

xLogo online - a web-based programming IDE for Logo

Master Thesis

Author(s):

Staub, Jacqueline

Publication date:

2016

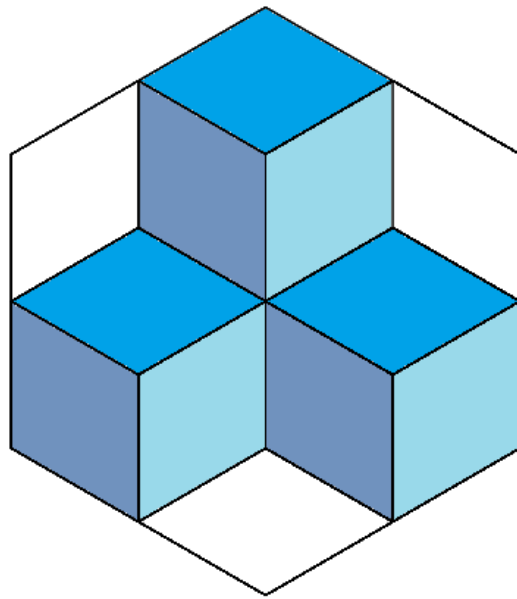
Permanent link:

<https://doi.org/10.3929/ethz-a-010725653>

Rights / license:

In Copyright - Non-Commercial Use Permitted

xLogo online - a web-based programming IDE for Logo



Jacqueline Staub

Master Thesis
4. October 2016

Supervising Professor:
Prof. Dr. Juraj Hromkovič

Supervising Lecturer:
Giovanni Serafini

Abstract

Algorithmic thinking is increasingly recognized as a vital skill and schools across Switzerland are widely adopting Computer Science in their curriculum. Teachers are now having to expand their lessons and are in need of informative and educational teaching materials. A difficulty they face, however, is how to teach programming in their classrooms, especially to young children who are both newcomers to the digital world and also need to understand and practice new and abstract concepts. Visual aids have been shown to greatly support this learning process because pupils are rewarded by appealing images and immediately see the effect of their changes, which makes exploration fun. We borrow on these ideas and in this thesis we present XLogo online, a web-based and graphical programming environment which features a turtle and a small set of drawing commands. Our solution is tailored to fit a widely-deployed curriculum, which has already been in use for more than nine years and is taught in in hundreds of courses spread around Switzerland. The contemporary circumstances ask for an intuitive and stable programming environment which is easily accessible from anywhere and which serves the pupils during their learning. Our system meets all these needs and can be deployed at the click of a button using only a web browser and, thanks to its reactive design, can be run on a broad set of devices with different screen sizes and platforms. A distinguishing feature from comparable IDEs is that our application is uncluttered and easy to use; everything is shown on a single page and the number of visual contents have been reduced to a bare minimum. This structure was proven to be intuitive and generally approved by primary school pupils in a user study. Comprehensive experiments with over 80 programs confirmed that our implementation works as expected, generates correct results and is sufficiently fast to render even complex drawings with thousands of lines within only a few seconds. Our system eases the learning process thanks to, among other things, immediate feedback shown directly in the editor. We implemented an on-the-fly syntax checking mechanism by smartly re-running the interpreter after keystrokes. Syntax checking eases the understanding of code, because of the direct feedback, which reduces the cognitive load.

Acknowledgment

“If your life does not have serious ups and downs you are doing something wrong.”

This sentence glared at me from the wall next to my spot in the office for the whole time of my master thesis. After a few weeks in the group it became quite evident, that everyone in this research team strongly believes in the deeper meaning of this exact sentence. I remember one day, when Prof Hromkovič stepped into my office at around nine o'clock in the evening. He was obviously surprised to find me still sitting in my chair, thus he asked how I was doing. This day was a particularly hard one, I was debugging for five hours straight without the slightest improvement. Something was just really wrong that day. I told him about it and found myself astonished by his answer. He told me that I should be glad about my struggle, because it is exactly those days that teach us the most.

I would like to thank the following people for their exceptional support, wisdom, friendship and that they took time for me. First, I would like to thank Prof Juraj Hromkovič and his group for open-heartedly welcoming me in their research group and sharing their knowledge with me. Within this group, a special thanks belongs to Giovanni Serafini, my direct supervisor in this thesis. He took good care of me and ensured that I had the best environment he could provide for me. Thanks to him I got a spot in one of the group's offices and in our weekly meetings he provided mental and technical support for every problem I was struggling with. Next, a big thanks to Zaheer Chothia. He took a huge amount of his time to mentor me, he provided many useful tips and taught me how to master difficult problems. Moreover he stood by me as a very good friend whenever I experienced a rough time in my thesis.

I am greatly thankful for everything the people surrounding me did and how I was treated during my thesis. It is anything but self-evident to find such good circumstances when writing a thesis and I want to be upfront and express my great gratitude for this to everyone who did his part in supporting me during the last six months.

Contents

List of Figures	ix
List of Tables	xi
1. Introduction	1
1.1. Problem Statement	2
1.2. Goals	3
1.3. Related Work	4
1.4. Structure of this Document	6
1.5. Conventions	7
2. Background	9
2.1. Research on Learning	9
2.2. Why is Computer Science a good Educational Background?	11
2.3. Teaching Computer Science	12
2.4. Swiss Schools and Computer Science	12
2.5. Prior Experience Teaching XLogo	13
2.6. Technical Background	13
2.6.1. Angular2 Core Concepts	13
2.6.2. TypeScript	19
2.6.3. ANTLR4	21
3. Requirements and Feature Analysis	23
3.1. Use Cases	23
3.2. Functionality and Requirements	24

Contents

3.2.1.	Core Functionality	24
3.2.2.	Optional Functionality	26
3.2.3.	Responsive Graphical User Interface	27
3.3.	Restrictions	30
3.3.1.	Offline-Mode	30
3.3.2.	Mobile	31
3.3.3.	Browser Statistics	32
4.	Design	37
4.1.	History of Web Development Technology	37
4.2.	Design Choices	39
4.2.1.	Overview	39
4.2.2.	Frameworks and Design Patterns	39
4.2.3.	Implementation Language	42
4.2.4.	Code Editor	45
4.2.5.	Rendering	45
4.2.6.	Libraries, Tools and other frameworks	46
4.3.	Prototypes	47
4.3.1.	Pure HTML5, CSS3, JavaScript	47
4.3.2.	Antlr4	49
4.3.3.	Pure HTML5, CSS3, TypeScript	51
4.3.4.	Angular2	52
4.4.	Final Design Choices	53
5.	Implementation	55
5.1.	General Structure	55
5.2.	Information Flow	58
5.2.1.	Case 1: Both the editor and the user input are valid	58
5.2.2.	Case 2: The editor is valid, but the user's input is not	59
5.2.3.	Case 3: The editor is invalid	60
5.3.	Realization on Selected Technical Features	60
5.3.1.	Loading and Storing Editors	61
5.3.2.	Error Highlighting	63
5.3.3.	Canvas Panning	65
5.3.4.	Window Resizing	66
5.3.5.	Multiple Languages in XLogo Online	68
5.3.6.	Colors	68
5.3.7.	Asynchronous Tasks in Single Threaded Environments	69
5.3.8.	Integration of Multiple Module Loaders	71
5.3.9.	Visitor Pattern in JavaScript	73
5.4.	How the Project is Instantiated	73
6.	Evaluation	75
6.1.	Benchmarks	75
6.1.1.	Benchmark: Small Test	77
6.1.2.	Benchmark: Loop Test	79
6.1.3.	Benchmark: Large Test	81

6.1.4. Benchmark: Timing Test	83
6.2. System Testing	85
6.3. User study	87
6.3.1. Goals	87
6.3.2. Setup	87
6.3.3. Evaluation	88
6.3.4. Conclusions	89
7. Conclusion	91
7.1. Conclusion	91
7.2. Future Work	92
7.2.1. Graphical user interface	93
7.2.2. Cooperative Programming	93
7.2.3. Movable Canvas	94
7.2.4. Responsiveness	94
A. Appendix	97
A.1. XLogo grammar	97
A.2. Master Solution to the Exercises in ABZ Booklet	99
A.3. Questionnaire for User Study	116
A.4. Supplementary Exercises for Collaborative Coding	119
Bibliography	127

Contents

List of Figures

3.1. Xlogo online, editor on the left visible at all times	25
3.2. Browser usage statistics	32
3.3. Operating System Statistics	33
3.4. Safari usage statistics	33
3.5. Chrome usage statistics	34
3.6. Firefox usage statistics	34
3.7. IE/Edge usage statistics	35
4.1. High-level Overview: Design Choices	39
4.2. Composite pattern in our web-application	41
4.3. Mockup for XLogo Online	48
4.4. First prototype using HTML5, CSS3 and JavaScript only	48
4.5. Prototype using Angular2	52
5.1. Components in our web-application	56
5.2. Component-tree	56
5.3. Information flow which happens whenever the user hits the ENTER key. Boxes show UI components, arrows show the control flow	58
5.4. Information flow with invalid user input but valid editor. Control flow interruption after syntax checking the user input or executing it.	59
5.5. Control flow interruption after performing syntax checking with invalid editor.	60
5.6. Offscreen canvas with viewport, allows panning within viewport	65
5.7. Upon requesting a string lang[C], the LanguageService checks the current language settings and accesses the respective string from the responsive LanguageServiceHelper.	68
5.8. Parsetree for program square with wait: repeat 4[fd 100 wait 100 rt 90 wait 100]	69

List of Figures

5.9. Program square with wait, correctly implemented, after setTimeout is invoked, we remove all siblings of any of our ancestors. We make sure every method invocation is only made once at exactly the time when it is intended, while staying synchronous and responsive at all times.]	70
6.1. Small benchmark program, which draws a square. Parsing it results in a tree with 92 nodes	77
6.2. Small benchmark results, Experiment 1	78
6.3. Small benchmark results, Experiment 2	78
6.4. Loop benchmark program, which draws a star. Parsing it results in a tree with 47 nodes	79
6.5. Loop benchmark results, Experiment 1	80
6.6. Loop benchmark results, Experiment 2	80
6.7. Large benchmark program, which draws skewed circles. Parsing it results in a tree with 52 nodes	81
6.8. Large benchmark results, Experiment 1	82
6.9. Large benchmark results, Experiment 2	82
6.10. Timing benchmark program, which draws a filled square using the command wait. Parsing it results in a tree with 81 nodes	83
6.11. Timing benchmark results, Experiment 1	84
6.12. Timing benchmark results, Experiment 2	84
6.13. Linear increase in runtime when re-executing the program without clearing the canvas	86
6.14. Most students assessed the IDE to be between good or even very good	88

List of Tables

2.1. Lifecycle hooks in Angular2	17
--	----

List of Tables

1

Introduction

We experience a technological paradigm shift from a single computer (which was usually a desktop computer and used as a single point of data-storage) to a world, where every person possesses and uses many devices, from desktop computers, to laptops, netbooks, tablets, smart-phones, smartwatches and many more. Today data is no longer stored on a single device, but rather shared among the many devices people own. As a means of connecting the devices, we make heavily use of the Internet. It provides a platform which is tailored for our needs: Sharing information, getting connected and being interactive on a global scale is now possible, due to the existence of the Internet. Browsers have become very convenient, universally usable and powerful eg. with modern technologies like HTML5 and others. We introduce a new programming environment, which uses these state-of-the-art web-technologies to give primary school pupils an understanding of the basic concepts in Computer Science and to let them practice algorithmic thinking by programming in a visual appealing and fun environment. We stand out against other programming platforms by supporting a wide variety of different devices, screen sizes and resolutions, on various operating systems and, beyond that, without the need of installing any software to the client's device. Using the browser offers possibilities, which go beyond the capabilities of enhancing the experience and learning process, by using supported technologies like 3D animations, video, audio or by providing a platform, which can be used in an interactive fashion. Our programming IDE was designed to accompany a curriculum whose main purpose it is to convey the basic concepts of programming to primary school pupils and which uses Logo as the underlying programming language.

1.1. Problem Statement

In a recent dispute about the Swiss educational system, a vast majority of politicians claimed an amendment of the current curriculum. In a draft proposal, known as *Lehrplan 21*, they state the new requirements, which have to be met by Swiss schools. One modification is that Computer Science has become a more central role and is henceforth included in the curriculum on all school levels. This inflicts lots of teachers to create a new curriculum, which serves their pupils understanding about basic concepts in Computer Science. They are in need of good teaching material and programming platforms for their courses. We built a programming IDE which is tailored around an existing curriculum which may be used in primary schools. It is based on web-technologies, is designed to be stable, intuitive and extensible. We will discuss the problem now in a more thorough and detailed way by first explaining the characteristics about Logo. Later we will continue by demonstrating the main features about the existing curriculum, we built our web application for. We will continue with a short explanation about the dialect we used, our prior experience with the curriculum and a conclusion about our project:

Logo in general

When teaching Computer Science, there are different opinions on what and how to teach the basic concepts. The Center for Computer Science Education (german acronym ABZ) ¹ decided to offer a programming course for children and their teachers using a programming language called Logo. Logo is a programming language invented by Seymour Papert and his team at the massachusetts institute of technology (MIT), designed to be used by novice programmers [Papert]. We implemented the dialect XLogo, whose language specification is given in the Appendix.

ABZ course

ABZ organizes courses for Swiss primary school pupils since 2005 [Matter 2011]. The organization strives for a broad motivation of pupils for Computer Science. Swiss schools can register for a Logo course, which typically lasts for either five or ten weeks, with two or four hours per week. During the course the pupils are taught the basic concepts of Computer Science, like repetition, modular development, programs and subprograms, parameters and others.

XLogo dialect

Loïc Le Coq, a french mathematics professor wrote a software called *XLogo*, which was used in the course in the beginning [le Coq 2014, le Coq 2009]. *XLogo* is also the name of the dialect used in the course. Three theses were written by former bachelor students at ETH to improve XLogo and to produce a portable software for mobile devices and robots. The teaching material for these courses is available online [ABZ 2016]. We encounter a predominantly positive feedback by the pupils and their teachers, who obviously enjoy the programming courses a lot.

Prior experience

The course is organized strictly problem oriented by instructing the children to solve problems

¹Ausbildungs- und Beratungszentrum für Informatikunterricht der ETH Zürich

to the current topic directly on their computers. Most of the time is used for programming, while the teacher or instructor occasionally gives direct instructions to the whole class, explaining new concepts [Reusser 2005]. The course introduces the children to the concepts of loops, methods and parameters on the basis of a minimal set of commands. The pupils learn to program step-by-step, writing code with an increasingly versatile set of commands. The teacher has the opportunity to support the children in a different way than usual: He does not need to teach the children, but simply to support them whenever they need help. The job of judging whether a solution is correct or not can easily be performed by the children themselves. The children usually help each other and enjoy the interaction a lot. A manifold of tasks helps the pupils to learn the concepts of repetition, modular design and parameters. After the course, the children are able to solve tasks involving any of the above mentioned concepts.

Our project

In this project we focused on the usage of web-technologies, while pursuing at least the same functionality as the previous software used in the class room. The Internet offers a few advantages over the previous systems: On the one hand we can argue that every modern computer has at least one browser installed and people know how to work with it. If a child does not know how to handle a browser, it is a great opportunity for him to get in touch with the Internet, since working on the web is considered a crucial skill nowadays. As every browser comes along with JavaScript already built-in, the installation of our program is as simple as it can be. No additional program is required in order for our XLogo online programming IDE to work. Schools which do not provide an Internet access for every computer are provided with an offline version of our website, which is run on a local web-server. It behaves the same as the online version, with a few minor differences in the functionality provided.

1.2. Goals

The goals of this project are to develop a web-application with state-of-the-art web-technologies for XLogo. The product should be usable online and offline ² and it should translate and detect the user's language and integrate it into some sort of support for multiple languages in the UI. The programming IDE must support basic commands to perform arithmetic, iteration, methods, conditionals and variables used in the teaching material provided by ABZ, the course book *Einführung in die Programmierung mit Logo* by Juraj Hromkovic [Hromkovic 2010] and few addition commands which were introduced for increased convenience in the class-room setting. The website should work on all modern browsers and should be performant. A complete list of all requirements can be found in Section 3. In summary, we pursued the following goals in this project:

1. Development of a programming IDE for XLogo as a web-solution
2. State-of-the-art technology
3. Universally usable among different browsers

²including a few extra requirements for the offline version, namely to install the client which runs the website on a local web-server

1. Introduction

4. Multilingual graphical user interface support
5. Online-Offline support

1.3. Related Work

In this section we are going to present related work to our field. We will begin with the most closely related work, which also uses XLogo and thus has the same audience and the same environment as we do. In this section we present the work of three Bachelor students, who developed prototypes with similar aims as this project on different platforms or with different technologies. We will describe their findings as well as the difficulties they faced.

1. *XLogo4Schools [Živkovic 2013]*: Marko Živković thoroughly inspected the features and the implementation of the application XLogo by Loïc Le Coq. The student found several flaws in the design and the implementation of the software XLogo and proposed improvement strategies for those deficiencies. He found, that in its specification the core of the dialect XLogo was not complete in regard of the language constructs used, which made it impossible to correctly highlight errors in the code. From those insights he concluded that, in order to implement an error highlighting mechanism, the language needs to be redefined. He refactored XLogo and reengineered the core of the application. Furthermore, he was able to enhance the interpreter to be more efficient and created a simpler user interface with several new features (automatic saving and multiple editor sheets).
2. *XLogo Mobile [Häfliger 2014]*: Lukas Häfliger implemented a programming IDE for XLogo on mobile devices. For this purpose he used Unity, which is an engine to develop code, which can be ported to a series of different platforms, Android and Apple devices among them. He mastered the problem of the incomplete language specification by separating the language into two parts, which each on their own, are complete. This comes along with the obvious drawback that he needs to manage twice the number of interpreters, which are actually needed. He faced some difficulties with ANTLR, which forced him to implement the interpreter by hand. In our project we use ANTLR and we did not encounter the bug in our project.
3. *XLogo Robot [Pietra 2015]*: Priska Pietra enlarged the work from *XLogo4Schools* to not just program a turtle on the screen of a computer, but to program a robot which executes the previously written routines directly on an Arduino Robot. The usage of robots has a motivating effect on the children and thus serves their learning progress. The code written in Logo is transpiled to C++, which then can be loaded onto the robot to execute.

In the following list we present some recent projects in which state-of-the-art web-technologies have been used with an objective related to ours:

4. *Web-based IDE for IDP [Ingmar Dasseville 2015]*: IDP is a declarative language. For IDP a web-based IDE was built. They support code sharing, as well as an offline-online mode. Also they support a tutorial-mode, which can be used to learn the language and to get some information on how to use the application ie. a live tutorial. They used Bootstrap as one of their key-technologies which lead to a clear and visually appealing overall structure.
5. *Cloud9 [Cloud9 2016]*: Cloud9 is an online IDE for JavaScript which runs in the cloud. It supports the access to a virtual machine and a lot of other features, like code completion and error highlighting. It has a debugger, and allows communication among multiple developers in form of a chat functionality. This is helpful for people who work on the same code-base.
6. *Codiad and ICEcoder [Codiad 2016, ICEcoder 2016]*: Codiad and ICEcoder are general-purpose online IDEs which can be used to code in any programming language. Both of them have an online as well as an offline mode. They are not able to run the documents, though, since their purpose is to write code which then can be compiled and executed in a separate step. They use PHP on the server and an editor called ACE in the front-end.
7. *Atom [Atom 2016]*: Atom is a purely offline programming IDE, which is based on state-of-the-art web technologies, which allows the software to be used across platforms. It is built using HTML, CSS and JavaScript and runs on Electron, which is a framework to build cross-platform apps.

We did not choose one of the above-mentioned approaches because they are not suited for the special needs of children, who are exposed to programming concepts for the first time and whose skills in typing might not yet have been developed. There are other programming languages which can be used to teach children to program. We would like to present some of those languages and highlight the differences of those languages to Logo and why they are considered to be good languages for a novice programmer, who might be a child.

8. *Scratch and Alice [John et al. 2010]* are two programming languages for children, which use a block-based programming approach. This means that the user is typically not typing code but dragging and dropping boxes which have a certain meaning in the program. This kind of programming experience is particularly suited for young children who might still struggle to type text on a keyboard. All of the following environments have been tested in classroom settings and have shown good usability with children.
9. *Droplet [Bau 2015]*: One drawback of Scratch and other visual programming languages is that they do not scale well for large programs or when multiple developers need to collaborate. Recognizing this limitation, Bau et. al developed an environment, which has both a text and visual representation. The transition is seamless and this helps the children to learn how to create their own text-based programs.
10. *JavaKara [Raimond Reichert and Hartmann 2001]*: JavaKara is a programming editor designed for learning Java. It is text-based with a syntax similar to Java. The language used is strongly typed and therefore useful as a preparation for learning a strongly typed

1. Introduction

language like Java or C#. Different versions of Kara have been created. They can be used as a preparation to learning JavaScript, Python or Ruby among others. JavaKara has a wide variety of different applications. In *MultiKara* the user has the opportunity to learn about parallel programming. *TuringKara* provides a platform to learn the concept of a turing-machine.

11. *IPython/Jupyter [Pérez and Granger 2007]* : They provide the programmer with code or text visualization plots, which provide additional support during the programming process. These programming environments also run in the browser and have a server, which runs every language in a server back-end environment.

This list of related work is not comprehensive and other introductory programming languages exist such as Lego Mindstorms, App Inventor and others. Broadly, these are all visual languages and the main interaction is clicking rather than typing. They all differ in that the IDE needs different widgets, which can usually be clicked and dragged into a distinctive part of the program, which is executable.

1.4. Structure of this Document

Before proceeding with our approach we first give a short overview of this document:

In **chapter 2** we will discuss the relevant background for this project. Therefore we will present some facts about teaching and learning new concepts in general and in regards of teaching and learning Computer Science. We will make a statement why learning Computer Science is essential for general education and why every pupil should be educated in Computer Science early on. We will conclude this chapter by telling some of the experience we gained in the Logo courses we held so far and what impact the previous projects had.

In **chapter 3** we will give a formal statement on the requirements we had when starting the project. These requirements affected the design choices and the technologies we decided for.

Chapter 4 shows every aspect of the design choices we made during the development of the project. A short summary of the milestones in web development history will be given. We will show some of the prototypes we designed when starting the project and describe the factors which lead the design choices we made.

In **chapter 5** finally an insight on the development of the project will be given with a few code snippets and explanations of how the IDE was realized from a technical perspective.

Chapter 6 is about the evaluation of the project we developed. We will demonstrate the results of a user study which we conducted as a usability study, we will demonstrate a benchmark we created for this evaluation and we will give some qualitative measures on how performant our system runs.

In **chapter 7** we conclude our work and state a few possibilities for extending our work in the future.

