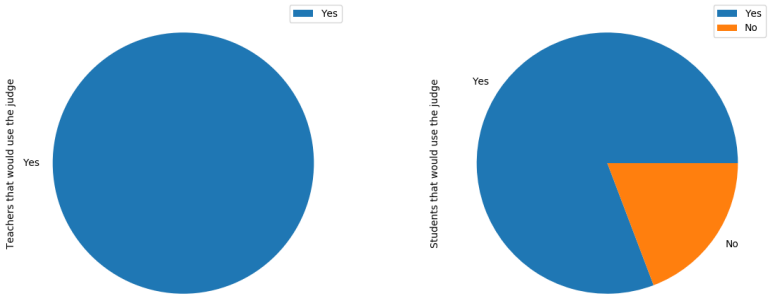


Figure 5.7: *Number of people that would use the judge*

One last thing that just stunned us was that before the beginning of each lesson, when the children slowly assembled in the classroom, every single one of them would come to us shake our hands and say hello. We think, we should do that more often. Without judging, welcoming another human.

6

Conclusion And Future Work

We have reached the last chapter of this work. We now take a step back and think about the implications of our work. Which doors does it open? Where and what are the limitations in our work. Are there any disadvantages in our approach to the problem we wanted to tackle in this work? In which directions can future work continue? In this chapter we are going to look at answers to these questions.

6.1 Impact

While doing research for a particular subject, it is easy to loose the overview. That's why in this section we take a step back and look at the implications of our work. Where does it make life simple? Which doors does it open? Why is our system cool?
In the introduction section we explained, what part teachers play in teaching young novice programmers how to learn to program. There we saw that it is not easy to generate meaningful feedback for each individual student and that this task is the most time consuming one. That's also a motivational factor behind this work: to support the teachers that teach how to program.
With the judge presented in this work, we definitely created an important supporting tool for teachers. The judge supports the teacher in the process of assessing programs of individual students and provides the students not only with exercise but also feedback about the program they submitted.

6.2 Limitations

Our judge is not perfect. In this section we have look at where we could not advance and encountered problems we could not solve. Also we discuss the metrics that we wanted to implement but were dropped due to the lack of time. Important to notice here is that the feedback of our final judge does not contain any feedback concerning these metrics because we want novice programmers to try out different programming styles and think as creatively as possible.

6.2.1 Drawing the task

One issue we encountered and were not able to resolve concerns the procedure of drawing the task for display. When a user clicks on the 'Contest-Mode' button, our judge will first draw the task to solve on the canvas in order to save it for later use in comparing the submitted program to it. With this, our judge decides whether a task was solved correctly. Afterwards the judge draws the task again, this time with a background grid, then saves the image and displays it on the screen. The problem here is the timing of these different functions. The order in which the functions are executed matters, and every function takes a variable amount of time. Our first approach was to wait for the previous function to finish executing, but that did not work due to several reasons. What our judge at the moment does is to wait for a fixed amount of time after each function is called. These amounts were adjusted over time by simple trial and error. This solution works most of the time but introduces waiting time for the user because all the waiting times sum up to more than four seconds, during which the user cannot use the environment. As already mentioned, this does not, however, work all the times and sometimes introduces render errors.
What we have in mind to fix this is to introduce a new instruction to the XLogo language. We will add this `go` instruction to get a signal when the program is finished with drawing the task. With this approach, we are able to wait a dynamic amount of time, which is necessary due to every computer running slightly differently.

6.2.2 Semantic analysis of a program

We want the novice programmers to experiment as creatively as possible when programming. Nevertheless there are some program paradigms one could use. In order to give feedback about these programming paradigms, we need to semantically analyze a program. This kind of analysis is hard to implement, because we need the computer to not only understand the language (we already have the interpreter for XLogo), but also recognize groups of instructions. This task is out of scope for a bachelor thesis. We still offer a few thoughts on this in the chapter about the design.

6.2.3 Disadvantages

Our approach to tackle the problem of teachers having a lot to do is to support the teacher in the tasks that can be easily done by a computer. We offer an autonomous way of learning

XLogo. Our goal was never to completely eliminate the need of a teacher, simply to support them. Nevertheless, one could argue that the novice programmers learn how to get feedback from a machine and give this feedback too much weight.
Another disadvantage in our approach concerns the way we present the task. We decided to display the task to be solved in the form of a picture and the programmer needs to redraw this picture. This form lets the novice programmers choose freely which programming paradigms, if any, they want to use. However, presenting the task in the form of a written description offers advantages over this approach. In a written description we can include the length of all the lines present. To pass on this information we now rely on our background grid, that is ideal for orthogonal lines but offers little information for triangles or circles. We go into more detail about the background grid in the next section about future work.

6.3 Future Work

Adding a simple judge to the XLogoOnline environment opens many different doors. In this section, we have a look at advancements that are possible with our judge and in what direction future research may go.

6.3.1 Improving the judge

In this work, we implement a basic judge for the XLogoOnline environment. In this subsection we have a look at further improvements of the judge.

The background grid In the background of both the drawing area and the task to solve we added a square grid to let the user see more easily which dimensions the shape to draw has. But it limits the shapes we can offer as tasks, since a circle won't fit in this grid, without requiring the users to calculate with π. What we could offer here are different grids, each one fitting the geometrical nature of the task. We could add a hexagonal grid, a circle grid, or any other geometrically shaped grid.