1.5. Conventions

In the remainder of this section we will introduce a few conventions. Those were made for the sake of readability and brevity of this document. Throughout this report, we will adhere to the following conventions:

- Code examples will be given in TypeScript (is a statically-typed programming language which is a superset of JavaScript) if not specified otherwise, (due to the fact that the project was written in TypeScript mainly).
- We sometimes use the term *screen size* for the effective size of the screen but more often as the resolution in pixels. The intended use should be clear from the context.
- To reduce redundancy and increase readability, we refer to *the user* or other pronouns as a male person. We do not intend to discriminate anybody, though.

1. Introduction

2

Background

In this chapter we will discuss a few background topics, which will be useful in later chapters for understanding the choices we made in our project. We will first present the current research topics on teaching Computer Science and state why these topics are important for a good educational background. Later we will follow up with a short introduction to our technical preliminaries. We will introduce the most important keywords and concepts which have been used in our project.

2.1. Research on Learning

The mechanics of the human brain are highly sophisticated; after more than a century of extensive research on the topic, we still do not understand the complex interactions taking place in the brain while learning. We would like to present the four most prominent and influential theories in the area of Psychology of memory from the 19th and 20th century as well as their impacts.

- *Theories around 1900*

Two German psychologists, Wilhelm Wundt and Hermann Ebbinghaus, were the first researchers who concentrated on the processes in the brain in self-experiments. They founded the basis of psychology of memory in the years around 1879. In their experiments Wundt and Ebbinghaus set themselves the task to learn a list of meaningless syllables by heart. After a certain time they tried to recall as many syllables as possible. Their key findings were the following [Seel 2012]:

2. Background

1. *Ebbinghaus effect*:

Ebbinghaus stated the theory that the number of repetitions influences the recall success rate: The more repetitions a test subject faces the better the recall.

2. *Learning curve*:

Apparently the amount of subject matter correlates with the recall success rate. To achieve the same success rate while facing more learning content, considerably more effort had to be undertaken by the test subject.

3. *Primary-Recency-Effect*:

Subject matter, which was learned in the beginning or the end of a learning phase are remembered best by the test subject. It is the hardest to remember contents which were learned in the middle of a learning phase.

● *Behaviorism*

In behaviorist theories the human mind is analyzed as a black box. A few psychologists (lead by John B. Watson) started the theory of *behaviorism* in the USA as a reaction on the findings of Wundt and Ebbingshaus. Rather than carrying out self-experiments, they performed their experiments on animals. Basically their experiments involved animals whose behavior was analyzed after rewarding or punishing the animal. In behaviorist learning theories this is often referred to as a concept called *reinforcement*. Iwan Pawlow invented a theory named *classical conditioning*. He showed that *neutral stimuli* (which are defined as stimuli that do not lead to a distinctive reaction) can become *conditioned stimuli*, by connecting the neutral stimulus to a certain object of interest for the animal (eg. food) which induces a certain behavior by nature (eg. barking).

B.F. Skinner proposed another theory called *operant conditioning*, in which he describes that the behavior of a test subject can be manipulated by treating the subject differently. It is possible to reinforce the subject by rewarding it for good behavior or by removing an unpleasant situation. Punishment on the other hand (which can be realized by directly applying a penalization or by removing a pleasant situation) will induce the destruction of a certain behavior. [Anderson 2000]

● *Cognitive revolution*

In the year 1960 a new approach was suggested by Swiss psychologist Jean Piaget, who approached the problem in a white-box approach. His theory, which lead to an area called the cognitive revolution, was out to analyze the mind directly, which contrasts with the black-box-approach in behaviorist theories. Piaget pointed out, that it is not appropriate to examine the human mind as a black box. The existence of learning by imitation was found and research started focusing on the various processes involved in learning. [Piaget 1952]

● *Constructivism*:

Constructivism is the latest movement in Psychology of memory. In this theory, the cognitive process of learning is assumed to be constructive meaning that the learner has a prior knowledge or experience, on which he bases theories. Whenever a new construct is to be learned, it has to be built into the net of already known facts and constructs. In that sense new concepts have to be composed from existing experiences. Leading researchers promoting this theory were John Dewey, Maria Montessori and Jean Piaget among others. [Perkins 1999]

2.2. Why is Computer Science a good Educational Background?

Computer Science is rather young compared to other topics taught at universities. A retrospective view of the last twenty years shows how much it has grown in importance in such a short time span. Computers have become omnipresent, they are used in various topics like biology, chemistry, mathematics and many more. We can expect Computer Science to grow to an even bigger importance in the future. The aspect of automation has a huge impact on today's industry.

So far Computer Science was not part of an official Swiss school curriculum. There were a few fears, which caused this, namely: Computer Science is evolving too fast making it useless to learn programming languages, for example, since they change too often and therefore are not useful anymore a decade after the concepts were taught. Most people will not become computer scientists, so why should we address a large audience with knowledge, which may be obsolete and might only be used by a few percent of them anyway? We would like to counter with a few examples of why Computer Science deserves its place in a modern school curriculum:

Agreeably, computers are of high importance in the industry and most private aspects of our lives. Everyone uses them, but only few know how they work. Most people treat computers as black-boxes, which they can use to do amazing things. Yet, most people just know how to operate mainstream applications and as a problem arises, they think the problem cannot be solved at all. What can a technology be useful for, if only a tiny fraction of it is understood?

Computer Science has a stable core, which has existed for a very long time and will stay persistent. This core is called *algorithmics* and is the science of solving problems. We try to find ideas and to formulate them as concise that even an unintelligent machine can execute the idea and find a solution. There will always be problems, which have to be solved. Thus Computer Science will never become old. The key ideas can be taught even to very young children, since solving problems is not necessarily bound to words and numbers and it can be used in lots of different topics.

Computer Science can be used to learn how to think, and how to tackle a problem in a productive way. Also, Computer Science is a very broad topic, which can be correlated to many different topics like: Mathematics, Electrical engineering, Physics, Communication, languages and even Psychology. Computer Science can be used to improve our understanding in those topics better too.

In contrast to other school topics which are very abstract, Computer Science is much more hands-on. It can be seen as a practical compensation to the other topics. It is allowed and even wished-for to learn by making mistakes and just putting hands-on. By making mistakes, children can learn a lot about how to approach a problem.

Computer Science also helps us to judge ideas. In search of efficient solutions, children are forced to judge their approaches. If they find a way to calculate the right solution to a mathematical problem, they are happy. But they do not think ahead, trying to find better solutions. Also, they do not learn to judge problems. In Computer Science they learn to compare solutions and to distinguish good and clever solutions from those which might not be optimal and still could be improved.

2.3. Teaching Computer Science

In this section, we would like to discuss how Computer Science can be taught on a primary school level and the impacts we expect from the proposed approaches.

When learning to think algorithmically, we can choose to learn the concepts themselves, as proposed already by a few sources [Bell 2016]. We argue, that it can be achieved to learn Computer Science without learning to code, but if someone learns Computer Science while coding, he really understands the topics and can explain them to everyone (since they already proved to be able to explain their idea to someone as unintelligent as a computer). Therefore the ABZ decided to teach pupils the basic concepts of Computer Science by teaching them to program. Of course there are several programming languages, each designed with different purposes. It might not be best to start learning to program in a programming language with thousands of commands, a large standard library and complex semantics, as usually found in object-oriented programming. It might be better to start with a smaller language, which is designed to be simple and educational. One of those languages which are designed to be simple, especially for newcomers, is Logo. The ABZ decided to use a dialect of this language called XLogo for the courses. The concepts we teach them are (among others): Commands, Arguments, iterative programming, iteration, modular design, abstraction as well as parameters. For more advanced pupils further exercises build upon this and introduce more advanced topics including lists, variables and recursion.

2.4. Swiss Schools and Computer Science

For a very long time Computer Science was not a part of any official curriculum in Swiss primary school. An extensive study among Swiss high-school teachers and pupils (as well as secondary school teachers and pupils) showed that in 2008 a vast majority thought of Computer Science in terms of applying prominent software products like for example Microsoft's Office Suite [Stiftung 2008]. Only about 50% acknowledge Computer Science as a scientific topic in the field of engineering and basic research. Most pupils stated an interest in Computer Science, but when being asked to choose their topics, they chose text and image processing. Moreover there existed massive stereotypes about computer scientists causing school girls not to choose the topic [Pöpper and Altenhoff]. Recognizing this deficiency, ABZ decided to counter this trend in the year 2004 and started to create teaching material for primary schools and to visit school classes on-site introducing to them some of the most basic concepts of Computer Science. Doing so, they tried to counteract the wrong image people had about Computer Science. ABZ decided to address young pupils, so they could engage with girls and boys, while they are still children and motivated to learn. In the next section we would like to tell about experience we had in previous XLogo projects.

2.5. Prior Experience Teaching XLogo

Having more than ten years of experience, the ABZ has subsequently improved a few things in their learning material and was able to gain a lot of experience. Their impression was, that a very high percentage of the pupils greatly appreciate the project and love to solve the assignments on a computer. With the teaching material the ABZ are able to keep every child busy all the time, working most of the time on assignments in a booklet. Every pupil has the chance to work at their own pace. This has a positive impact on the motivation of the children, since no one is bored at any moment. Especially very heterogeneous classes can profit from such a setup. Girls are just as motivated as boys and as successful as boys. The course motivates both genders equally, which is great, since we can expect those girls to be less afraid of Computer Science later on, which might have a positive impact on the percentage of females studying Computer Science in a few years. The children learn to work individually and to judge their own programs, which is a skill they are supposed to learn at that age (nine to twelve year old pupils). Also we encountered a positive impact on the teachers. They take on the role of a supporter as opposed to an active lecturing role. He is there to help the children overcoming problems, while the computer (indirectly the children themselves) impersonate the judge, who decides whether a program is correct or not.

2.6. Technical Background

In this chapter we are going to introduce the frameworks and libraries we used in this project. We will talk about Angular2 first, and explain how it works and what it is good at. Afterwards we will present TypeScript, which is a language transpilable to JavaScript. We will explain the main characteristics of TypeScript and how it relates to and in what ways it differs from JavaScript. Later, we present ANTLR4, which is a tool to help developers generating parsers from grammars. We will explain how ANTLR works, what it is and how to interact with it.

2.6.1. Angular2 Core Concepts

Angular2 is a client-side framework, which is designed to facilitate the implementation of single-page web-applications. On September 15th an announcement about Angular2's alpha release was published. For the entire project we worked on beta version 17, due to the fact that the alpha release came out just at about the same time as this report was submitted. Its strongest features are the following: Angular2 enforces a clear structure using the MVC and Composite design patterns. We will explain more about the general structure in the next section. Its core programming language is TypeScript, which is a statically typed superset of JavaScript. TypeScript code is transpiled to JavaScript at compile-time. Angular2 provides a way to ensure two-way data binding as well as a completely modular approach of the web-page's setup, using components. We are now going to explain a few of these core concepts shortly. This will be useful to understand our web-application and design choices. More information is available online [Savkin 2016].

2. Background

Components

In the very core of Angular2 we find a concept called *components*. They replace the controllers and allow a more modular design in general. Instead of having a single controller, we implement several smaller functional and visual parts of the application, which we call components. Each of which has an individual controller instance. components are used to represent semantically distinct parts of a website, like headers, navigation-bars, independent sections and paragraphs, which do not share too much information with other parts of the website. The developer is free to decompose the website into components as he wishes. A component holds the information about how the respective part of the website should be rendered (in the component's template markup) and the according business logic in the component's class. We describe the rendering procedure and relevant aspects of this process in the component's so-called decorator. The decorator can easily be identified by the keyword *@Component*. A component might look like this:

```
1 import {Component} from 'angular2/core';
2
3 @Component ({
4   selector: 'my-app',
5   styleUrls: ['mystyle.css'],
6   template:
7     `
8       <h1>Hello World!</h1>
9     `
10 })
11
12 export class AppComponent {
13
14 }
```

In line 12 to 14 we see a usual TypeScript class. In this case it is a class called `AppComponent`, but it might have any other name too. The `export` keyword is used by the module loader. Similar to JavaScript we encapsulate functionality into independent modules, which can be loaded into other modules. We use modules to not pollute the global namespace, used by window for example, instead. In line 1 another module is imported, namely the one used by Angular to make a class become a component. In lines 3 to 10 we find the component's decorator. Within the decorator we have a selector, which declares the tag-name that can be used in order to instantiate a component, a reference to a style sheet as well as the template itself, which indicates what the component will look like and what HTML-elements (or probably other components) it will contain. If we decide to add a component to the current component's template, we build a component tree.

There always needs to be a root component. Which one we use as root is declared in our `main.ts` file. We call the `bootstrap`-method on a particular component. This component is our root in the tree.

Component Trees

Components can contain other components. This is a useful mechanism to engage in creating a higher level of abstraction. If we had another component which contained a text paragraph, it might look as follows:

```

1 import {Component} from 'angular2/core';
2
3 @Component({
4   selector: 'my-paragraph',
5   styleUrls: ['paragraphstyle.css'],
6   template:
7     `
8       <p>This is a paragraph and it contains some text.</p>
9     `
10  })
11
12 export class ParagraphComponent {
13
14  }

```

This component has the same structure as AppComponent did, except for the different name and the different selector. If we decided to include the paragraph into our AppComponent, we would end up with the following changed AppComponent:

```

1 import {Component} from 'angular2/core';
2 import {ParagraphComponent} from './paragraph.component'
3
4 @Component({
5   selector: 'my-app',
6   styleUrls: ['mystyle.css'],
7   template:
8     `
9       <h1>Hello World!</h1>
10      <my-paragraph></my-paragraph>
11     `
12   directives: [ParagraphComponent]
13 })
14
15 export class AppComponent {
16
17  }
18

```

Line 2 imports the ParagraphComponent as a module, on line 10 we use the tag declared in the ParagraphComponent in our own template and on line 12 we state that we use ParagraphComponent as a directive. A directive is another Angular2 concept and as such defined as a component with or without a template, which may be used by the current component.

2. Background

Lifecycle Hooks

Angular2 provides the programmer with lifecycle hooks. These are callback functions, which notify the programmer whenever a certain stage in the development of a component is reached. A few examples for available lifecycle hooks are the following: `OnInit`, `AfterViewInit`, `AfterViewChecked` and `AfterViewChanges`. The lifecycle hooks are available as modules in the core library of Angular2. In the following example we demonstrate how to use lifecycle hooks:

```
1 import {Component, AfterViewChanges, AfterViewInit, OnInit, ↵
   ↵ AfterViewChecked} from 'angular2/core';
2
3 @Component({
4   selector: 'my-app',
5   styleUrls: ['mystyle.css'],
6   template:
7     `
8     <h1>hello World!</h1>
9     `
10  })
11
12 export class AppComponent implements AfterViewInit, OnInit, ↵
   ↵ AfterViewChecked, AfterViewChanges{
13
14   ngOnInit() {
15   }
16
17   ngAfterViewInit() {
18   }
19
20   ngAfterViewChecked() {
21   }
22
23   ngAfterViewChanges() {
24   }
25 }
```

In general we distinguish lifecycle hooks, which trigger just before some event, from such which trigger just after an event. Usually the child-components are checked first and as a result, the parent component is checked afterwards. We have a few hooks, which trigger once only, and a few which trigger several times. Have a look at table 2.1, which states all lifecycle hooks and the circumstances under which they apply.

Lifecycle hooks are a handy way of manipulating change detection and reducing the amount of code necessary to catch a certain point in the lifecycle of a component.

Lifecycle Hook	When does it trigger
OnInit	Called once directly after the constructor finishes.
OnChanges	Called whenever the value of a bound property changes.
OnDestroy	Called once just before the component is definitely destroyed.
DoCheck	Called whenever the Input properties are checked.
AfterContentInit	Called once directly after the content is initialized.
AfterContentChecked	Called whenever the content was checked.
AfterViewInit	Called once just after the view is initialized.
AfterViewChecked	Called whenever the view is checked.

Table 2.1.: Lifecycle hooks in Angular2

Services and Dependency Injection

In Angular2 data is supposed to stay separated from the components, since various components might try to access the same data. Therefore we use services. A service provides a component with data. Usually a service is passed to a component through its constructor. Thus, the component already has the data when it is initialized. The following example shows how to inject a service into a component:

```

1 import {Component} from 'angular2/core';
2 import {MyService} from './my.service';
3
4 @Component({
5   selector: 'my-app',
6   styleUrls: ['mystyle.css'],
7   template:
8     `
9       <h1>Hello World!</h1>
10      `,
11   providers: [MyService]
12 })
13
14 export class AppComponent{
15
16   service: MyService;
17
18   constructor(s: MyService) {
19     this.service = s;
20   }
21 }

```


2. Background

If we include a service into a component, it can be used by the component itself as well as all of its children recursively. To use a service in a component we add a parameter of the corresponding type in the component's constructor. Everything which happens between passing the service as an argument to the constructor to really getting a reference is taken care of by Angular itself. We do not need to instantiate the service by hand. Instead we can just inject the service as an argument to the component's constructor. After initialization we can request data through the public interface of the service. The service is an ordinary TypeScript class, like the following:

```
1 export class MyService{
2
3     private text: string[] = ["a", "b", "c"];
4
5     constructor() {
6     }
7
8     addString(value: string) {
9         this.text.push(value);
10    }
11
12    getTextAt(index: number):string {
13        return this.text[index];
14    }
15
16    getAllData(): string[] {
17        return this.text;
18    }
19
20 }
```

Since a service is no component per se, we do not need to import the component module. The service is but a simple class with three public methods: `addString`, `getTextAt` and `getAllData`. These signatures are visible to the components and thus may be used from the outside.

Data Binding

Data binding is a means of synchronizing the component's view with the underlying model. Whenever the virtual machine finished one turn, all property bindings are checked and updated if it is necessary. We can profit from the support we get from the Angular2 framework, because it tremendously reduces the code length. Angular uses the `Zone.js` library to perform change detection. With this library it is possible to write wrappers for the built-in directives, like for example the `setTimeout` function, which creates a function handle, that can be executed after some time. If we want to dynamically change the text in the header tag of our `AppComponent` we write the following code:

```

1 import {Component} from 'angular2/core';
2
3 @Component({
4   selector: 'my-app',
5   styleUrls: ['mystyle.css'],
6   template:
7     `
8       <h1>{{mytext}}</h1>
9     `
10 })
11
12 export class AppComponent {
13   private mytext = "hello World";
14
15   constructor() {
16     this.changeText("show new Text!");
17   }
18
19   changeText(newText: string) {
20     this.mytext = newText;
21   }
22 }

```

Calling the method `changeText` results in an automatic update of the component's view, due to the changed property binding in the `h1`-tag.

2.6.2. TypeScript

TypeScript is a state-of-the-art programming language, which can be used, among others, for web development. It is a super-set of JavaScript, which means that any valid JavaScript program is a valid TypeScript program. In addition to all the features offered by JavaScript, TypeScript is statically type-safe. This means, that in addition to all the flexibility we get from JavaScript, we are given the safety from static types. The compiler supports the programmer and issues warning whenever his type inference algorithm is not sure whether a certain operation is valid or not.

We are going to explain the TypeScript basics in the following sections using a few examples. We will explore classes, constructors and methods, modules and interfaces.

Classes

One main improvement over JavaScript is, that TypeScript allows for classes. Within a class we can offer fields and methods. Once a class is instantiated, all of the class' public interface becomes usable by the client class. A class in TypeScript might look as follows:

2. Background

```
1 export class ExampleClass {
2
3     private a: number;
4     b: string;
5
6     constructor(){
7         this.a = 1;
8         this.b = "hello world!";
9     }
10
11 }
```

The class `ExampleClass` contains two fields and one constructor. One of the fields, `a`, is private and thus not visible from the outside, while the other field, `b`, is publicly available in every context. For both fields we declared a static type. TypeScript has a type checker, which informs the programmer whenever the expected and actual types do not match.

Constructors, Methods and nullable Types

As seen in the previous section, we can add constructors to classes in TypeScript. One special feature, which is offered by TypeScript is the one of optional parameters. This type is called *nullable*, which means that its type is either the static type itself or null. An example of a TypeScript class with constructors, nullable parameters and usual methods is shown here:

```
1 export class Employee {
2
3     salary? : number;
4     name: string;
5
6     constructor(age? : number, myName: string){
7         this.name = myName;
8     }
9
10    myMethod(wert1:number, wert2: number): number{
11        return wert1 + wert2;
12    }
13
14 }
```

In the example above we find a nullable field, which actually stays null in this case, because we do not assign to it in the constructor. Another nullable type can be found in the constructor. In this case the parameter is used as if it were a private field. In method `myMethod` you see that we require two parameters, which may not be null in the strict mode and that we return.

Modules

TypeScript works, similar to JavaScript, with modules, in order not to pollute the global namespace. You may see this in the previous examples whenever you find the keyword *export*. This keyword is used by the module loader to recognize which parts of the class to export.

2.6.3. ANTLR4

ANTLR4 is a tool for language recognition. In particular it is used to generate parsers, lexer, tokenizer and visitors for structured texts. The only thing which ANTLR needs in order to generate a functional parser is a grammar. A very simple example for an arithmetic grammar is the following:

```
1 grammar MyGrammar;  
2  
3 prog: (expr)+ '=' number;  
4 expr: expr '+' expr | expr '-' expr | number;  
5 number: NUMBER;  
6 NUMBER: [0-9]+;
```

This grammar (with main rule prog) identifies every equation, which is built from one or more expressions followed by an equals sign and a number. An expression is either a plain number or two expressions connected by a plus or minus sign. A number is any sequence of positive digits from 0 to 9.

In the next section we are going to explore the different use cases, requirements and features, which are important to our project.

2. Background

3

Requirements and Feature Analysis

In this chapter we would like to analyze the features and requirements, which we declare as scope of this thesis. We will start explaining the main setting of the Logo projects, which are carried out by ABZ and we will give a formal explanation of our use cases in general. Afterwards we will discuss the required features as well as a few statistical facts, which affected our design choices. Finally we will conclude the chapter by stating a few constraints which are imposed on our solution by the browser environment we are building upon.

3.1. Use Cases

ABZ carries out Logo courses in primary schools since 2004. In these courses classes of size 10 to 25 pupils are taught the principles of programming. In a smaller setting ABZ sends out just a single teaching assistant, who teaches the class and supports the children during programming with their teacher assisting him. In classes with many students, usually a teaching assistant is assigned one or two assistants as they teach the courses and support the pupils together. Teachers who already carried out the project a few times with different classes are encouraged to teach the children themselves, without any external help. In order for a teacher to be independent enough to teach a programming course, even if he is not experienced in programming himself, requires for a programming interface with a simple and comprehensive user interface, which can be easily used by novice students, who might not even be familiar in using the computer, mouse and keyboard at all. The programming IDE should provide the user with feedback which is comprehensive and sufficiently meaningful for a student to find errors themselves. A set of meaningful error messages and preferably some kind of error highlighting should be available.

3. Requirements and Feature Analysis

The children aged 10 to 12 usually are not familiar in using the computer and sometimes lack the fine motor skills for clicking the mouse, and typing on the keyboard. Hence a short language, that is to say a language whose commands can be written compactly) is helpful, so the children do not get frustrated and are able to make fast progress. The graphical user interface should adapt to the fact that the target audience's fine and gross motor skills are not yet fully developed and assist them by providing a larger size for text for example.

A vast majority of the children taught by the ABZ so far had no prior experience in programming whatsoever. Hence the cognitive load imposed on them by our curriculum was rather high. In order to reduce the cognitive load and to facilitate writing code without being hindered by a too complex system, we decided for a user interface, shows every functional component at all times.

3.2. Functionality and Requirements

In this section we would like to discuss the desired functionality and requirements we were aiming for in this project. We will begin with an explanation of the core features, before discussing some additional functionalities, which add to the project nicely.

3.2.1. Core Functionality

We will address the supported command set, discuss requirements for the drawing, the editor and the history.

Command-Set

The original language specification for Logo allowed to define lists containing either strings or code. This is problematic because executing a list either works or not, depending on the list's content. If we try to execute a string, we might (quite rightly) fail. Executing a list containing syntactically correct code on the hand should work. This discrepancy makes it difficult to provide the user with a good error highlighting. More on this topic can be found in the thesis by Marko Živković, who proposed to solve the issue by either not allowing having lists with strings or code, which look syntactically the same or by changing the language specification. We decided to resolve the problem by using only a subset of the available commands. Nevertheless a core requirement is that all exercises in the teaching material (namely the textbook and all the concepts in the textbook by Dr. Prof. Juraj Hromkovič) can still be run by our interpreter. We use the same subset of commands as used in the thesis by Lukas Häfliger. A comprehensive list of all commands can be found in Appendix A.1.

Drawing

Our platform is based on a graphical user interface which includes elements like an input field, a canvas, a small text field as command history and a larger text field as editor. An important

question regarding the requirement analysis is the positioning of these elements on the screen. The positioning might have an influence on the user experience and could depend on the screen size, resolution and orientation of the target device. A more detailed analysis of this topic will follow in section 3.2.3.

Editor

The editor is a text area, which can be used to declare new methods, which then can be invoked from the input text field. After first learning the most basic commands as well as handling the graphical user interface, pupils learn how to write new commands which can be used to build more complex drawings in a modular manner. We implement a feature to create multiple editors, which can all be used simultaneously and whose purpose it is to store the methods from the booklet in a orderly fashion. For every chapter in the booklet a new editor entry can be created. Similar to the approach chosen in the previous projects, we also decided to use one large file, which can be separated into multiple editors and thereby creating the illusion of multiple files.

Moreover we propose to keep the editor at a position, where it can stay open all times, in order to reduce the cognitive load imposed on the children. In case the screen is rather small, there needs to be a mechanism to maximize or minimize parts of the website on demand. A good user interface design takes care that no key element is not visible at any time [DONALD A 1983, Vaniea et al. 2012]. Out of sight, out of mind - i.e. we reduce the cognitive load by keeping all parts of the user interface visible all times. The ability to recognize errors in the editor increases if the editor, input area and history are visible at the same time. This was not possible before (see 3.1).

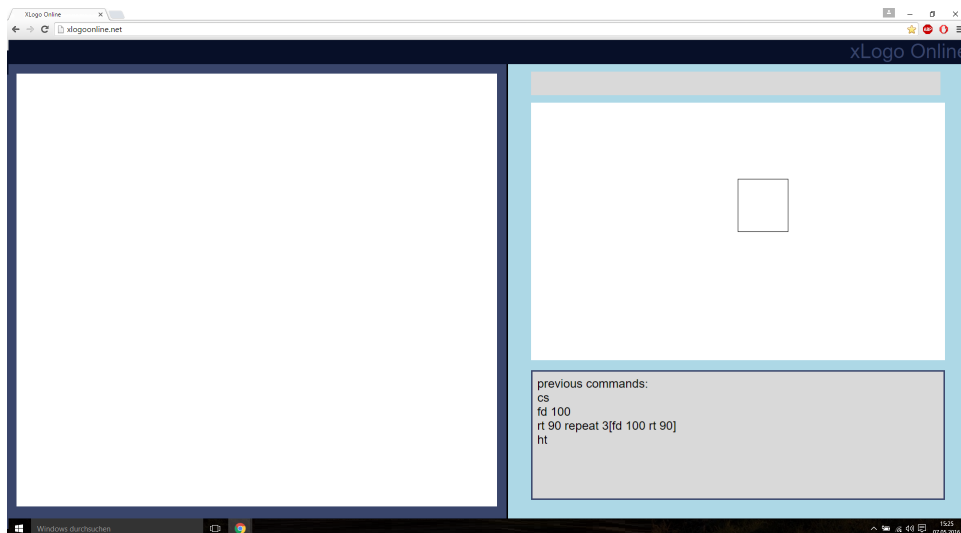


Figure 3.1.: Xlogo online, editor on the left visible at all times

Some school computers have small screen sizes. In those cases it is desirable to be able to hide the editor. For this reason we decided for a layout where the editor can be resized by the user. The pupils may maximize it on demand.

3. Requirements and Feature Analysis

History

The last core element is the command history. While working with children, the ABZ found, that children quickly start using the history element to reuse code they wrote recently and that it may be helpful to find repeating patterns. The command history enables the teacher to reconstruct the chain of thought a child has gone through and provides the pupil with a more concrete and personalized advice. Moreover we may use the command history to give feedback to the user, telling about errors, their possible sources and giving hints.

3.2.2. Optional Functionality

Next, we would like to address some additional functionality beyond the requirements, which we added in order to make the programming IDE more supportive for pupils in their learning process. We will talk about syntax highlighting, error highlighting, cooperative mode as well as the graphical user interface.

Syntax Highlighting

A first feature we wanted to implement was to provide a simple means of syntax highlighting. Children could profit from it by visually recognizing which of their commands are keywords, which are arguments and which are names. This may be helpful for the children to understand to structure and to spot errors and typos based on the color of the code, even before executing a sequence of commands.

Error Highlighting and Interactivity

With a good error highlighting we can provide the user with feedback directly at the location of the error. A consequence is that it is possible to spot errors quickly and to fix them, without any help from a teacher or teaching assistant. On the other hand we need to be careful not to use too many error types, so the user does not get confused, especially since we are working with children. Moreover we need to be careful to formulate the error messages as simple and understandable as possible. We used the same approach of error handling as in the thesis of Lukas Häfliger, which is based on the idea of a primitive error resolution [ARUNKUMAR PALANISAMY 2014]. We analyze an erroneous program for the following attributes:

- *Missing token at the end:*
Do we end up with a working program if we insert the token END at the end of the current method?
- *Missing token at the beginning:*
Do we end up with a working program if we insert the token TO at the beginning of the current method?

- *No such method name:*
Is the current token interpreted as a method name but we lack a method with such a name?
- *Extraneous token:*
Do we end up with a working program if we delete or alter the current token?
- *Generic error:*
None of the above applied.

Once our parser ends up at a token and detects an error, it attempts each of the above-mentioned diagnoses. If one of them applies we inform the user of the error and suggest an approach to resolve the error. If none of them applies, we issue a generic error message and show that to the user.

Cooperative Mode

Cooperative coding mode would allow pupils to work together in groups and force them to organize and to identify chunks of independent tasks. This helps their development by allowing them to practice communication and diagnosing errors in a group. The core feature in a cooperative mode would be, to let children submit their code to a shared document which holds all the code. Children could then load the shared document into their editor and choose which methods they want to include in their personal editor. In one use case, the teacher might decide to produce code which writes text onto the canvas in Logo with the whole class. Every pupil should write his name on the canvas and implement methods to write individual letters of the alphabet. Normally we experience massive speed differences among the pupils in the class. Children with short names and fast learners make faster progress than their classmates. In order to guarantee the success of every child we could distribute the work among all students. The children will immediately see, that there is no need to implement several programs which write the letter E for example. Instead they could work together, organize themselves, distribute the letters and reassemble their names once all letters are available. This way we can assure that all students will be finished at roughly the same time, no matter how long their names are and we can profit from positive social dynamics.

3.2.3. Responsive Graphical User Interface

The graphical user interface is of course another part of our project which heavily influences the experience the children have. A good UI might affect children to be more motivated, less confused and more successful. A bad UI affects the children to be less motivated, to be confused and to dislike the topic per se. A desirable feature of a user interface reacts to changing screen sizes, device types and orientation. In order to prevent the latter from happening, we explored a few prototypes, which can be compared against each other for varying screen sizes and orientations. When working with web-technologies we can expect our application to be run on various platforms and devices, the website can be accessed from desktop computers, laptops or even smart phones and tablets. In the normal case we will see the application in landscape mode, but since it is easy to resize the browser window or to rotate a smart phone or tablet, we have to be prepared that in some cases the website might be accessed in portrait mode. For this

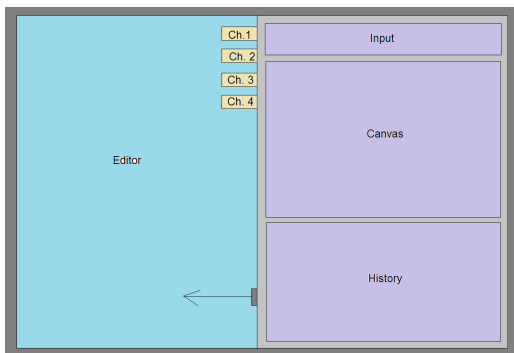
3. Requirements and Feature Analysis

reason we listed a few options for different layouts, which will be presented in the next two sections.

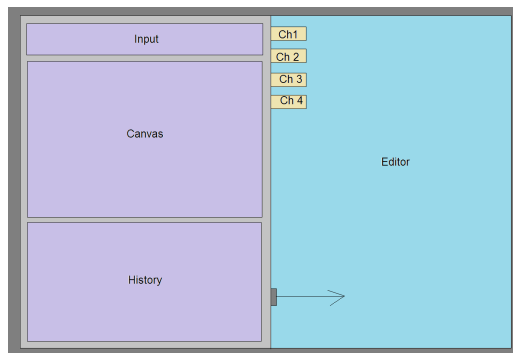
In landscape mode as well as in portrait mode we have to decide where to put the editor, where to put new editor entries on the editor, where to put the history and whether or not it should be resizable or not.

Landscape Mode

When deciding where to put the editor, we have essentially only two options in landscape mode: Either right or left.

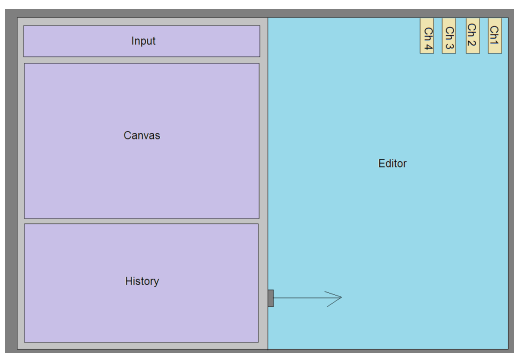


a) A: Editor on the left

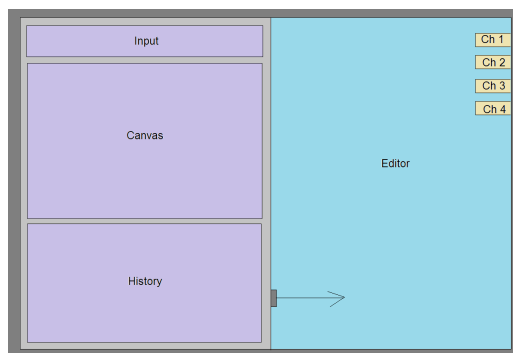


b) B: Editor on the right

New editor entries may be added horizontally to the editor or vertically.

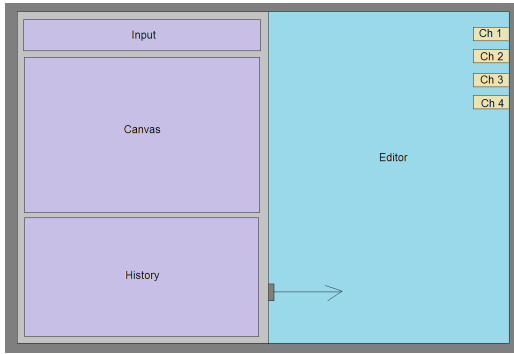


a) C: Post-its on top

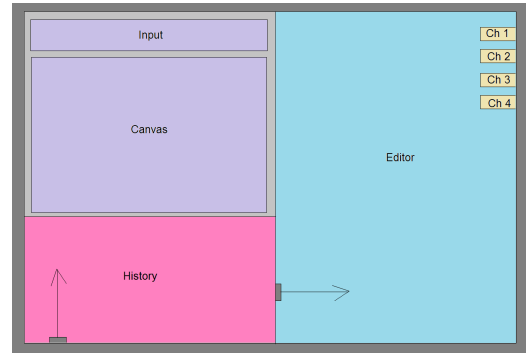


b) D: Post-its on the right

The history element could either be resizable or not:



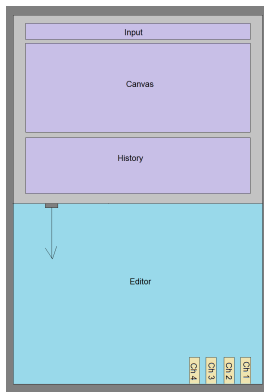
a) E: History fixed



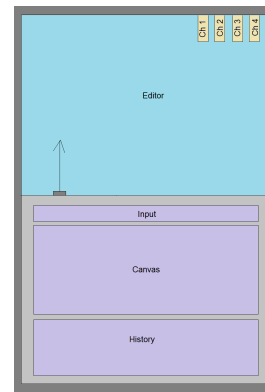
b) F: History resizable

Portrait Mode

In portrait mode we can distinguish the same options as before. We may choose where to put the editor:



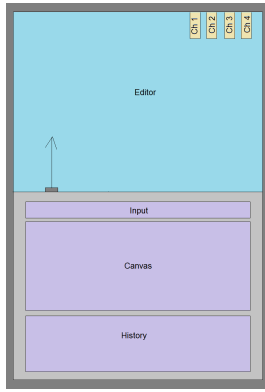
a) G: Editor on the bottom



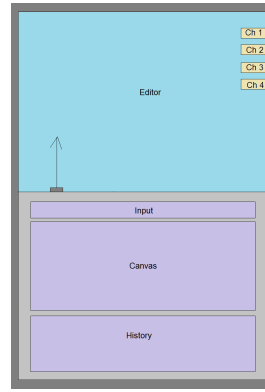
b) H: Editor on top

New editor entries may either be added horizontally or vertically. Note that in theory there are multiple ways of positioning the new editor entries. For example if we decide to add them horizontally, we still have to decide whether to put them on the inner or the outer side of the screen.

3. Requirements and Feature Analysis

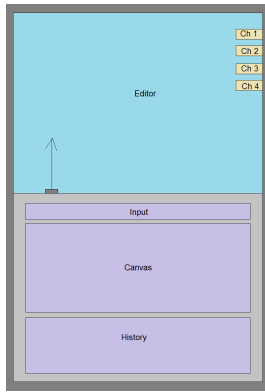


a) I: New editor entries on the bottom

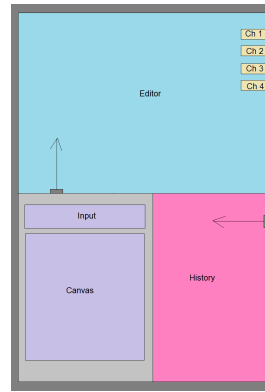


b) J: New editor entries on top

The history element may be resizable or not:



a) K: History fixed



b) L: History resizable

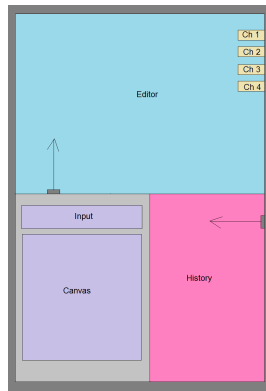
In this scenario one might also discuss whether it is truly the best decision to put the input field on top.

3.3. Restrictions

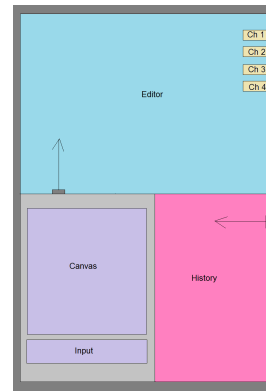
In this section we would like to discuss a few restriction we had to keep in mind when choosing technologies and implementing the website. The two main restrictions we had to keep in mind were that the website needed to be runnable either online and offline. Secondly we had to make a decision about which browsers to support.

3.3.1. Offline-Mode

Most primary schools in Switzerland do not have the facilities to provide internet access to laptops for the whole class. Some do, but others do not. We need to provide a suitable solution for both of them. Websites in general can be downloaded and executed on a local web-server too. Of course any communication to a server is impossible if there is no internet connection.



a) M: Input on top



b) N: Input on the bottom

But the browser is still able to execute code on the client-side. For this reason we decided to implement our website using only client-side technologies.

3.3.2. Mobile

A web-application can be accessed from various devices. Some of them are mobile. The difference between mobile devices and desktop or laptop computers are:

- *Different screen resolutions*: Newer devices have the same screen resolutions as TVs and screens. But older versions might have significantly less pixels, which could have an impact on the user experience, since the canvas will consequently be smaller.
- *Different screen sizes*: Most mobile devices are, for obvious reasons, significantly smaller in size than desktop computers. We work on a solution, which is independent of the screen size and the screen resolution.
- *Input using touch events*: Smaller screens are harder to handle. Very precise gross motor skill are required to hit a small button. We are not implementing any special features to facilitate the handling on smaller devices. Also we do not implement any gestures like swiping or similar, which are often used on mobile applications. Our intention is to implement a software targeting desktop computers. We try to achieve a nice visual experience on mobile devices, but we cannot guarantee a good user experience on mobile devices.
- *No hardware keyboard*: Last but not least mobile devices often do not come along with an external keyboard. Therefore an on-screen keyboard is shown whenever the user has the option to give input. The keyboard might occupy a lot of space and thereby create a bad user experience. We do not implement any additional feature to counter this problem.

3. Requirements and Feature Analysis

3.3.3. Browser Statistics

When writing a website, it is important to think about browsers. Not all browsers support all features and all technologies. For our web-application to be usable on as many browsers as possible, we inspected statistics about the usage of web browsers (the results are shown in 3.2). We found the following results which are calculated with the statistics for March 2016 [w3schools 2016]:

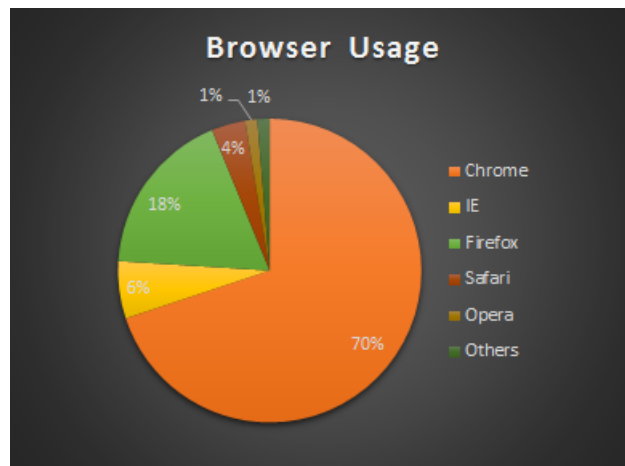


Figure 3.2.: Browser usage statistics

From the chart we see that today's most used browsers are: Chrome, Firefox and Internet Explorer. Keep in mind that the new browser, called Edge, is too new to appear on any of those statistics, introducing a little uncertainty for the evolution in future. When inspecting the three most used web browsers we analyzed which versions were used the most and found the results shown on the next page. Note that Apple devices are disproportionately strongly present in Swiss school rooms. Some of the schools using Apple devices use an iPad, others use MacBooks. The default browser on those devices is Safari, which is why we explored this browser too, even though it is currently (April 2016) not among the three most used browsers.

From these statistics we decided to focus on Chrome (C48 and above). Developing a web-solution, which is guaranteed to work on all browsers is rather cumbersome and we did let us to ensure that the website will be properly working for 70% (state: March 2016) of users who use some sort of browser to access the internet.

In a statistical evaluation (depicted in 3.3), we found that 43% of all internet users are accessing from a Windows 7 platform. Another 20% are Windows 10 users and 15% are Windows 8 users. The remaining 22% are users accessing from a Windows XP, Linux, Mac or Mobile device [w3schools 2016]. These values represent the statistics for the overall usage of the internet. In our Logo projects the statistical distribution of different platforms is different. Windows is used in most projects. The percentage of Linux users is drastically lower and the percentage of Mac users is drastically higher.