The length and angle of lines Another thing not yet present in the judge is the length or angle of the individual lines of the displayed reference solution. The grid already offers intuition about the length and angle of the lines, but since not all angles are orthogonal to each other, we suggest to add the length and angles of the different types of lines at least once to the reference solution image.

Difficulties for the tasks Another feature that would nicely add to the functionality of our judge is having varying difficulty levels for the tasks. This could be achieved by letting the programmers that solve the tasks rate a tasks difficulty afterwards. This information can then be used in a way that the novice programmer is able to choose from (for example three difficulty levels). Such an addition to our judge would maybe offer more exercise motivation since now

there are three tasks to solve each day. A programmer could have the goal to always be able to solve the hardest task.

Degrees of Feedback We deliberately left out the metrics in the feedback because we want the novice programmers to think in the most creative way about solutions to a task. An addition here is to let the users decide for themselves what kind of feedback they want. We can add different levels of feedback to the judge, that offer evaluation of different metrics and generate a more detailed feedback.

Tasks with colors For now, all the tasks the novice programmer could solve, were painted black. Adding tasks with different colors allows for more tasks, although we have to make sure that we communicate exactly which color was used. Another potential issue is that when drawing with several distinct colors, the order in which it is drawn over the same pixel matters. If we decide to include this feature, we have to consider these issues.

The differences metric We discussed the metric to highlight the differences in the submitted image and the reference solution. What we could add here was a mechanism that detects whether a picture was just drawn wrongly scaled (too small or too big). This allows the user to find the errors in their programs even faster.

Beautify the interface As of now, our judge offers just two buttons and a feedback window as an interface. Especially to make the judge more attractive to young novice programmers, we can add a judging-turtle that represents the judge and not only introduces the programmers to the judge but also give a personal feedback. This can lead to an easier understanding of the feedback presented for the children.

"XLogo goes Blocks" and "XLogoOnline Turtle emerges into real life" Since our judge is a part of XLogoOnline, we want recent development and new features added to XLogoOnline, such as "XLogo goes Blocks" and "XlogoOnline Turtle ermerges into real life", also to be able to provide the programmers with a judge. Depending on the changes in the interface for the different environments of XLogoOnline, we need to adapt our judge accordingly.

6.3.2 Follow-up projects

Training area One follow up idea that was discussed was to offer a training area for XLogo. The basic idea behind this is to offer different training programs that train different program paradigms or programming skills. The novice programmers can decide what skill they want to learn and then get adequate training. The metrics we discussed in chapter three can be all used individually. We also can add different kinds of tasks. An example here is to give each task a program paradigm to use. This can

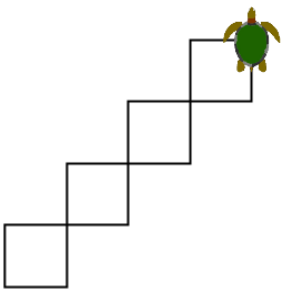


Figure 6.1: *A Logo program using* `repeat`

be for example the usage of sub-programs (modular design). Or in a task with a lot of repetitions (see Figure 6.1) we can require the use of at least one `repeat` command and therefore grade how many times a `repeat` was used and give feedback about how many times it is possible to use `repeat`. This could be achieved by grouping together the different types of programs (different program paradigms) and create a judge for each of the types.
Other possibilities for this training area are to have tasks that build upon each other. As an example we can have tasks that first draw a square, then add a roof to it in the next task and in the last task let the programmers build a "road with houses", in which they can use the previously programmed programs.

Cooperation with schoolbooks The schoolbooks written by the chair of theoretical computer science, Prof. Dr. Juraj Hromkovič and the ABZ group about programming with XLogo offer lots of different ways to exercise programming with XLogo. One thing we can use the judge for is to offer all these tasks in the XLogoOnline environment. This allows for concise feedback and also lets us build into the feedback the most important points and skills of the relevant chapters. This addition to XLogoOnline offers another way to take in the knowledge presented in these schoolbooks.

Exam system for schools Another follow-up project can be to use parts of the judge to design a exam system for the classroom. This system should be able to let a teacher activate a sheet of exam questions for each student and then automatically collects all the results and analyzes parts of it according to the metrics the teacher set.

In conclusion, our judge is still incomplete. With this work, we want to provide future research with a foundation to carry on this project in different directions. There are many ways to improve our judge and also many ways to use the judge to add a new feature to XLogoOnline.

Bibliography

[1] *XLogo Benutzer-Handbuch*. https://downloads.tuxfamily.org/xlogo/downloads-de/manual-html-de/toc.html[Last accessed: 2019-01-15].

[2] Kirsti M Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2):83–102, 2005.

[3] John Hattie and Helen Timperley. The power of feedback. *Review of educational research*, 77(1):81–112, 2007.

[4] Juraj Hromkovic. Einführung in die programmierung mit logo. *Vieweg+ Teubner*, 2010.

[5] Jacqueline Staub. *xLogo online - a web-based programming IDE for Logo*. https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/155855/eth-49742-01.pdf?sequence=1&isAllowed=y[Last accessed: 2019-03-09].

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

WELL DONE! KEEP IT UP!
XLOGOONLINE TURTLE IS READY TO JUDGE YOUR CODE

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
WEIBEL	DOMINIC

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZURICH, 15.03.2019	D. Weibel

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.