Safari is, besides Windows, one of the most used platforms in our project. Therefore we investigated on the distribution of users on the different versions of this browser. The results (shown in 3.4) makes clear, that most users access the internet with the most up-to-date safari browser,

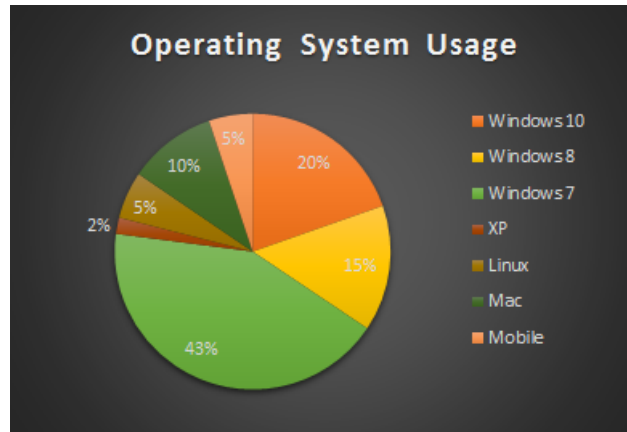


Figure 3.3.: Operating System Statistics

namely version 9. In March 2016 up to 75% of all Safari users were accessing the browser version 9. Only 8% were using the browser version 8. The remaining 17% were using version 7 or below:

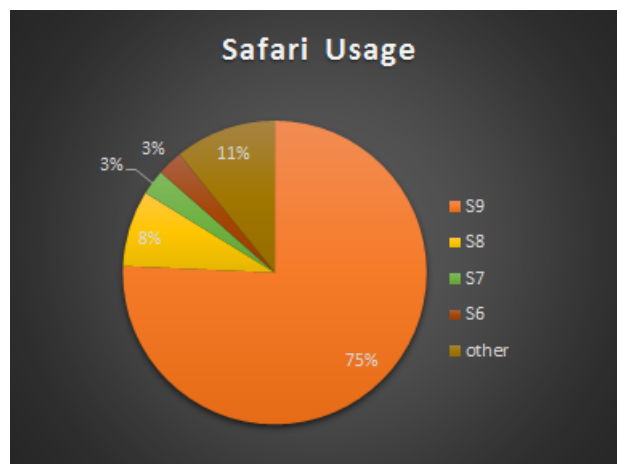


Figure 3.4.: Safari usage statistics

3. Requirements and Feature Analysis

Most Chrome users access the internet with a browser version 48 or 49. Only 10% of all internet accesses using a Chrome browsers do not use version 48 nor version 49 (see 3.5):

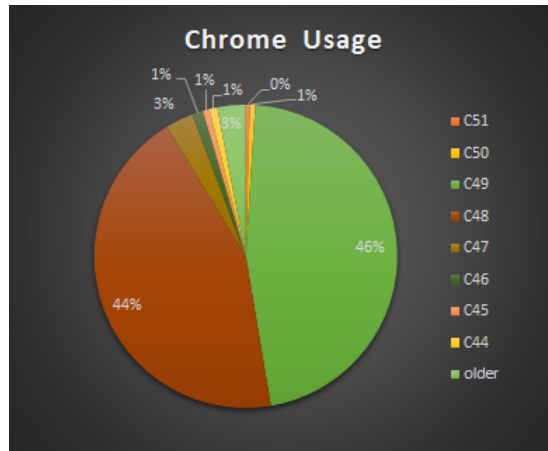


Figure 3.5.: Chrome usage statistics

Firefox users access the internet with browser versions 47, 54 and 44 the most. Those three versions make up 85% of all internet accesses using Firefox (depicted in 3.6):

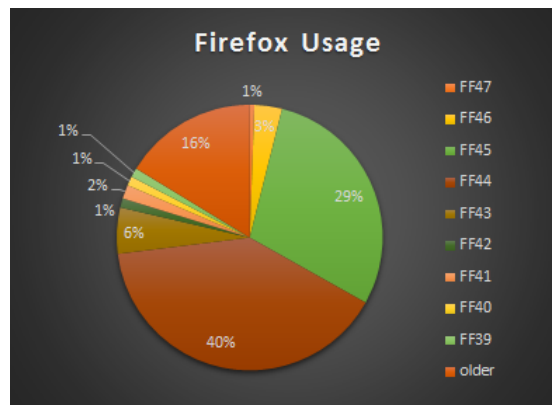


Figure 3.6.: Firefox usage statistics

Windows used Internet Explorer as default browser until Windows 10 was released in July 2015. In Windows 10 the default browser was set to be Edge. We combined the statistics for those two browsers. We observed that 67% of all users accessing the internet through Internet Explorer and Edge in March 2016 were using Internet Explorer version 11. Another 17% are using Edge version 13 (values visible in 3.7).

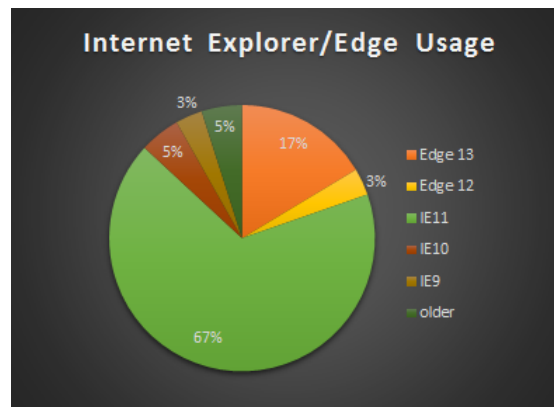


Figure 3.7.: IE/Edge usage statistics

To conclude the contents of this chapter we state, that there are many different browsers, on different operating systems on different devices. We decided to focus on Chromes most frequently used versions in our project but we were constantly checking on various browsers (Firefox, Internet Explorer and Edge) too.

3. *Requirements and Feature Analysis*

4

Design

In the last chapter we argued that web technologies provide a suitable platform for programming environments. The core concepts of our web programming IDE rely on the principles of web technology. For this reason we will explore the history of web technology and web programming IDEs and explain several design choices for our programming environment in this chapter. A few possible alternatives to our final prototype have been designed and we will discuss the trade-offs we were confronted with in terms of the system design.

4.1. History of Web Development Technology

The internet has grown a lot in the last twenty-five years, leaving a huge connected net of websites and web-apps which can be used on a wide variety of browsers, operating systems and machines with different settings and behaviors. Thanks to a continuous development of the available technologies to the web and a perpetual adjustment of the capabilities of browsers we encountered a massive progress in possibilities with web technologies. We would like to give a short overview of the key milestones during the development of web technologies to the point we know currently:

In the year 1991 the *hyper text transfer protocol* was invented, providing the basis for a new kind of communication on a network of distributed computers, which represent nodes in a global network. Shortly thereafter the first version of HTML (hypertext markup language) was created, enabling web-developers to describe the contents which should be displayed to the user via a browser. HTML is, along with JavaScript and CSS, a cornerstone technology used in web development. By 1995 five browsers were invented, namely: Mosaic, Netscape, Opera and

4. Design

Internet Explorer. By 1997 cookies, SSL, Java, Flash and JavaScript enabled us to send tiny data packages from the server to the client. In order to keep information about the user even after a page refresh we have three possibilities: Either we encode information in the URL, in a cookie or in a session on the server. The former two are visible to the client, whereas the latter consumes memory on the server.

Cookies allowed us to store the user's activity on the web. The DOM (document object model) is a programming interface for HTML and XML, which holds all elements, which are presented on the website. JavaScript is a programming language which can access the elements on a website and change their state by accessing the DOM. JavaScript was a cornerstone technology used to manipulate the DOM of a website.

By the year 2000 a few new technologies came along: XML (which is a superset of HTML) provides a broader way to represent and interlace data, which can be used in various ways to transmit data over the internet. CSS is the third cornerstone which became an almost omnipresent technology used predominantly in web-apps and websites to define the style of the view. AJAX is a technology used to retrieve data asynchronously when reloading some part of the website. Being asynchronous is a crucial attribute when reloading data, because being unresponsive would be an unsuitable burden for the user. Web-fonts allowed the browser to display a wider variety of fonts. This (together with the invention of CSS) was the first step towards a more visual appearance on the web.

Two new browsers were created in the years 2003 and 2004, namely Apple's Safari and Mozilla's Firefox. In the same year as the latter was published, HTML's newest element was added, the canvas. The canvas was another big step towards a more graphical web appearance. In the year 2008 the last, currently well-known browser was published, Google's Chrome. Since then a lot has happened: HTML5 came out, bringing web technologies closer to the hardware, allowing for example accessing the geolocation of a device. In the same time mobile phones became more powerful, leading to a completely new market for web-apps. A technology for web-apps to be used in an offline mode, was introduced, as well as support for touch-events as well as drag-and-drop.

In the year 2012 Microsoft announced the first version of a new programming language called TypeScript [S. Tobin-Hochstadt 2008, TypeScript 2016]. TypeScript aims to become the next version of JavaScript, supporting classes, inheritance, generics, interfaces, while being a strongly typed programming language. For JavaScript being no strongly typed language it was difficult to create correctly working large code bases in JavaScript. At the moment Microsoft's TypeScript is transpiled to JavaScript before being executed on the browser. Only last year, 2015, Microsoft exchanged its default browser on Windows 10 from Internet Explorer to Edge, leaving us with the following five most used browsers at the time of writing: Chrome, Firefox, Internet Explorer, Safari and Edge.

Based on this evolution of web technologies [Bonk 2009] we had to decide for a concrete set of technologies to use in this project. We will present the available choices in the next section.

4.2. Design Choices

In this section we would like to present the key design decisions we made in order to implement our Logo programming IDE. First, we will discuss several alternatives in the design of various aspects of the application. Later, we will proceed by presenting our actual design choices and the trade-offs which lead to our decisions.

4.2.1. Overview

The design choices we intend to discuss in more detail in the following subsections are *design patterns*, *implementation language*, *code editors* as well as *rendering*. In Figure 4.1 you find an overview over the mentioned design options, their central role in the application and a few alternatives we had to decide upon.

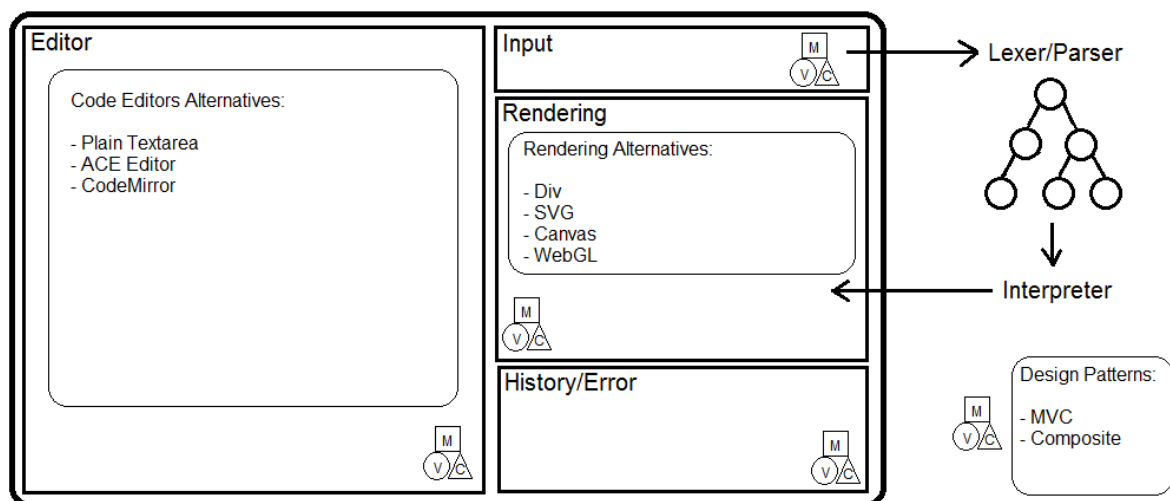


Figure 4.1.: High-level Overview: Design Choices

4.2.2. Frameworks and Design Patterns

The choice for or against a specific design pattern for the system design is coupled with the decision for a particular framework. Usually these frameworks (i.e. toolsets or libraries, which provide skeletons to build larger programs on top) are built with a concrete intention and thus force the use of a set of design pattern which are well suited for their particular purpose. In the context of web-applications a multitude of frameworks are available, each of which has different properties and uses different design patterns. Their implementation impacts the architecture in the high-level perspective. We explored the most popular frameworks:

1. *AngularJS*

AngularJS is a framework by Google for creating single-page-apps with MVC. It supports

4. Design

some innovative ideas like two-way data binding (for connecting model and view) or templating for creating custom HTML-tags with a certain behavior. The basic language in AngularJS is JavaScript. It is a framework supporting most of the well-known JavaScript libraries like jQuery or CoffeeScript. This framework can be used to create standalone web apps too.

2. *Angular2.0*

Angular2 is an improved new version of AngularJS and is thus also used mostly for generating single-page-applications in a MVC fashion. In spite of being the successor of AngularJS, Angular2.0 differs from its predecessor in various aspects. The most prominent difference is, that the core language in Angular2 is TypeScript (a statically typed superset of JavaScript), not JavaScript. Moreover it is completely component-based. The code for controllers, models and views are all written in the Composite design pattern.

3. *ReactJS*

ReactJS is a framework published by Facebook. Its purpose is to simplify handling the view part in a MVC designed piece of code. So in contrast to AngularJS and Angular2.0, which handle the model, the controller and the view, React provides support for writing the view in a comfortable way. Its main advantage is, that it is able to reduce the overhead working on the DOM if it is not necessary, due to the fact that data may stay unaffected by some internal operations. It uses a virtual DOM, so thinking about implications to DOM for every operation is not necessary anymore with ReactJS.

4. *Polymer*

Polymer is a tool made by Google, which can be used in JavaScript for building custom components. Web components are a new design philosophy to keeping the view modular. Instead of just having one HTML file containing everything, custom HTML-tags can be instantiated, which are tailored for a specific purpose in our application. This framework makes use of the Composite design pattern.

5. *Meteor*

Meteor is a fullstack (client-side and server-side) JavaScript platform for building web-applications, which are designed to be simple. Meteor is built on top of Node.js in the back-end. It uses two-way data-binding by default, which means that any changes in the data in the database will be displayed on the website immediately.

Of course using no framework at all is always another option, which is possible. When not using any frameworks (or when deciding to use Angular2.0, as we did), we find in general two very prominent design patterns. We use a combination of the following two patterns in our system design which was enforced by the Angular2.0 framework:

- For applications with a graphical user interface (which is usually the case in web-development), the MV* architectures are very popular and well-established choices for the system design. The reason for their popularity is an idea called *separation of concerns*. This means that the graphical part, the system model and the controller should be as well separated from each other as possible. Solely the controller is supposed to interact with the model and the view. It manages the whole application and processes the user's input.

- Beside the MV* design architecture there is another design pattern which is of interest in web development, especially when working with hierarchical structures: The so-called Composite design pattern. It structures hierarchies as trees. Using trees allows us to structure the internal communication between independent components differently. Information can be forwarded to components selectively.

We combined the two design patterns in our web-application. Using a concept called *templating* we started to group visual components of the graphical user interface into components, which were then interconnected into a tree. Figure 4.2 depicts the components and their interaction in the tree. At the root of our tree structure we find a component called App. This component is the most general one and it simply includes the whole windows. Its children are three components, namely the header part, the editor and another component including the input area, the canvas and the history, which in turn are components on their own.

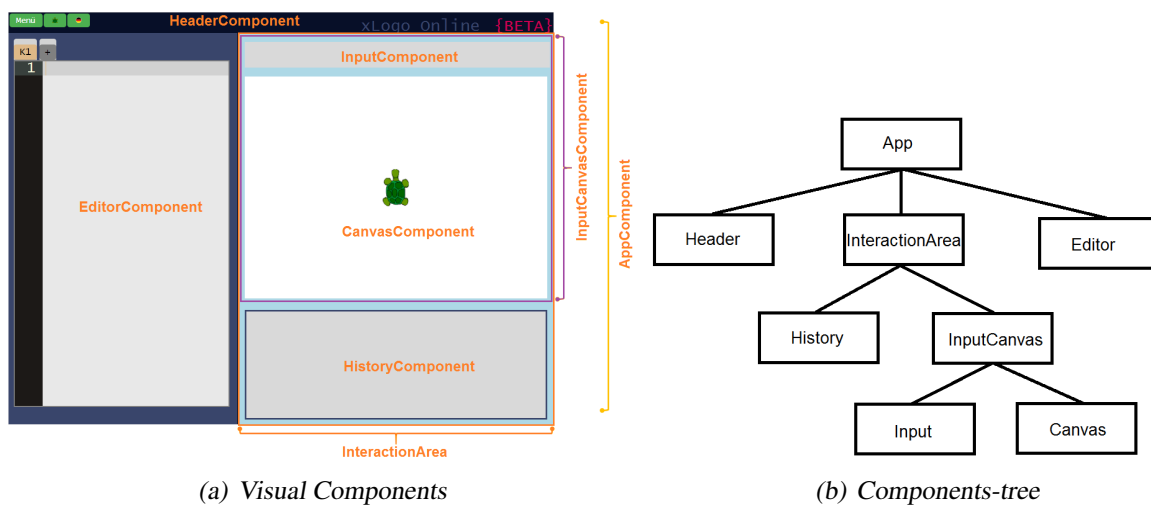


Figure 4.2.: Composite pattern in our web-application

On every component (i.e. every node in the tree) we apply the MVC design pattern paradigms. That is, we handle the view, the model and the controller in a component-wise fashion. Obviously it is not always possible to provide separation of concerns, especially in the communication between parent- and children-nodes in the tree structure. This question is handled by Angular2.0, the framework we used in our web-application, in a way such that the root component is informed whenever an event is triggered. The root component may forward the information to its children, who in turn forward it to their children.

4.2.3. Implementation Language

Before exploring the spectrum of different implementation languages, we would like to discuss the trade-offs of working on a server or rather directly on the client. We state a few properties which characterize the two cases:

- *Server-sided web development:*

Working in a server-side environment means, that the client is provided with whatever he needs from a server. The communication between server and client is based on HTTP requests, which can be triggered upon user inputs (e.g. interaction with the user interface on the client-side). The underlying communication is subject to some physical bandwidth limitations. Also the server can become a bottleneck in case he has to handle many requests by several clients at the same time. Also this solution works only under the premise, that the connection to the server is given at all times, which means that an Internet connection needs to be available. A few advantages of working on the server are, that there are quite few restrictions towards the programming language, which is used on the server. For this reason there are a plethora of different programming languages, which may be used on a server. The most common ones are: Java, C#, Python, php, and JavaScript (Node). Also working on a server allows us to store the user's data persistently, which is not really possible when working on the client exclusively. The code cannot be seen or manipulated by the user.

- *Client-sided web development:*

Working on the client directly means, that we use a programming language, which can be parsed by the browser directly. At the moment of writing JavaScript is de-facto the only programming language which is fully supported on (almost) all modern browsers [Flanagan 2014]. Most browsers understand JavaScript itself, and all frameworks, which are transpiled to JavaScript, natively. The whole DOM (e.g. all components on the website, their properties and handlers) is directly available in JavaScript and thus it is possible to react to user inputs without a detour via server. The fact that every client is able to operate on his own with almost no interaction with the server, implies, that this solution scales better with a large amount of users. There is no risk of bandwidth issues in this scenario. Moreover, it is even possible to run the application without a web server and Internet at all, as long as the code is physically available on the client's machine. On the downside, having the code on the client's machine implies, that the user is able to see and manipulate it. Also it is not possible to store data persistently in the website, beyond the point of a page refresh. A few alternatives to the mainstream programming language JavaScript for client-side web development are TypeScript and CoffeeScript.

We decided for a client-side web development approach. The reasons for this decision are, first of all that we cannot ensure the availability of a stable Internet connection in all use cases. Since we intend to create a web-application which can be used in an online and offline manner, we are on the safe side to work directly on the client. In addition to that we want the web application to scale even with a possibly very large amount of clients, who try to run the application simultaneously. In this category we have three prominent programming languages, which could be used to implement the application, namely: JavaScript, TypeScript and CoffeeScript. In the next section we are going to present the implications of using either of these three languages.

Client-side programming languages

When speaking about client-side web development, the first thing which pops up is JavaScript. That's because JavaScript is the only programming language which is built into most modern browsers. Besides JavaScript there are a few frameworks and other languages, which are transpiled to JavaScript and can then be run in the browsers. We would like to present the advantages and disadvantages of choosing JavaScript, TypeScript or CoffeeScript as implementation language.

- *JavaScript*

JavaScript is a structural, dynamically typed programming language, which is widely known as the programming language designed for web development [Flanagan 2014]. By definition a structural language does not feature static type declarations, or classes, but rather groups objects at runtime according to their functionality. Each object has a dynamic type, but at compile time there is no information about the types available. This can be very handy, when working with the DOM - we are able to do duck-typing, which is a very convenient, flexible way of programming. At the same time it is quite insecure and (especially for larger projects) error-prone to work with a dynamically typed language. Static types are a means of ensuring more structured and clearer code. JavaScript operates on the DOM model directly. The DOM provides a prewritten set of basic functionality, event handlers and properties for all its elements. Due to the fact, that JavaScript runs locally it comes along with a few security issues. In principle JavaScript could be used to exploit the user's system. The browsers try to suppress possibly malicious behavior by not allowing a few operations, like for example overwriting local files or loading local files directly into the website. Different browsers react differently, which is another issue with JavaScript. We cannot guarantee that the rendering and basic behavior will be the same across all platforms and browsers. In addition to that, JavaScript is single-threaded, which makes it more difficult to run code asynchronously.

- *TypeScript*

TypeScript is a new programming language, which is transpiled to JavaScript. It is based on the ECMAScript6 syntax standards, and will, once all browsers apply the new standards be directly executable on browsers. At the moment TypeScript can be used already, because ECMAScript6 is a superset of ECMAScript5. TypeScript is, in contrast to JavaScript, statically typed. This allows for a more safe way of programming, which is useful especially in larger projects. TypeScript supports polymorphism, inheritance and generics. Those features are very helpful to create a better abstraction. A disadvantage is, that TypeScript is currently not able to integrate pure JavaScript code as is. It needs to infer the static types, which is not possible, because all non-trivial variable types are inferred to *ANY*, the most generic type. This is problematic, because the TypeScript compiler is not able to find the methods used on those objects. In order to prevent this, declaration files need to be created. Declaration files state about the shape of a JavaScript library, e.g. methods which are available, their parameters and return types. An example for such a declaration file is the following:

4. Design

Original file:

```
1 myVar;  
2  
3 class MyClass {  
4     method1() {  
5         myVar = method2(true, "World");  
6         return "hello!";  
7     }  
8  
9     method2(p1, p2) {  
10        return 0;  
11    };  
12  
13 }
```

In the original file we do not find any static type annotations, thus TypeScript needs to infer their types, which may be fairly complex. There are times, when the compiler simply assumes the type to be ANY.

declaration file:

```
1 declare module "SomeClass.js" {  
2  
3     myVar: number;  
4  
5     class SomeClass {  
6         method1(): string;  
7         method2(p1: boolean, p2:string): number;  
8     }  
9 }
```

In this declaration file we added to the method and variable declarations additionally the type information about argument types, return types and variable types.

- *CoffeeScript*

CoffeeScript is a language that compiles into JavaScript. The language itself is based heavily on JavaScript, which makes it necessary that the programmer is already familiar with JavaScript when writing CoffeeScript. It improves on a few flaws, which were known from JavaScript (like for example the lack of a construct to create objects in a clean and intuitively understandably way, an object existence check or array comprehension). It has a some drawback too, though. First there is no proper support for linters (which is used to mark suspicious usage in system software, which is likely to be a bug). Also, the language is single threaded, as JavaScript is, and thus does not provide a means of executing code asynchronously. And lastly, it is not clear, whether CoffeeScript will be continued, because most of the JavaScript flaws, which were solved in CoffeeScript have been solved in ES6 (a new syntax standard from 2015 which is used by JavaScript) too.

4.2.4. Code Editor

In this section we would like to discuss a few alternatives we had when choosing a web component for code editing. It is a central part for our application and will be used by the user intensively. It should allow as much support for user feedback as possible. There is a very wide range of libraries which provide text editors with additional functionality. We found the following three editors to be the most interesting. And we are going to explain their properties and the reason why we find these the most interesting ones now:

- *Plain text area*
The plain text area is the option, when no library is available. In a first prototype we implemented the web application using a plain text area. In Figure 4.2 (a) you see a plain text editor on the left. It provides the most basic features like displaying (possibly multi-line) text and scrolling. Everything beyond this (like line number, text highlighting, text formatting and error highlighting) is not available.
- *ACE editor*
The ACE editor is an open source JavaScript library, which is a Sublime/Microsoft Visual Studio clone, which can be integrated into basically any JavaScript web project. It is free and underlies the New BSD License which imposes minimal restrictions on the redistribution of their library. It is supported by Firefox 3.5+, Safari 4+, Chrome, IE 8+ and Opera 11.5+. And last argument, which makes this text editor interesting is, that another library, called Firebug, uses ACE as their basis to implement a cooperative text editor. This is a feature, which we would like to implement in the future, see Section 7.2.2.
- *Code monkey editor*
Code Monkey is a free, open source, JavaScript library as well and its appearance is based on a plain text area. It underlies the MIT license, which is also very permissive and thus complies very well with other licenses. It provides support for Firefox 3+, Chrome, Safari 3+, IE8+ and Opera 9+. Therefore it is compatible with older browsers than ACE editor does. But on the downside, there is no library for cooperative text editors, which uses Code Monkey.

We decided to use the ACE text editor, due to its good compatibility with other libraries and as a basis to implement cooperative coding on top in the future.

4.2.5. Rendering

The web application is based on a graphical user interface with an input field, a canvas, a text element for command history as well as a larger text element as editor. We implemented the canvas using two HTML5 canvas elements. One canvas is used to display the direction and position of the turtle, while the other one is used to display the drawing. Other options for implementing the painting would be to use a div or a SVG element. We decided for a HTML5 canvas element due to following factors:

- *Div*: Divs are supported on all browsers and provide backwards compatibility. But they are very clunky and their performance is bad when loading many elements. Moreover

4. Design

divs cannot scale, which is essential in our application, except when using a hardware accelerated css engine and thereby offloading heavy graphical computations onto the GPU. This would have a direct influence on the battery lifetime because it consumes much more memory. This is not an option, since we are expecting to use the web-application on laptops in classrooms, where it might not be possible to provide a socket for every student.

- *SVG*: SVGs are vectorial and easier to implement. Still we decided for Canvas due to the following drawback: SVG elements are used to describe 2D-pictures in XML. Therefore every element in SVG is stored in the DOM, which implies massive DOM-manipulations in case we write a large Logo program. Also we work towards a solution in which it is possible to work on the same canvas in cooperative mode with the whole class. For such a scenario it is inevitable to keep the DOM small, since we need to share its state, which is very time-consuming and difficult if it is too large. Using SVG in such a scenario would imply a massive performance issue, because it would take a lot of time to keep track of all DOM-manipulations and to update them accordingly, in the right order, without compromising the state of the DOM. This would increase the effort for rendering a lot. In conclusion: SVGs are slow when rendering many elements and they pollute the DOM. Also SVGs are vector-based, while logo programming is inherently pixel-based. The resulting images might look better when using SVGs, but they behave counter-intuitively. The images are supposed to be pixelated a little to convey the concept of pixels.
- *Canvas*: Canvas are useful when drawing pictures on a pixel base, without the need for event handlers, text rendering or support for very large pictures. Depending on the browser the maximum area we are allowed to fill is around 300 megapixels (which corresponds to roughly 17000 pixels in height and 17000 pixels in width). We do not use anti-aliasing in this web-application, because it is an important aspect of the project to experience pixels. Screens supporting 4K have 8.3 megapixels, with 8K we have images with 44.2 megapixels, which is still far from 300 megapixels which is the limit of canvas. Note also, that most schools are still using 720p or 1080p screens anyway. The canvas itself is a single DOM element, no matter its content, it does not influence the DOM. This is an important feature, when working towards a cooperative mode, in which the DOM is shared. Also the application itself is inherently pixel-based rather than vector-based. In that sense we can argue that it makes more sense to rely on canvas (which might lead to slightly pixelated images sometimes), rather than on a vector-based version in SVG, because it behaves counter-intuitive in point of view of the learned concepts of pixels. The application is supposed to use pixels since it is a part of the concept which is transferred to the children. Additionally a canvas can be saved as a .png or .jpg image, which is something the application needs to support. In our web application one step of the turtle fills roughly one pixel.

4.2.6. Libraries, Tools and other frameworks

When it comes to libraries for web development you will find an extremely versatile spectrum of different technologies, which allow us to work in a smoother, less cumbersome way. There are JavaScript libraries like jQuery, Dojo, Prototype, Mootools, which help us writing

JavaScript code. We will not use any of those tools, due to the fact that we will use TypeScript in our project, which makes any of those libraries obsolete. But there are other libraries with interesting utility, which we will list below:

1. *ANTLR4*:

ANTLR (Another Tool for Language Recognition) is a tool which can be used to generate lexer, parser and visitor for a given grammar [Terence Parr 2016, Parr 2013]. In its newest version 4 ANTLR can generate sourcecode for various languages. One of them is JavaScript, which can be used in the web-application. Building and traversing a parse tree is simplified by this tool a lot. ANTLR is used often when working on a project containing a compiler or a transpiler [Gross 2012].

2. *ReactJS*:

ReactJS is a library published by Facebook, which can be used for real-time notifications. Whenever there is new information available for the user, ReactJS pushes the updates to the user. This might be useful especially when interaction between multiple users occur.

4.3. Prototypes

Over the time a handful prototypes were developed with the purpose to deliver the advantages and disadvantages of a certain technology while still being rather small and manageable. We decided for a few of these technologies after developing three major prototypes with or without the use of a additional framework. We will give a short overview in the following sections, presenting the technologies we actively tested.

4.3.1. Pure HTML5, CSS3, JavaScript

The first prototype was a pure HTML-CSS-JavaScript solution, which was also used for the authors to learn the basics in web development in general, since they were not familiar with any of those technologies before.

The design involved a HTML5 document containing the information about HTML elements, we used for this prototype. The structure of the website was quite simple. We had the following setup in mind:

4. Design

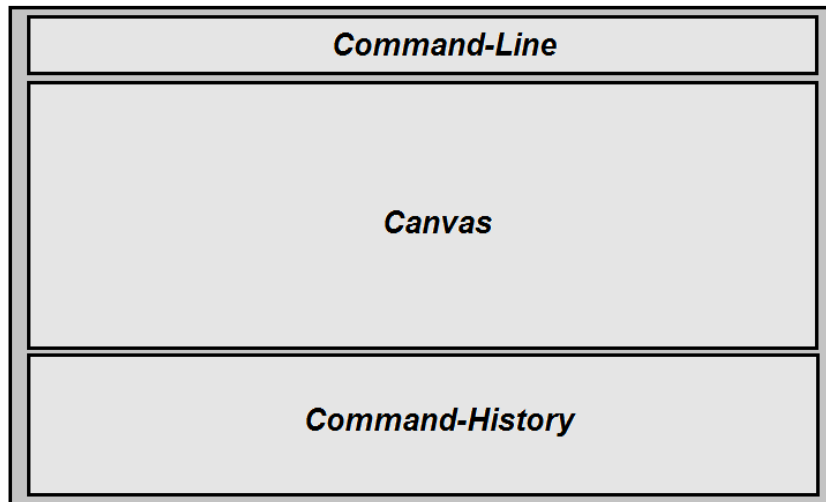


Figure 4.3.: Mockup for XLogo Online

On top of the website a command-line waits for the user to enter a command. This element was realized by a `HTMLInput` element. Once the user entered a command and confirmed by pressing *ENTER*, the commands are stored in the command-history on the bottom of the website. Using the arrow-key (*UP*, *DOWN*), it is convenient to retrieve earlier commands and to traverse the list of previously committed commands. This element was realized as a `HTMLAreaElement`. Browsing through previous commands was easily implemented using a bit of JavaScript code. Upon pressing *ENTER* and committing a command a parse tree was built too. This parse tree could be walked and simultaneously executed piece-wise. The parse tree can be built by hand, since the language is rather small. Instead it is also possible (and this is what we decided to do in the end) to use a tool called ANTLR, which generates the lexer, tokenizer and parser for us.

In the end our prototype looked like this:

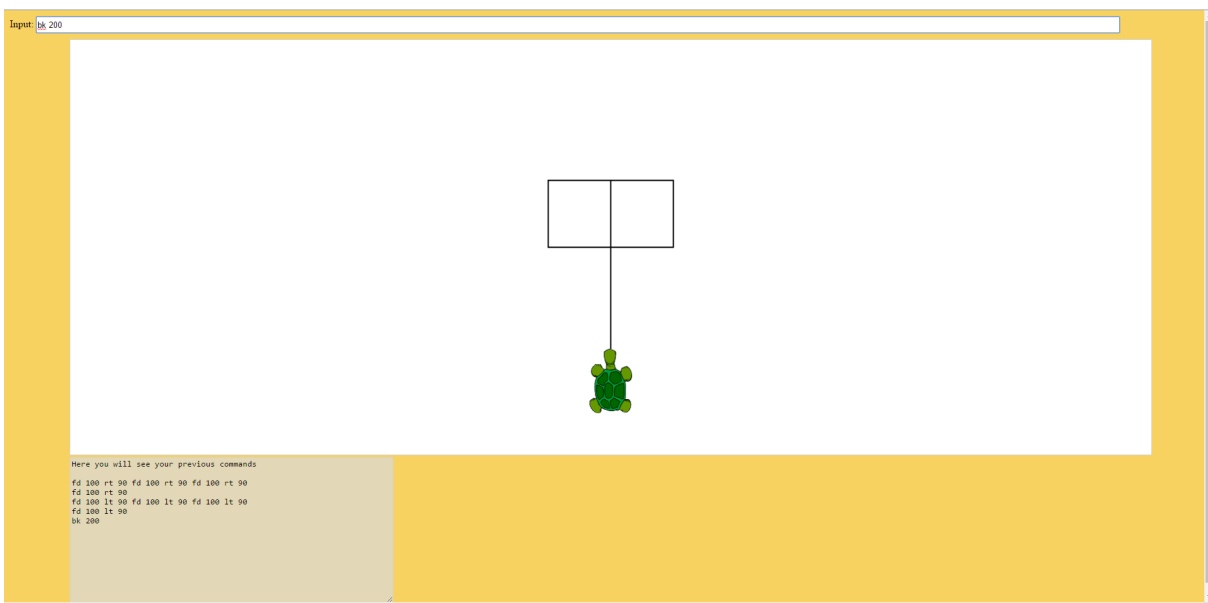


Figure 4.4.: First prototype using HTML5, CSS3 and JavaScript only

We encountered the following characteristics for this solution:

1. The pure HTML, CSS, JavaScript solution shines in terms of flexibility for the developer. There are no restriction whatsoever. It is possible to put declarations of the appearance of some element into the CSS part (where it belongs), or in the HTML part directly to the declaration of the element itself. We are even able to change the appearance of some element dynamically, in the JavaScript part. Also, JavaScript itself excels in terms of flexibility. Being a weakly typed, dynamic language, we are able to do whatever we want in JavaScript.
2. The previous point can be seen in two ways. Either we can state the flexibility given to us as an advantage, or we can argue that too much flexibility can affect the safety. We are in favor of a language which is clear, flexible, expressive and safe. The second and the third point are met by JavaScript for sure, whereas the first point depends on the experience of the developer and the safety is clearly not met in JavaScript. We are not informed about typing issues at compile time. We might be able to notice some issue at runtime if we are lucky. This is a disadvantage especially for large projects.
3. JavaScript lacks static types, which affects the style of programming, which can be done in JavaScript. Since an argument passed to a method cannot be distinguished in terms of its type, we cannot use a language construct called *method overloading*, which is present in other popular languages today. Method overloading is just syntactic sugar, since we can always call a method a little different and then call whatever method we need. Still it makes certain design patterns much less clear. One of those design patterns is the *visitor pattern*, which is very useful when traversing a parse tree for example.

4.3.2. Antlr4

ANTLR is a tool, which can be used to generate a parse tree from a given grammar. The grammar we used in the prototypes was a small subset of commands, which can be used in XLogo. XLogo features a few commands to steer a turtle around, leaving a drawing behind it. The language knows several commands, which can be used interchangeably. For example there are two commands *FD* (forward) as well as *BK* (back) for moving the turtle forwards and backwards. For obvious reasons we can use one of these two commands (along with *RT* or *LT*) exclusively, simulating the behavior of the other.

The same holds for the commands *RT* and *LT* (left), which induces the turtle to turn to the left or to the right. These commands can be reduced to one another as well. The language XLogo has a few control structures too, like the commands *REPEAT*, *WHILE* and *IF*. The control structure of *REPEAT* works similar to a for loop. In our mini-grammar we included the *REPEAT* command only. Finally we allowed a command to clear the screen *CS*. This led us to the following mini-grammar for our prototypes:

4. Design

```
1 grammar Logo;
2
3 prog
4   : (cmd)* EOL?;
5
6 cmd
7   : repeat
8   | fd
9   | rt
10  | cs;
11
12 repeat
13   : 'repeat' number block;
14
15 block
16   : '[' cmd + ']';
17
18 fd
19   : ('fd' | 'forward') number;
20
21 rt
22   : ('rt' | 'right') number;
23
24 cs
25   : 'cs'
26   | 'clearscreen';
27
28 number
29   : NUMBER;
30
31 NUMBER
32   : [0-9] +;
33
34 EOL
35   : '\r'? '\n';
36
37
38 WS
39   : [ \t\r\n] -> skip;
```

This grammar can be used to generate a parse tree in ANTLR, which then can be traversed by a visitor or a listener. ANTLR4 is capable of producing the parse tree in multiple target languages, JavaScript being one of them. We examined the generated parse tree and included it into the prototype explained in the previous section.

4.3.3. Pure HTML5, CSS3, TypeScript

TypeScript is a language developed by Microsoft, with the purpose to replace JavaScript one day. For backward-compatibility reasons TypeScript is a superset of JavaScript. This means that JavaScript code is valid TypeScript code. The part which exceeds JavaScript involves quite a few useful concepts. TypeScript is a strongly typed language, which helps us to develop correct code, especially if the codebase is large. Among others, its set of novel features and syntactic structures includes classes, interfaces, generics and mixins.

TypeScript itself can be used to replace JavaScript only, HTML5 and CSS3 still are needed for the whole application to work. For this reason our prototype, which was implemented in TypeScript, HTML and CSS looked the same as the prototype using JavaScript instead. The HTML5 and CSS3 files could be reused from the prototype we presented earlier (see 4.3).

We would like to reflect on our experience with this prototype.

1. In comparison to the JavaScript prototype we encounter much more safety in this prototype. Due to TypeScript's strong typing we get static safety, which is a great deal in a larger project.
2. Also classes, interfaces and generics are very useful when developing a large application which contains this kind of structure. The project we developed involved a few components, which can be completely decoupled from each other. In JavaScript this decoupling is cumbersome, but in TypeScript it is easy to establish such a decoupled structure. In our prototype we have both classes and modules, which allow us to provide a nicely decoupled model.
3. TypeScript is novel and thus still futuristic. In the future we will encounter a change in web development technology and the improvement of the core language JavaScript will be a part of this development. TypeScript (or another language related to it for this matter) will take its place. In order to examine the newest web technologies TypeScript is definitively a part of it.
4. In terms of performance we can expect this solution to be just as fast as a pure JavaScript solution. Currently TypeScript is first transpiled to JavaScript and then executed in the browser. After the compilation a TypeScript solution should not be much faster or slower than a comparable JavaScript solution.
5. Choosing TypeScript does not make us suffer a lost in flexibility. We are able to use any kind of design and implement it in TypeScript. For our prototype we chose the same model as our JavaScript prototype and thus we encountered the same issues. We realized that it would be better to keep unrelated concepts strictly apart. It is a good thing to divide the website into multiple independent parts which can be developed and tested individually and in isolation.
6. Another downside is, that including JavaScript files into a TypeScript project and using it in a TypeScript file is critical. JavaScript has no types, whereas TypeScript variables are strongly typed. If there is no type-declaration, type inference is applied to determine the most precise type a variable can have. In JavaScript type inference cannot determine

4. Design

what kind of HTML-element we are interacting with, which is problematic since most methods used on HTML elements are not applicable on a generic `HTMLElement`, so TypeScript's type checker will complain. In order to resolve these issues we need to create a declaration file, containing type information for variables, arguments and return types in a JavaScript file. This introduces a lot of effort, which cannot be avoided if we wish to use the JavaScript files generated by ANTLR. An alternative would be to write the parser ourselves.

4.3.4. Angular2

Angular2 is a new framework published by Google, which uses TypeScript. The framework helps us to decouple various modules in the website from each other. Using component-based design conceptions, we can create custom HTML-tags. Angular2 helps us to distinguish model, controller and view. AngularJS is a framework, which does the same, based on JavaScript and is the predecessor of Angular2. Since Angular was very popular we can expect Angular2 to become a great deal as well. This new design paradigm forces us to connect HTML, CSS and TypeScript files in another way than we did in the prototypes we had before. Therefore it is not possible to reuse parts of the previous prototypes (especially the layout or HTML files) in this prototype.

This is what the prototype looked like in the end:

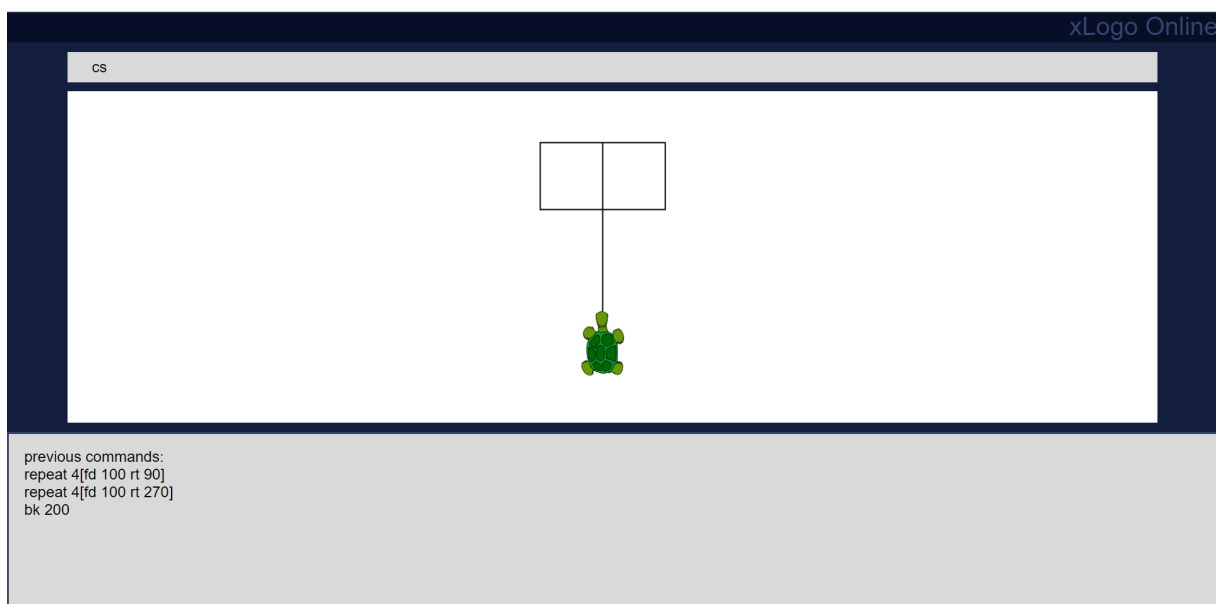


Figure 4.5.: Prototype using Angular2

Now we would like to point out some of the characteristics of this solution. After this section we will conclude this chapter by making a statement about the design choices we made:

1. AngularJS and Angular2 are frameworks to write single page applications. The key idea is, that there is only one HTML file to be loaded. In case the user wants to create the

illusion of multiple pages, we need to change the DOM to achieve the illusion of multiple HTML files.

2. In regard of the issues using JavaScript files in TypeScript, we faced the same issues as in the previous solution. If we want to use the generated JavaScript files containing the parser, we need to create a declarations file, telling the TypeScript document what types certain methods take and return, in order to help the process of type inference.
3. In this prototype we encountered a clear decoupling between model, controller and view. We were able to create custom components, containing just a part of the functionality of the whole website, which can be tested nicely. On the other side this feels like a larger restriction in flexibility. While having a neatly decoupled structure, we cannot program in as many ways we could if we didn't use the framework.
4. With Angular2 it is easy to create a web-application as well as a mobile application. This can be useful since we are independent of the platform and thus can the web-application can be used on any devices (desktop computer, android phones, apple devices, windows phones).
5. Angular2 features some powerful and neat concepts like for example dependency injection and data binding, helping us to maintain a large codebase, without the need of gluing the parts together manually. It has been tested on the most popular browsers as well as their earlier versions.

4.4. Final Design Choices

Now we would like to discuss the final design choices we made in our project. We will argue what technologies we decided to use and we will explain the reasons for these decisions:

- *TypeScript rather than JavaScript:*
Rather than using JavaScript, we decided to use TypeScript. We decided in favor for static safety, which can be very useful in larger projects. Also we chose in favor for classes and interfaces, which helps to create a solution much more clear and understandable for maintenance.
- *Angular2:*
Also we chose to use the framework Angular2. The decision for Angular2 rather than AngularJS was clear, since we already decided to use TypeScript, which is the language used in Angular2. Also this decision helped us to create a clear, modular setup with strictly decoupled components in the paradigm of MVC.
- *ANTLR4:*
ANTLR is a tool for creating parsers, lexers and tokenizers. We decided to use this tool to create our interpreters. ANTLR creates JavaScript classes, which can be used in a TypeScript document if either for every class a declaration file is added to the document or if we declare every method in those classes to be arbitrarily (e.g. taking as many arguments of any type and returning a result of any type). We decided to choose the second option due to the fact the generated code is supposed to be type-correct and thus

4. Design

we do not need the static safety which is provided by TypeScript for this part of our solution.

- *No back-end:*

In order to allow our web-application to be used in an offline-setting we decided to create a heavy client, rather than a heavy server. Currently web-development turns out to make a transition from previously heavy servers to currently heavy clients too.

5

Implementation

In this chapter we are going to present implementation-specific aspects in our project. We will start by explaining the general structure, namely how we put the MVC and Composite design paradigm into practice and how these individual parts communicate. We will demonstrate the individual components, and list the most challenging problems we encountered in the respective components. Later on we will explain the general work flow in our application after a enter keystroke occurs. In the second part of this chapter we will focus on the above mentioned problems and we will explain how we solved those problems. The problems are structured by graphical user interface elements, namely history, canvas, editor and input text area.

5.1. General Structure

In Angular 2.0 projects are usually composed of several components. One component describes the layout and functionality of one particular encapsulated part of the website. Everything in Angular 2.0 can be seen as a component, including Directives and Services. Directives are components without a layout, whereas Services are data collections, which can be used to feed and retrieve information from components. In Figure 5.1 all the components we used in our application are depicted with their according graphical representation.

5. Implementation

All eight components and their relation to the graphical user interface:

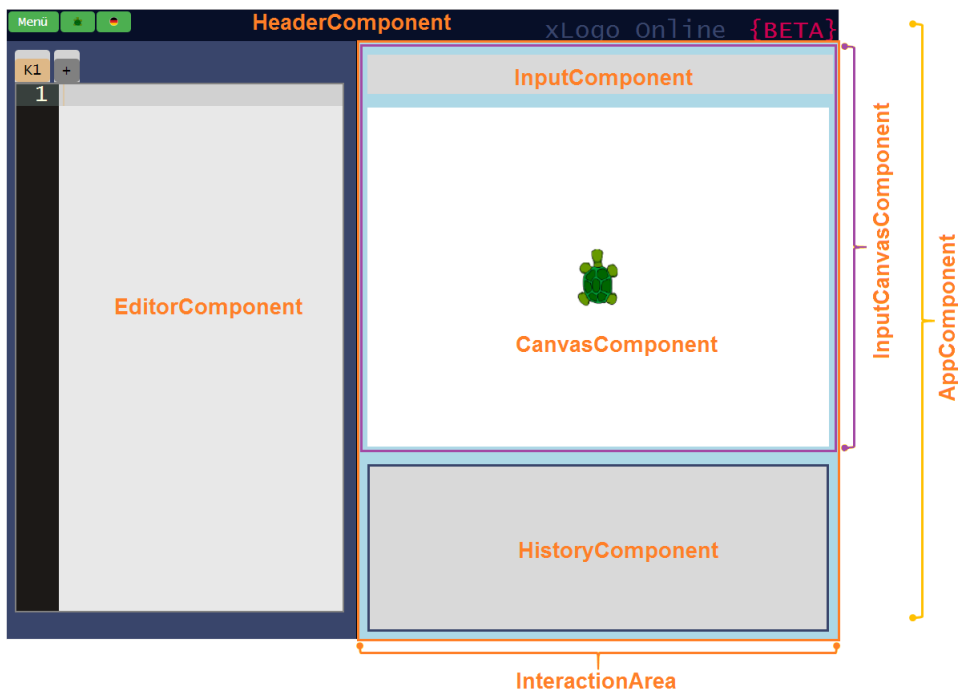


Figure 5.1.: Components in our web-application

As you can see in the figure above, we decided to divide our application into eight components, which build a tree. The corresponding tree to our component setup can be seen in Figure 5.2. The tree contains inner nodes and leaves. Inner nodes (App, InteractionArea and InputCanvas, depicted in blue) are used as wrappers to provide the application with more structure, whereas leaves (Header, History, Input, Canvas and Editor, depicted in purple) are visible components with a user interface representation.

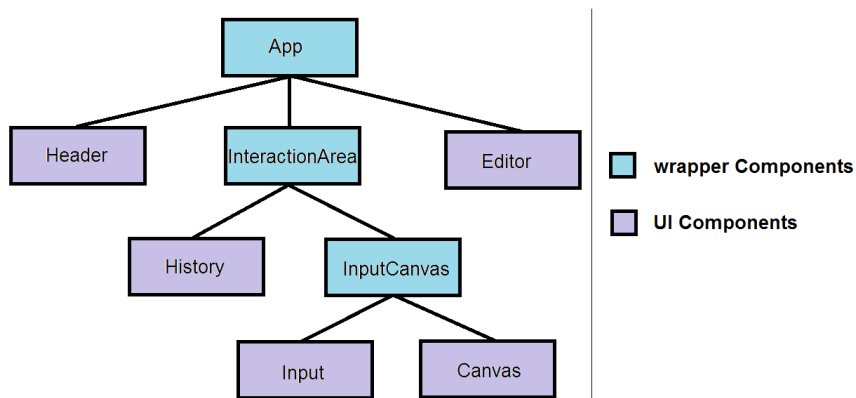


Figure 5.2.: Component-tree

Before explaining how information from a user input is forwarded through the system until it is finally shown on the canvas or reported as an error, we will present and explain the five UI components we used in our application, what they are used for and what interesting problems we had in the respective components. We leave wrapper components away, since their main purpose is to provide more structure in general.

In the following list we present our five UI components:

- *Header component:*
In order to display the header of our website we use this component. It comprises a DIV element, which in turn holds an <h2> element, used to represent the headline of the website as well as a few setting and menu buttons. In this section the most interesting problem was how to react to a user request to load or store the editor's content from a local file, independent of the platform we are working on.
- *Editor component:*
In this component we describe the editor's behavior and view. We describe a few of the special characteristics of this component in the controller. The editor features many challenging problems. The most interesting and important one is how to implement error highlighting in the text editor.
- *History component:*
The History component is used to display previous commands, which were committed from the Input component. We use a service to store previous commands. This service is fed by the InputCanvas component and it is used to retrieve data from the History component.
- *Input component:*
The Input component is used as an interface for user input, which is then executed. The effect of the user input (XLogo command) will be displayed on the canvas and the command will be registered in the service called commandsservice, from where the previous commands are written to the History. If there is an error in the user input, the error is reported to the user via History as well. In this sense the input is the starting point of every user interaction. We are going to explain this process in more detail in the next section (section 5.2).
- *Canvas component:*
In the Canvas component we render the results of any command which has a graphical impact. The Canvas component holds two canvas elements. One is used to display the turtle. The other is used for drawing. One of the most interesting problems in this component was how to implement a canvas which allows panning in all directions in a continuous manner.

The tree of components can be traversed from the root node (App) to the leaves. All components in the tree are rendered are provided with data from top to bottom. This means, whenever Angular's change detection algorithm reports some change in the model, all components are informed. The information is automatically propagated from the root to the leaves and can be observed by the lifecycle event handlers.

5.2. Information Flow

In this section we will explain what happens once the user hits ENTER on his keyboard in detail. We will focus on the interaction and information flow between the following user interface elements: Input, Editor, Canvas, and the History.

The whole process starts when the user presses ENTER on his keyboard. There are a few plausible scenarios, depending on the state of the application. Either the user has a valid (i.e. compiling) set of methods in his editor, or not. An invalid content of the editor can for instance happen if a method declaration is still missing its END or TO keyword or if it contains too many or too few arguments for one of the basic methods. If this is not the case, the application is allowed to continue with syntax checking for the user's input. But it might still be the case that those commands do not compile or throw an error during execution.

For this we distinguish three cases: Either both the editor's content and the user's input are valid XLogo programs, the editor is valid but the user's input does not compile or not even the editor's content is a valid XLogo program and thus the user should not even be allowed to start the execution of his program. We discuss the three cases now separately.

5.2.1. Case 1: Both the editor and the user input are valid

We present the following two-stage process of information flow after the ENTER key was hit in case both the editor's content and the user's input are valid and thus will compile successfully:

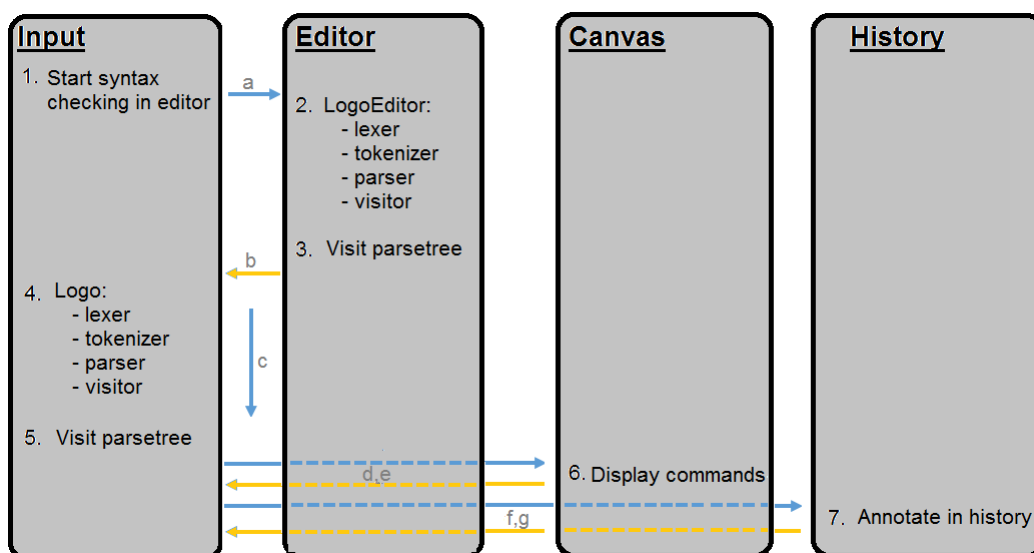


Figure 5.3.: Information flow which happens whenever the user hits the ENTER key. Boxes show UI components, arrows show the control flow

The Input component is the main component in this process and it is in charge of managing the whole information flow. First it initiates a syntax checking of the current content in the editor (arrow a). We need to check for syntax errors in the editor before executing the actual user

input because it might contain a method invocation, thus we need to guarantee that whatever method the user could invoke, the program behaves correctly. In order to ensure that we first check the editor for syntax errors. We perform a syntax check by building up a parse tree of the editor's current content (using a separate grammar, specially designed for the editor) and by traversing the according parse tree. We make sure the parse tree does not contain any wrong parent-child-relationships and by checking for a positive response by the syntax checker (arrow *b*) assure that the editor's only contains valid XLogo programs.

Once the syntax checking is over we perform another syntax checking, but this time with the user's input (arrow *c*). Hence we start the lexer and tokenizer, we build the parse tree and traverse it. Traversing the parse tree of the actual user program results in methods calls within the Canvas component (arrow *d*). The user input is transpiled to graphical operations on the canvas. Once the parse tree is fully traversed (arrow *e*), we pass the command itself to the History component, where it is stored and annotated for the user (arrow *f*). Executing the command finished successfully (arrow *g*).

5.2.2. Case 2: The editor is valid, but the user's input is not

Similar to the previous case we try to perform the above-mentioned two-step approach of first checking the editor for syntax errors and later trying to execute the user input.

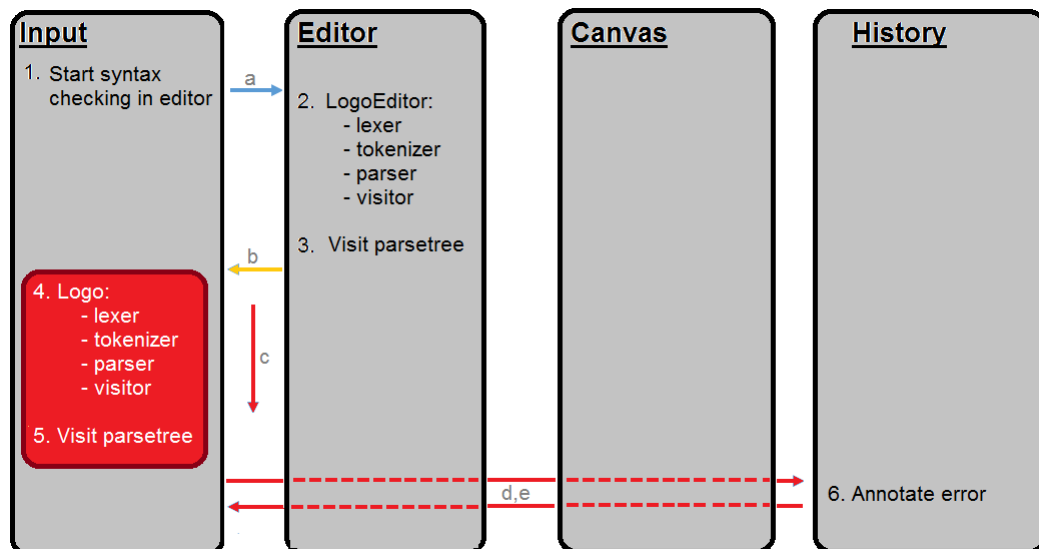


Figure 5.4.: Information flow with invalid user input but valid editor. Control flow interruption after syntax checking the user input or executing it.

In this case we assume the editor contains a valid XLogo program. Thus initiating a syntax check for the editor's content (arrow *a*) should complete successfully (arrow *b*). However, during the syntax check for the user's program (arrow *c*) we will encounter an error because of the invalid user input. Somewhere along the way of building lexer, tokenizer, parse tree and traversing it, we observe an error, analyze it and report it back to the user via an annotation in the history (arrows *d*). Then the execution is finished (arrow *e*).

5. Implementation

5.2.3. Case 3: The editor is invalid

In this case we assume that the editor does not compile properly. We cannot allow the execution of the user input to start if the editor's text is invalid because this implies that at least one method contains a syntactic error, which would lead to a runtime exception if the method would be invoked by the user. We try to prevent this by disallowing the execution of any kind of user input while the editor is still invalid. This is an over-approximation and it is possible that the user input is harmless, but we still refuse the execution of those programs due to the fact that it might lead to faulty behavior in some cases if we were to allow user input to be executed while the editor contains syntactic errors.

We propose the following approach:

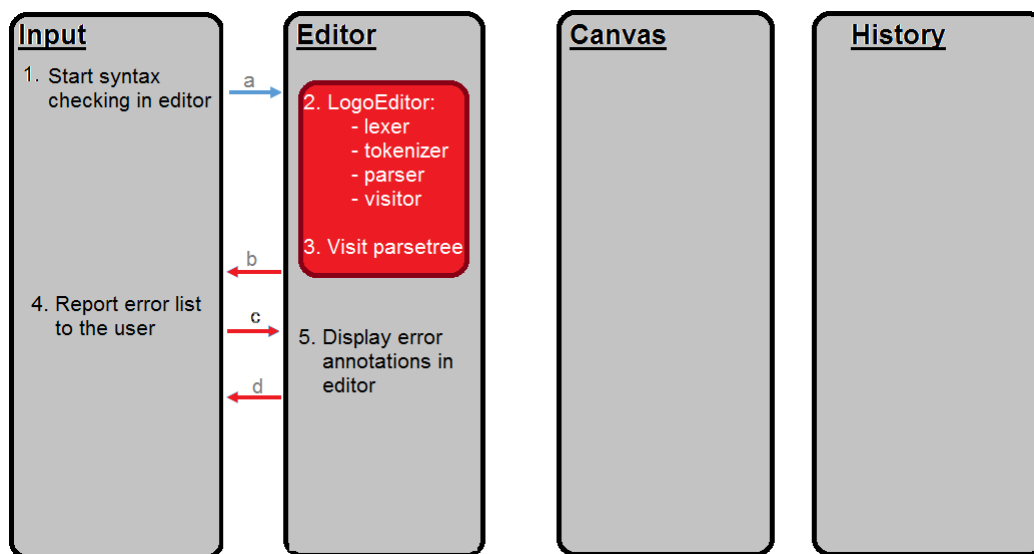


Figure 5.5.: Control flow interruption after performing syntax checking with invalid editor.

First, we initiate a check on the editor for syntactic errors (arrow *a*). If this check fails (arrow *b*), we know that the editor contains errors, which need to be passed on to the user. We analyze the error, create a more meaningful response from it and notify the user about the syntactic error (arrow *c*) directly in the editor on the corresponding line where the syntactic error appeared during the syntax check. Afterwards we conclude the current execution (arrow *d*).

5.3. Realization on Selected Technical Features

Next we will discuss the implementation details of our user interface elements. We will explain a few interesting problems related to those UI elements and how we solved them. Namely we will present our approach of loading and storing the editor's content if there are multiple tabs open. Next we will discuss how to allow the user to manually pan the canvas in all directions. Lastly we will explain how we did error highlighting in the editor and how to implement dynamic window resizing. We will close this section by a presentation of our most challenging

problems. At this point we present the problem of working with multiple module loaders (eg. more than one program, which resolves dependencies among the modules used by a module), the problem how to implement the visitor pattern in a programming language, which does not feature method overloading (like in JavaScript), how we handled the problem of JavaScript being single threaded in respect of an interrupted execution (using the XLogo command *wait* and intercepted execution).

5.3.1. Loading and Storing Editors

Xlogo online offers the possibility to create new editors. This can be useful in our use cases at primary schools to provide the pupils with a possibility to give their collection of programs more structure. In our application we provide the option for any number of tabs, which each may contain any number of methods. Internally we store each editor content as an entry in a large hashmap, which uses the tab's unique ID as a key and the content text as value. Whenever the current tab is changed we need to save its content in the hashmap and load the text of the according tab we are switching to.

In order to save the editor we use a library called `FileSaver.js`, which facilitates the task to write to a local file. In the following code you see how we save the content of all tabs to a local file:

```

1 saveFile() {
2     //load current text
3     this.editorText[this.currentHighlightTab] = ←
         ↪ AppComponent.editor.getValue();
4
5     //accumulate texts from all editors, separate with $
6     var filesaver = require('lib/FileSaver.js');
7     var text = "";
8     for (var key in this.editorText) {
9         let value = this.editorText[key];
10        text = text + value + " $";
11    }
12    var blob = new Blob([text], { type: ←
         ↪ "text/plain;charset=utf-8" });
13
14    //create name for file (use current date)
15    var dateObj = new Date();
16    var month = dateObj.getUTCMonth() + 1;
17    var day = dateObj.getUTCDate();
18    var year = dateObj.getUTCFullYear();
19
20    //write file
21    var currDate = day + "/" + month + "/" + year;
22    filesaver.saveAs(blob, "Logo " + currDate + ".lgo");
23 }

```

5. Implementation

In line 8 to 11 we accumulate all texts from all editor tabs in one string. Individual tabs are separated using the String \$ so we can reconstruct the tabs and their content when loading a file. We decided to use the character \$ as a separator because it is not contained in the language itself and it is a special character, which means that it is not allowed as a method or parameter name, i.e. it is unique and thus fulfills the requirements as a suitable separator. In line 12 we create a new blob which is based on the utf-8 encoding and which is then used on line 22 to save the editor to. On lines 15 till 18 we determine the current date and then we use it on line 22 as file name for our new file with the file ending lgo.

When we decide to load a file from the local file system we do the following:

```
1 //... within initialisation phase:
2
3     var x = <HTMLInputElement>document.createElement("INPUT");
4
5     //load content of files on input file select
6     x.addEventListener("change", function (event) {
7         var file = x.files[0];
8         var reader = new FileReader();
9         reader.onload = readSuccess;
10        function readSuccess(evt) {
11            localRef.currentIndexTabs = 0;
12
13            //delete all editors and all texts
14            localRef.editorText = [];
15            var ul = document.getElementById("tabs");
16            while (ul.children.length > 1) {
17                ul.removeChild(ul.firstChild);
18            }
19
20            //read input, separate after $, distribute to tabs
21            var allText = evt.target.result;
22            allText = allText.split("$");
23            var j = 0;
24            localRef.addTab();
25            for (var j = 0; j < allText.length-2; j++) {
26                //add new tab with this content
27                AppComponent.editor.setValue(allText[j], 1);
28                localRef.addTab();
29            }
30            localRef.editorText[localRef.currentHighlightTab] = ↔
31                ↔ allText[j];
32            AppComponent.editor.setValue(localRef.editorText ↔
33                ↔ [localRef.currentHighlightTab], 1);
34        }
35        reader.readAsText(file);
36    }.bind(this), false);
```

First, in line 3, we create a new input `HTML`Element. This object is then manipulated to show the filemanager and to allow file endings with `lgo` only (code omitted here). We define an event listener which listens for events of type *change* only, in line 6. Whenever the file explorer is used to choose a file this event listener is triggered and a new `FileReader` is instantiated which reads the file (line 8). We create a new hashmap (line 14) and remove all tabs from the current editor (lines 16 to 18). After reading the input, splitting a the character `$` (lines 21 and 22), we add a new tab for every entry in the hashmap (line 28) and finally we set the content of the editor to the current tab ID (lines 30 and 31).

5.3.2. Error Highlighting

Error highlighting, or syntax checking, is an important mechanism to provide the user with more information about his code. Without syntax checking it is much harder to find typos, missing keywords and missing parameters for instance. We want to provide the user with a real-time feedback about his code so he is able to evolve his code gradually, while knowing that no syntax errors are present at every moment. Previous versions of XLogo IDEs did not feature this attribute. We argue that the user experience is amended when using syntax checking due to the fact that the user knows about the state of his program at all times. He learns that he needs to fix syntax errors before executing a program. The knowledge about the state of a program is not available if no syntax checking is used. Thus it might not be clear whether the reason for the error lies within the editor or in the structure or form of the user input directly, if no syntax checking in the editor is available.

We approached the problem checking for syntax errors in the editor by using our two-stage approach described in section 5.2. Every keystroke in the editor schedules a callback, which is executed after exactly one second of inactivity. We decided for a timeout of one second because it is not really necessary to check for syntax errors after every keystroke. This would result in many syntactically incorrect programs and thus too many notifications for the user. We find an approximation by waiting for one second of inactivity. Hereby we reduce the number of unneeded notifications (which are not interesting to the user since he is still working on a method) tremendously compared to the approach where a syntax check is performed after every keystroke. Newer keystrokes can cause an abort of the most recent callback and replaces it with a new callback for the updated text in the next second. The callback itself analyzes first the text in the editor in order to find all methods. Then it traverses each method using a visitor (which was created on a grammar which defines a correct editor content). The visitor searches for syntax errors by traversing the parse tree. Once a syntax error is found we throw an exception in the parser, which is caught by the caller. With the error message we get the exact position of the syntax error (line and column), which allows us to add an annotation to the editors line numbers if an error appeared during syntax checking. We provide error messages for the following kind of errors:

- Too few parameters:
no viable alternative at input X
- Too many parameters:
extraneous input X

5. Implementation

- Missing keyword *to*:
extraneous input X expecting 'to'
- Missing keyword *end*:
mismatched input X expecting 'end'
- No such method:
Missing STRING

The caller checks which kind of syntax error it is and looks up the error message (in the language specified by the system), continues displaying it along with the error annotation directly on the editor, in the according line. In the following code we demonstrate how we detect which kind of syntax error we found:

```
1  try {
2      //build parsetree for content and visit it
3
4      } catch (e) {
5          var extraneous;
6
7          //analyze input and change text accordingly
8          if (e.msg.indexOf("\'to\'") !== -1) {
9              //missing TO
10             e.msg = AppComponent.currentLang["missingTo"];
11         } else if (e.msg.indexOf("\'end\'") !== -1) {
12             //missing END
13             e.msg = AppComponent.currentLang["missingEnd"];
14         } else if (e.msg.startsWith("extraneous")) {
15             //too many parameters
16             extraneous = content.slice(e.start, content.length);
17             extraneous = extraneous.split(/\s,+/)[0];
18             e.msg = AppComponent.currentLang["extraneousInput"] + ↵
19                 ↵ extraneous;
20         } else if (e.msg.startsWith("no viable")) {
21             //too few paramteres
22             extraneous = content.slice(0, e.start + ↵
23                 ↵ 1).split(/\s,+/);
24             extraneous = extraneous[extraneous.length - 2];
25             e.msg = ↵
26                 ↵ AppComponent.currentLang["noViableAlternative"] ↵
27                 ↵ + extraneous;
28         } else if (e.msg.startsWith("missing STRING")){
29             e.msg = AppComponent.currentLang["noMethodName"];
30         }
31
32         this.errorIndizes.push({ row: e.line + offset, text: ↵
33             ↵ e.msg, type: "error" });
34     }
35 }
```

The general approach is, to build a parse tree for the editor's content and to visit it (code omitted here). If the visitor finishes successfully, the editor is valid. But if it throws an exception, we know that some kind of syntax error is present in the editor. We catch the exception and determine whether it is a missing TO (line 8-10), missing END (line 11-13), too many parameters (line 14-18), too few parameters (line 19-23) or an unknown method name which is used in the program (line 25). In all cases we exchange the error message by something more meaningful for the user and append it to a list of errors (line 28) in the systems language.

One drawback we have with this approach is, that we are only able to find the first occurrence of a syntax error in a given method. If we wanted to find and annotate all syntax errors in a method, we needed to remove the critical part from the method and reparse it searching for the next syntax error. This would be highly inefficient for long programs with many syntax errors. With our approach we only find the first syntax error per method and restart the parser once the input changes and is not altered for one second. Once the first syntax error is resolved, we will display the next and so on.

5.3.3. Canvas Panning

Sometimes it would be useful to have a mechanism to pan the canvas' content within its viewport. For example if we already know that some picture will take more space than usually (if we write text for example or draw pixel images), then it would be of interest for the pupils to have a mechanism of panning the canvas within its own viewport. We implemented a way of panning in our application and we will now explain its core mechanism. We decided to tackle the problem by creating an offscreen canvas which is nine times as large as the viewport visible on the screen (see Figure 5.6).

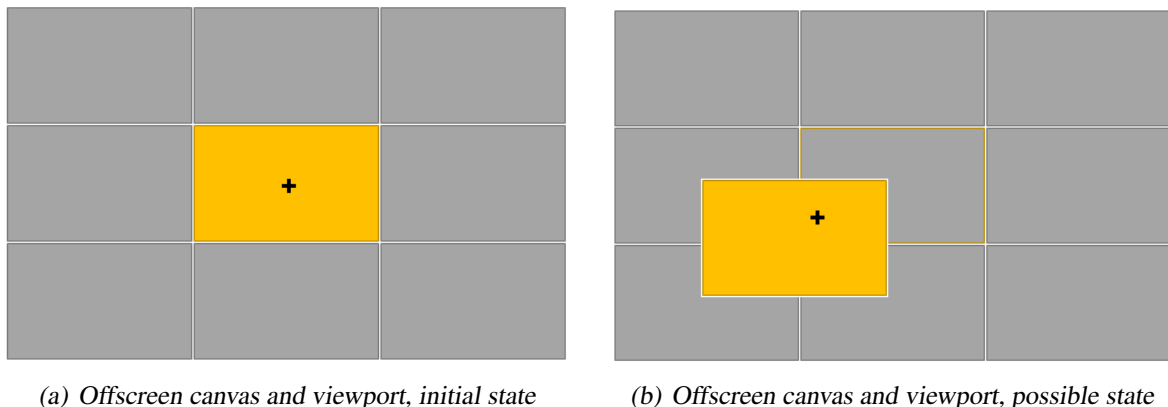


Figure 5.6.: Offscreen canvas with viewport, allows panning within viewport

Whenever the user draws something on one canvas it is automatically backed up on the offscreen canvas too. If the user starts to pan the viewport, we measure the relative distance to the original position and copy the respective part (of the same size as the viewport) from our offscreen canvas to the viewport. We disallow the turtle to leave the viewport to ensure that we do not need parts which lie beyond the range of our offscreen canvas.

5. Implementation

A few questions worth to discuss about this approach:

1. **What happens if the turtle leaves the viewport programmatically?**

We disallow the user to pan the turtle out of the viewport, because otherwise it is easy to get lost. But what should happen if the user types a command which programmatically transfers the turtle to a position which lies beyond both the viewport and the offscreen canvas? One possibility would be to disallow those methods. For this to work we would need to anticipate the effect of each method call and all basic operations. Another possibility would be to always show the last part on the screen, such that the turtle stays in the middle of the viewport, while the world moves around it. This approach contradicts with the methodology of the teaching material, where it is clearly the case that we move the turtle, not the world surrounding it. One last option is to keep everything as it is and to let the user realize that he has to correct the mistake programmatically as well if we intends to get the turtle back into his viewport. This solution might lack some intuition.

2. **What happens if the viewport's size changes?**

Whenever the user pans his viewport, the according part of the offscreen canvas is copied to the viewport. But our application allows the user to resize his viewport. Since we do not want to see any stretched images in our viewport, we need to check the size of the viewport before copying anything from the offscreen canvas to the viewport and make sure the viewport and the copied area have the same size. Also we need to assure that the initial size of the offscreen canvas matches with the maximum possible size of viewport with panning. Our solution is to always copy an area of the exact same size as the viewport and to initiate the offscreen canvas to be nine times as large as the largest possible viewport.

3. **Which part of the canvas contains interesting information, file download?** We offer the option to download the current content of the canvas on demand. What is supposed to happen if the picture does not fit on the viewport but it uses a bit more that what is available of the viewport. We could try to save the part from the offscreen canvas, which contains all the information. But for that we need a notion of where the edges of the interesting parts in the canvas are. We left it open to implement as a future work. At the moment we allow the user to download the current viewport only.

4. **How to implement an infinite canvas?** If we were to implement an infinite canvas, we would need to implement a manner of self-localization. If the turtle is lost, we need to support a way to find it again. In a technical point of view we could store everything we explored so far in some kind of buffer and then simply search through the buffer to find the according entry.

Our web-application supports panning the canvas in a rudimentary way. We presented a few option for improvement in the above list. The current implementation could be improved to an infinite canvas, whose idea is described in Section 7.2.3.

5.3.4. Window Resizing

We implemented our web-application as a single-page app. On this page everything is visible at the same time (see Figure 4.3 for instance). To ensure that the user experience does not suffer

from ill-designed ratios between editor and right-hand side, we added a slider in between; the respective window sized can be adjusted. The mechanics behind this are implemented by hand. We added three event handlers for click, release and move gestures, whose implementation we are going to present now:

```

1  resizeStart() {
2      this.resizeOn = true;
3  }

```

This piece of code is the event handler for the `mousedown` event on the divider between left- and right-hand side. The only thing which happens here is that we set a flag, which will be used in `resizeStop` and `resize` to determine, whether we are currently in a resize mode.

```

1  resizeStop(e: MouseEvent) {
2      if (this.resizeOn) {
3          this.resizeOn = false;
4          this.width = e.clientX - this.padding + "px";
5      }
6  }
7  }

```

This method is the event handler for the `mouseup` event on the divider between left- and right-hand side. If we are currently actually resizing the window and now the mouse button was released, we calculate the ratios for left and right-hand side and update a variable width, which is a bounded property with data binding. Once the value is adjusted, the ratio is changed by Angular automatically.

```

1  resize(e: MouseEvent) {
2      if (this.resizeOn) {
3          var value: number = e.clientX - this.padding;
4          this.width = value + "px";
5          if (value < this.padding * 10) {
6              this.editorused = false;
7          } else {
8              this.editorused = true;
9          }
10
11         if (value > window.innerWidth - window.innerWidth / 5) {
12             this.inputused = false;
13         } else {
14             this.inputused = true;
15         }
16
17         AppComponent.editor.resize();
18     }
19 }

```

5. Implementation

This method is the event handler for the mouseMove event on the divider. If we are actually resizing the windows, we calculate the ratio which the two parts would have and then decide whether we want to set the hidden flag for either of the two sides. We do so if the ratio of the editor is smaller than 10% or if the right-hand side uses less than 5% of the available space.

5.3.5. Multiple Languages in XLogo Online

For a platform, which is specially designed to be widely accessible, it is important to provide support for various languages. In our web application we decided to implement this feature using a Service, which manages hashtables containing all strings visible in the GUI. For every language a hashtable is available (at the moment German and English), all using the same keys. A Service takes care of managing the hashtables. When requesting a string, we use the Service as a proxy, which hides implementation details. In Figure 5.7 we illustrate the information flow for string accesses:

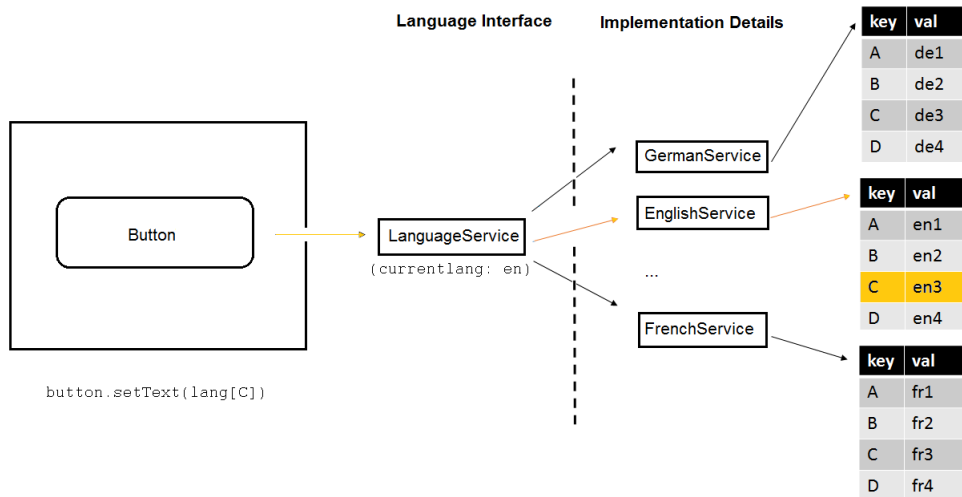


Figure 5.7.: Upon requesting a string `lang[C]`, the `LanguageService` checks the current language settings and accesses the respective string from the responsive `LanguageServiceHelper`.

The Service has to suggest a language upfront. He does so by inspecting the system language and loading the respective language (or English per default if the system language is not among the available languages). Similarly, the user can change the UI-language manually, which results in an internal exchange of hashtables as well. Our system is extensible and designed to be seamless for the end-user.

5.3.6. Colors

The dialect XLogo offers multiple representations for colors. One of the possible representation involves the color-name as string, another representation is a mapping to numbers between 0 and 16 and the third is the notation as RGB values. All of the following commands have the exact same effect:

```
setpc red
setpc 1
setpc [0 0 255]
```

Internally we simplify this ambiguity by converting every color to its respective RGB value. Color-names and numbers provide a way for the user to express colors in a shorter notation. But Color-names and numbers are only a small subset of all possible colors we provide, thus in our system we work with RGB values internally exclusively. For this reason we have a Service, called ColorService, which transforms every color (in whatever representation it is) into the equivalent RGB value and stores it.

5.3.7. Asynchronous Tasks in Single Threaded Environments

JavaScript is inherently single threaded. This means creating new threads and outsource time-consuming actions to different worker-threads is not as easy as it is in other programming languages. We encountered a related problem, when implementing the *wait* keyword, used in the XLogo dialect. This keyword leads to an interruption of the current execution, which is continued after a predefined amount of time. Since we need to execute everything on the main thread, it is difficult to react to the command using a synchronous waiting, because it would result in an unresponsive design, which is undesirable. The wait keyword itself is supposed to create the effect of synchronous waiting. We simulated this effect with an asynchronous wait (using the JavaScript *setTimeout* command). Whenever a wait command is visited by the visitor, a timeout is set. After this timeout a method is invoked to resume the execution of the remaining commands.

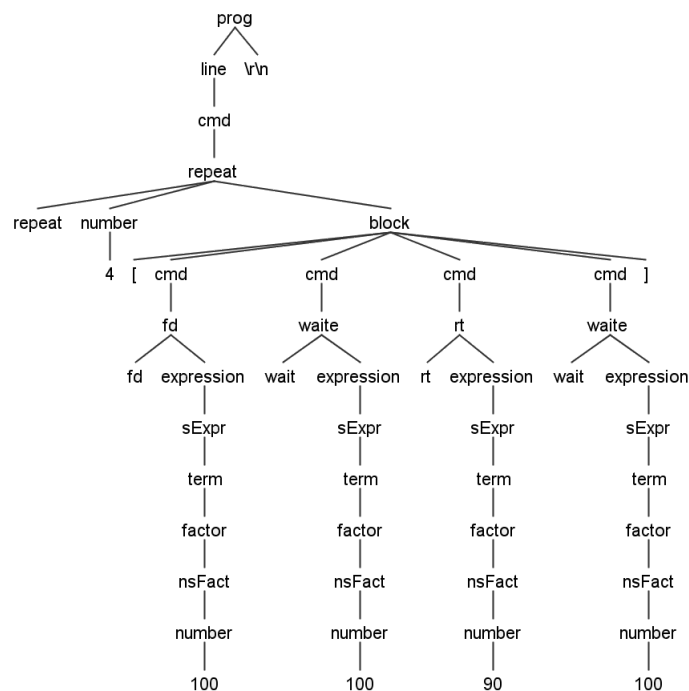


Figure 5.8.: Parse tree for program square with wait: repeat 4[fd 100 wait 100 rt 90 wait 100]

5. Implementation

In Figure 5.8 you see the parse tree for a Xlogo program `repeat 4 [fd 100 wait 100 rt 90 wait 100]`. We expect this program to draw a line, wait for a second, turn to the right, and wait another second until the loop body reexecutes. If we simply check in the visitor whether the current child is a wait instruction and then set a timeout for the requested amount of time until we resume the execution of the other children, we do not end up with what we intended. Set-Timeout initiates an asynchronous wait, which means that its content is pushed to the queue, where it stays until the timeout is over and the main-thread is idle for the first time. But in the meantime the visitor still continues with its execution. Thus, if we set the timeout for, let's say, 4 seconds, we end up with a scenario, where the whole square is drawn first and afterwards we wait for a long time. We fix the problem using the following approach: JavaScript works asynchronously. Thus we disconnect any siblings of any ancestors to ensure that they are not executed while the program is supposed to be waiting. After the timeout elapses, we reattach and visit them, have look at Figure 5.9.

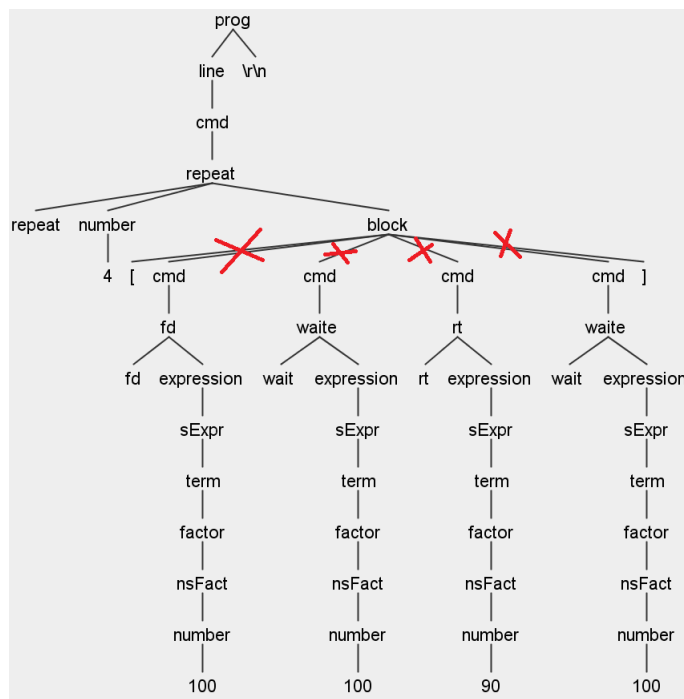


Figure 5.9.: Program square with wait, correctly implemented, after `setTimeout` is invoked, we remove all siblings of any of our ancestors. We make sure every method invocation is only made once at exactly the time when it is intended, while staying synchronous and responsive at all times.]

A second problem we would like to discuss is the problem of how to disrupt the execution once it is started. One of the main issues we encountered were, that it is not possible to interrupt the execution of the visitor once it is started, because this would require a second thread which could interrupt the visitor thread. But how should we proceed if the user implemented an infinite loop or any other structure which does not terminate per se? We do not want to wait until the browser announces, that the website is not responsive anymore and whether the user wants to close the tab. Instead we decided to count the number of visited nodes in the parse tree and issue a notification for the user that his program takes a very long time to terminate and

whether he wants to cancel. This notification is produced directly from the visitor himself. We set an upper bound of the number of visited nodes to one million. If the user decides to continue for a little bit longer, we double the threshold and continue with our work. One million is an experimental number. We found out that this number allows all exercises from the book to be completed without a message. If a message appears, we do not really want to use this help. In the following code we present our approach:

```

1 function visitChild(childnumber, ctx, ref) {
2
3     timeCounter++;
4
5     var operationConstructorString = ↵
        ↵ ctx.children[childnumber].constructor.toString().substr( ↵
        ↵ 'function '.length);
6 operationConstructorString = ↵
        ↵ operationConstructorString.substr(0, ↵
        ↵ operationConstructorString.indexOf('(') - 7);
7 var operationName = "visit" + operationConstructorString;
8
9 //do we exceed the threshold?
10 if(timeCounter > threshold){
11     //ask user whether he wants to continue
12     if (confirm(lang.getLang()["timeout"])) {
13         //increase current threshold to be twice the current ↵
        ↵ number
14         threshold = threshold * 2;
15     } else {
16         stop = true;
17         return;
18     }
19 }
20
21 //invoke the next child
22 return ref[operationName](ctx.children[childnumber]);
23 }

```

First we increase the hop counter in line 3. Then we extract the method name from the next child (lines 5 to 7), then we check whether the threshold of hop counters is already exceeded and lastly we call the next child's visit method.

5.3.8. Integration of Multiple Module Loaders

The ANTLR4 solution for the parser of course involves a multitude of files, which build a large tree of dependencies. In web-development it is important to make sure everything is available (e.g. loaded) before it is used. For this reason module loaders exist. A module loader usually is given the path to a file and it loads the file by searching for it at the respective path and giving it a name. In JavaScript everything is an object. Loaded modules are too. It is possible to call a method from an external module, by simply calling the method-name on the module's object name. It happens quite frequently that modules use other modules in their own code and thus need to load other modules first. The module loaders take care of those dependencies whenever they occur. Without module loaders, the programmer would need to know the exact order in which modules need to be loaded, which is a very cumbersome task for larger applications with a large number of modules, which are nested. This approach does not scale well.

Over the time several module loaders with the same functionality but different syntax have been invented. Due to the fact, that the field is very diverse and a great number of different versions, frameworks, tools and plugins exist, several module loaders became well known in the field of web technology. One of them is called *requireJS*. It is used mostly by the NodeJS community. RequireJS makes use of the *Asynchronous Module Definition API* (AMD) as opposed to the older community of CommonJS participants. CommonJS was a format, which was designed without keeping the limitations of the JS web environment in mind. Also in this format only one module tag was allowed per file in order to break circular and nested dependencies. The AMD format is designed to provide better debugging characteristics than CommonJS and it does not need any server-specific tooling, as it is required by the CommonJS format and it works much better across platforms.

ANTLR4 was implemented with RequireJS as module loader, whereas the Angular2 framework uses SystemJS as module loader. SystemJS works both with AMD and CommonJS and it is implemented for NodeJS too. In order to include the parser, we needed to include RequireJS into our Angular2 framework. We declared the keyword `require`, which is used by the RequireJS module loader on the most generic way, as a method with an arbitrary number of arguments of any type and a return type of any type:

```
declare var require: (...any) => any;
```

This declaration can be found in various classes in our project. Whenever we work in a component (which uses the Angular2 framework and thus the SystemJS module loader while still using another library or framework which uses RequireJS).

5.3.9. Visitor Pattern in JavaScript

JavaScript is by nature a language which does not support method overloading. This is an issue when implementing the visitor pattern. Instead of just invoking a *visit* method with a specific argument type (which covers all possible dynamic types of any argument which might come along), we need to invoke a *visitX* method, where X corresponds to the name of the node. Having method overloading is nothing but syntactic sugar and we can circumvent the problem by introducing unique method names for every Argument type. But for this to work it is crucial that every node in the parse tree knows the exact name of the children so it can invoke their visit method. We inspected the current node, extracted the names of its children and built a function call based on the string containing the name. Have a look at the following implementation of this functionality (*visitChild* method from the *EditorVisitor* class):

```

1 function visitChild(childnumber, ctx, ref) {
2   var operationConstructorString = ↵
      ↵ ctx.children[childnumber].constructor.toString().substr( ↵
      ↵ 'function '.length);
3   operationConstructorString = ↵
      ↵ operationConstructorString.substr(0, ↵
      ↵ operationConstructorString.indexOf('(') - 7);
4   var operationName = "visit" + operationConstructorString;
5   return ref[operationName](ctx.children[childnumber]);
6 }

```

We are given the child number, which indicates the index of the child we want to visit next. We also know the context *ctx* in which the method is called. This context belongs to the current node and its properties. As a last argument we are given a reference to a *this* object, which is in case of the example code an element of type *LogoEditorVisitor*. In line two we extract the full name of the child's constructor method after the *function* keyword. In line 3 we truncate the string even further to everything after *function* until the first occurrence of left parenthesis. In line 4 we prepend the keyword *visit* and finally in line 5 we run a method-call to a method with the name we assembled before.

5.4. How the Project is Instantiated

The framework provided by Angular2 is designed for single-page websites, meaning that there is supposed to be only one HTML file, which is loaded in the beginning. As usual this HTML file contains a header, which is used to import Angular2 libraries which are used to start calling the root component (have a look at lines 9 to 12 in the following code).

5. Implementation

```
1 <html>
2 <head>
3   <title>XLogo Online</title>
4   <meta name="viewport" content="width=device-width, ↵
5     ↵ initial-scale=1">
6   <link rel="stylesheet" href="app/styles/main.css">
7
8   <!-- 1. Load libraries... -->
9   <script src="../../../angular2.dev.js"></script>
10
11  <!-- 2. Configure SystemJS -->
12  <script>
13    System.config({
14      packages: {
15        app: {
16          format: 'register',
17          defaultExtension: 'js'
18        }
19      }
20    });
21    System.import('app/main').then(null, ↵
22      ↵ console.error.bind(console));
23  </script>
24 </head>
25
26 <!-- 3. Display the application -->
27 <body>
28   <my-app></my-app>
29 </body>
30 </html>
```

Next (lines 14 to 23) we find some instructions, which are used to transpile TypeScript to JavaScript. The HTML file contains a body too (lines 29 to 31), in which both usual HTML tags and custom tags, which originate from our own components. On line 24 we see, that Angular2 is being told to execute `main.ts` when the configuration of the SystemJS module loader finishes. In `main.ts` we find the following code:

```
1 import {bootstrap} from 'angular2/platform/browser';
2 import {AppComponent} from './app.component';
3 import {enableProdMode} from 'angular2/core';
4 enableProdMode();
5 bootstrap(AppComponent);
```

The only thing which happens here is, that we enable the production mode, which means that less errors are reported and we are in silent mode, not bothering the user any longer. Secondly we bootstrap our root-component, namely `AppComponent` on a browser setting. After `AppComponent` is created, the whole component tree is built and behaves as we explained earlier.

6

Evaluation

In this chapter we will demonstrate the measurable characteristics we evaluated our programming environment on. We present four benchmarks, which were chosen as representative programs to cover the teaching material for the booklet. We will analyze them in terms of performance and calculate a measure of how many drawing instructions we have per second for an average program. Based on those results we will make a comparative statement between the two systems. We will conclude the chapter by presenting a user study we conducted in a Swiss primary school class as a usability study.

6.1. Benchmarks

We will use four benchmark programs, which were designed to test different aspects of the program. The four benchmarks have the following properties:

1. *Small Test*
The small test is supposed to check whether the basic functionality works properly and can be used as a reference point for larger benchmark programs.
2. *Loop Test*
The loop test examines the program behavior with loops.
3. *Large Test*
The large test is intended to measure how the program reacts to large and time-consuming programs which are supposed to run as fast as possible.
4. *Timing Test*
This test program can be used to determine how the program behaves in a setting with

6. Evaluation

external interruptions (in terms of wait instructions which originate from the user).

For all the experiments mentioned in this chapter we used a Hewlett Packard EliteBook 840 G2 with Chrome version 53.0.2785.116. The computer has average computational power for a computer which is one year old. We conducted for all benchmarks two experiments:

- **Experiment 1**

How long does it take to traverse the parse tree (including the time needed to draw to the canvas) in average, over 1000 executions?

- **Experiment 2**

How long does it take to build and traverse the parse tree? This time includes the time to build the lexer, tokenizer and parser, as well as the time required to draw) in average, over 1000 executions?

The following two code segments illustrate the functional principle of our experiments.

```
1 //for testing purposes
2 for (var i= 0; i < 1000; i++){
3     var start = window.performance.now();
4
5     visitor.visitProg(tree);
6
7     var end = window.performance.now();
8     var time = end - start;
9     console.log(time);
10 }
```

The above code is found in the Input component within method executeCommand. We iterate 1000 times over the same method invocation. In the example above it is the method invocation to start the visitor. Surrounding the tree traversing visitor we record the time. Using a simple subtraction we figure out how long the execution of the commands in between took. The second experiment is more or less the same, except that this time we record the time not only for the visitor but also for the setup of lexer, tokenizer and parser:

```
1 //for testing purposes
2 for (var i= 0; i <1000; i++){
3     var start = window.performance.now();
4     var chars = new this.antlr4.InputStream(content + ↵
5         ↵ "\r\n");
6     var lexer = new this.LogoLexer.LogoLexer(chars);
7     var tokens = new this.antlr4.CommonTokenStream(lexer);
8     var parser = new this.LogoParser.LogoParser(tokens);
9     parser.buildParseTrees = true;
10    var tree = parser.prog();
11    this.lang.setLang(AppComponent.currentLang);
    InputComponent.speed = ↵
        ↵ parseInt((<HTMLInputElement>document. ↵
        ↵ getElementById("speedRange").value) * 100;
```

```

12     var visitor = new this.LogoVisitor.LogoVisitor( ←
        ↪ this.canvasComponent, this, this.editor, ←
        ↪ InputComponent.speed, this.lang);
13     visitor.visitProg(tree);
14     var end = window.performance.now();
15     var time = end - start;
16     console.log(time);
17 }

```

In both cases we keep track of the runtimes for either the tree traversal and drawing exclusively or the full process of building the parse tree, traversing it and reacting to its nodes. We show the results in a box and whisker diagram. Outliers are shown as circles. The upper whisker shows the 95% percentile, while the lower whisker shows the 5% percentile.

6.1.1. Benchmark: Small Test

The first program we use to analyze the performance of our system for is a small program with only nine basic instructions, which draw a square and then clear the screen. (Remark on why we clear the screen: In JavaScript's canvas we use the method *stroke* to actually draw onto the canvas. We point out, that the method *stroke* will actually redraw every line in the current segment and thus it will take more time if we do not clear the screen in the meantime):

```

1 fd 100 rt 90
2 fd 100 rt 90
3 fd 100 rt 90
4 fd 100 rt 90
5 cs

```

This small test is translated to the following parse tree:

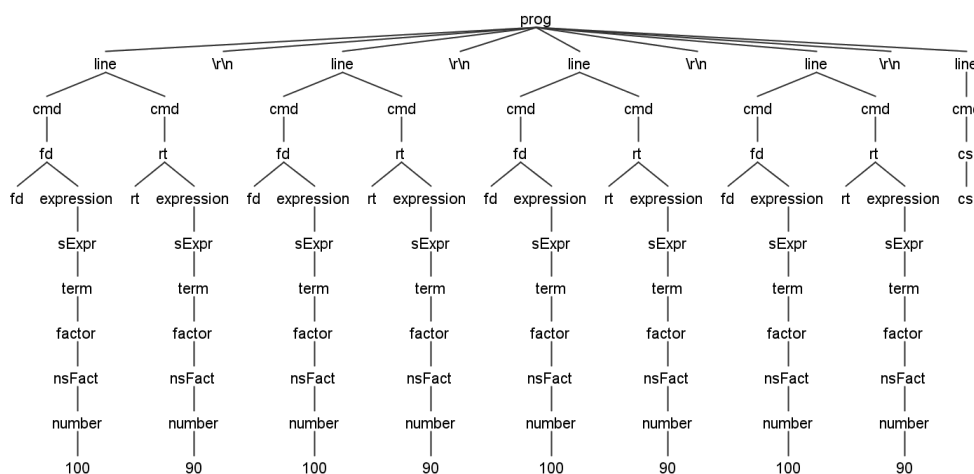


Figure 6.1.: Small benchmark program, which draws a square. Parsing it results in a tree with 92 nodes

6. Evaluation

Experiment 1: Execution time for drawing and parse tree traversal

In experiment 1 we want to make a statement about the performance of our system. We focus on the tree traversal and drawing part only. In our experiment we found the following results:

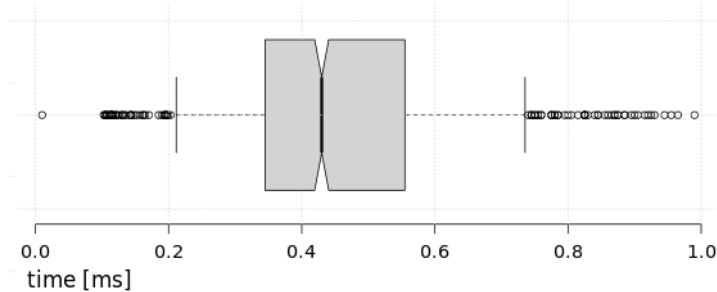


Figure 6.2.: Small benchmark results, Experiment 1

	performance [ms]
upper whisker	0.74
3rd quartile	0.56
median	0.43
1st quartile	0.35
lower whisker	0.21

The results show, that all 1000 iterations were finished within 0 to 1 milliseconds. The smallest data point traversed the tree and draw onto the canvas within 0.21ms, while the largest data point took 0.74ms to finish. We found the median at 0.43ms with a 1st quartile of 0.35ms and a 3rd quartile of 0.56ms. We have quite a few outliers on both sides (in Figure 6.12 you see a few data points to the left of the lower whisker around 0.2ms and a few others which lie beyond the upper whisker, which is around 0.7ms) but none of them lie very far from the others.

Experiment 2: Cumulative execution time

In experiment 2 we are interested in getting to know the performance for the whole process of building up the parse tree, traversing it and interpreting it. In comparison to the first experiment we had some additional tasks which were measured here too, namely build the lexer, tokenizer and parse tree, whose composition was not counted for in the first experiment. We found the following results:

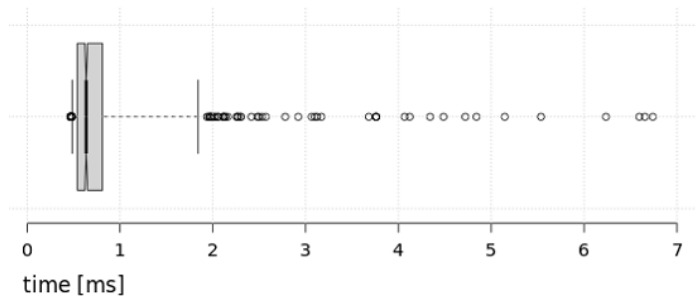


Figure 6.3.: Small benchmark results, Experiment 2

	performance [ms]
upper whisker	1.84
3rd quartile	0.81
median	0.63
1st quartile	0.54
lower whisker	0.48

The results show that most data points lie within 0.54ms and 0.81ms. In comparison the experiment 1 most data points are shifted to the right, which is completely reasonable, since we are doing more work in this experiment anyway. In this experiment we have more outliers which lie farther away from the median, though. Especially large data points are more prominent.

6.1.2. Benchmark: Loop Test

In the second benchmark we find a program which makes use of loops. Thus, in this experiment we are going to examine how the system behaves with loops, in respect of its performance. The benchmark is the following program:

```
1 repeat 30[fd 100 bk 100 rt 360/30] cs
```

The parse tree for this program looks like this:

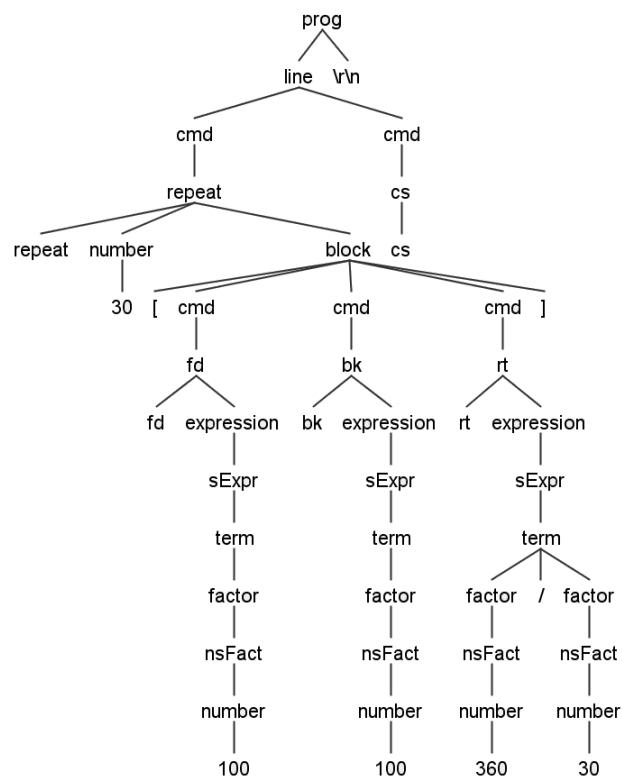


Figure 6.4.: Loop benchmark program, which draws a star. Parsing it results in a tree with 47 nodes

We will now discuss the two experiments for this benchmark. The parse tree itself contains less nodes than the parse tree in the previous benchmark, but this time we re-traverse certain parts of the tree, due to the control structure of a loop. Thus the visitor visits much more nodes after all.

6. Evaluation

Experiment 1: Execution time for drawing and parse tree traversal

In the first experiment we were again mostly interested in a performance measurement for traversing the parse tree and drawing onto the canvas. We use the code presented for experiment 1 and found the following results:

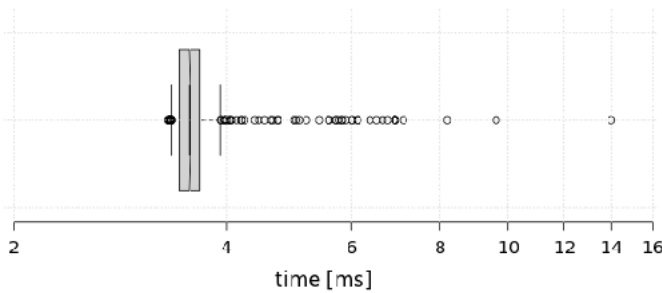


Figure 6.5.: Loop benchmark results, Experiment 1

	performance [ms]
upper whisker	3.91
3rd quartile	3.65
median	3.55
1st quartile	3.43
lower whisker	3.34

Due to the larger number of nodes which are visited in this benchmark program, we have in general a longer runtime. Most data point lie within the range of 3.43ms and 3.65ms - only a few lie beyond the the upper whisker and can take as long as up to 14ms for the shown program to execute.

Experiment 2: Cummulative execution time

In experiment 2 we run the program and measure the elapsed time between instantiating the parse tree (via building a lexer and a tokenizer) until the outcome is drawn onto the canvas. We found the following results in this experiment:

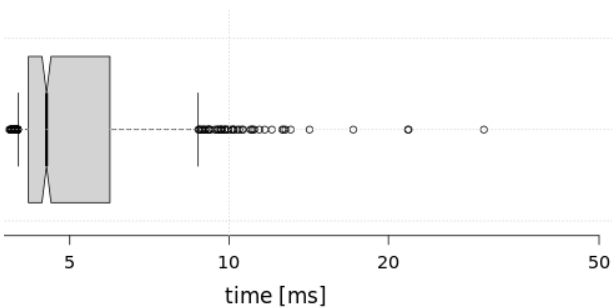


Figure 6.6.: Loop benchmark results, Experiment 2

	performance [ms]
upper whisker	8.74
3rd quartile	5.95
median	4.52
1st quartile	4.18
lower whisker	3.99

The thing which is most noticeable in those test results is the large variation. While the median takes around 4.52ms to finish building up the whole parse tree, visiting is and drawing accordingly onto the canvas, we observe, that 25% of all data points lie below 4.18ms (and the lower whisker is at 3.99ms, which is fairly close the the first quartile). At the same time we notice that only 75% of all data points lie below 5.95ms which is surprisingly far away from the upper whisker at 8.74ms. If we take a look at the outliers, we see immediately, that there is an enormous variation in the upper part of the data points.

6.1.3. Benchmark: Large Test

Our third benchmark is used to test our programming environment's behavior for programs, which take a long time to execute. We decided on a benchmark instance which draws skewed circles.

```
1 repeat 30[repeat 360[fd 1 rt 1] rt 360/30] cs
```

Have a look at the parse tree which belongs to this program:

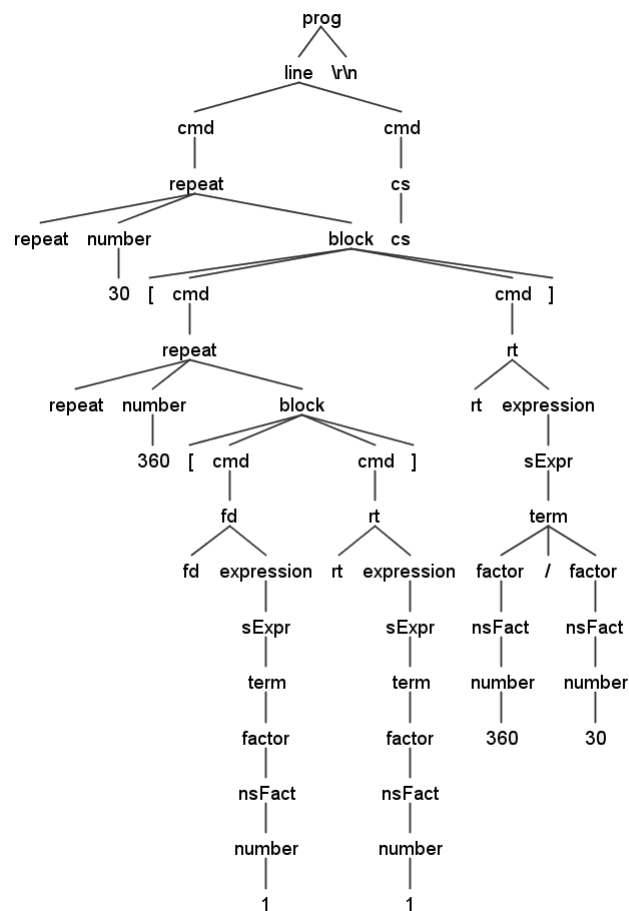


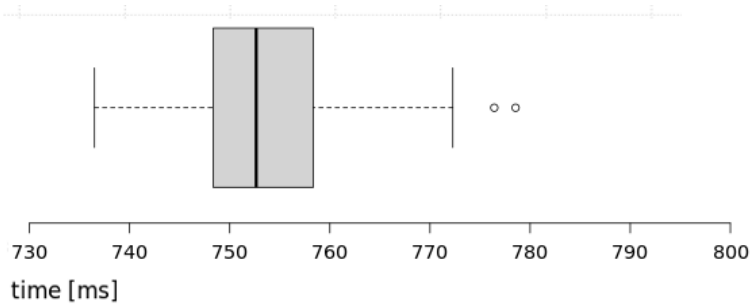
Figure 6.7.: Large benchmark program, which draws skewed circles. Parsing it results in a tree with 52 nodes

In terms of its size, this benchmark is clearly the largest program of all our benchmarks. We find nested loops in it, which result in a total of over 11000 nodes, which are visited in this program. Due to the timeout which is usually used in browsers to determine whether web pages are still responsive or not, we had to alternate the experiment to 10 rounds with 100 repetitions each. This helped to stay responsive during the whole experiment. For this alternated procedure we have to expect more outliers than in the other experiments. Usually the experiment starts with outliers, for instance because of empty caches.

6. Evaluation

Experiment 1: Execution time for drawing and parse tree traversal

In our first experiment we observed the time between starting the visitor on the already prepared parse tree and finishing to draw on the canvas. The following box plots show the results to our experiment:



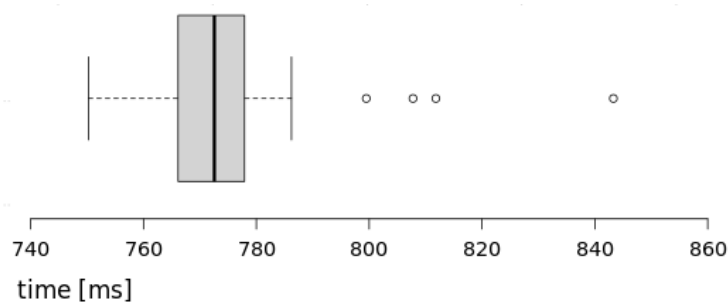
	performance [ms]
upper whisker	772.28
3rd quartile	758.35
median	752.65
1st quartile	748.34
lower whisker	736.50

Figure 6.8.: Large benchmark results, Experiment 1

We notice, that in this experiment we are only marginally slower than in the loop benchmark. This contradicts our intuition since this time the visitor has to visit hugely more nodes than in the other experiment. We explain this observation by the fact, that traversing the parse tree is done on the CPU and is thus quite fast in terms of processing time. The drawing would take longer if it were not for the fact that we do not render all changes individually, but all at once, in the end of the computation.

Experiment 2: Cumulative execution time

In the second experiment we are interested in the performance overhead for building the parse tree first, before visiting it and drawing to the canvas. The results are illustrated in the following diagram:



	performance [ms]
upper whisker	786.23
3rd quartile	777.91
median	772.55
1st quartile	766.12
lower whisker	750.24

Figure 6.9.: Large benchmark results, Experiment 2

In this experiment we observe drastically higher runtimes in general - around 750ms in most cases. This performance drop cannot be due to the traversal of the parse tree and the drawing, because those aspects were already covered by experiment 1. Thus it involves longer runtimes to build the lexer, tokenizer, and to build the parse tree itself.

6.1.4. Benchmark: Timing Test

So far all benchmarks were designed to run as fast as possible. Their command set did not involve any kind of wait signal, which could be used by the user to enforce an interruption of the execution. In this benchmark we test our system against this using the following benchmark program:

```
1 repeat 100[fd 100 rt 90 fd 1 rt 90 fd 100 rt 180 wait 1] cs
```

The according parse tree looks as follows:

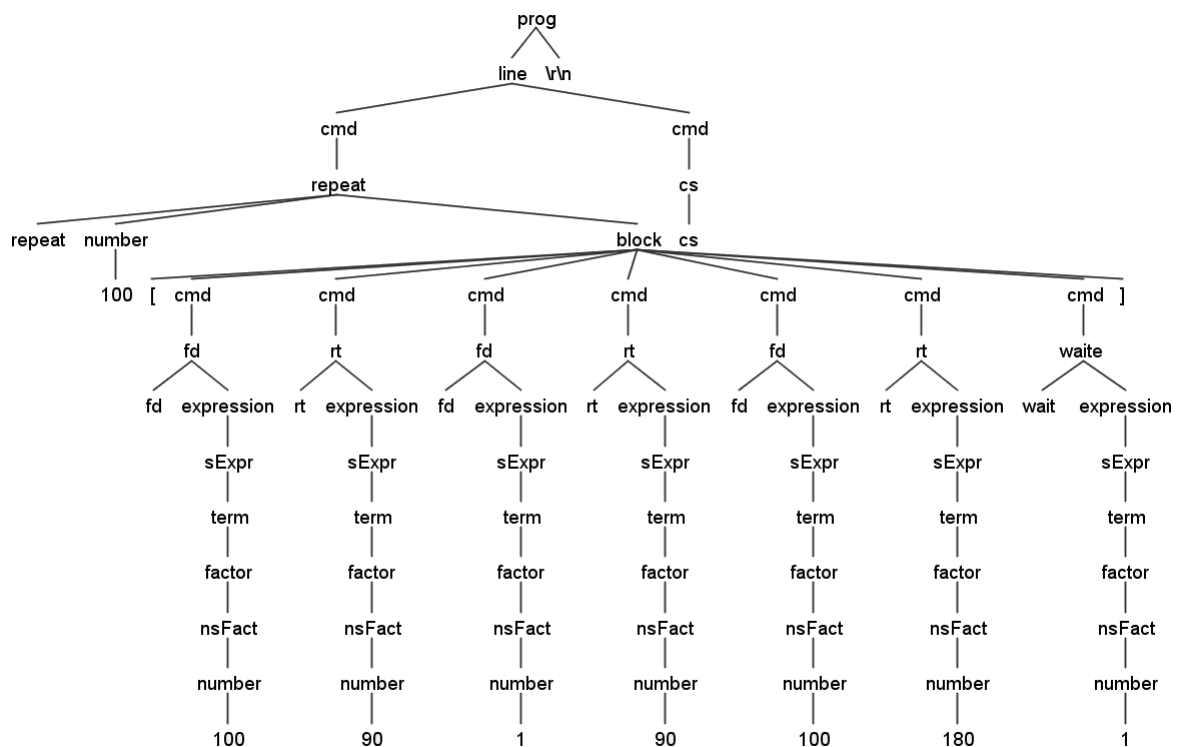


Figure 6.10. Timing benchmark program, which draws a filled square using the command `wait`. Parsing it results in a tree with 81 nodes

In this experiment we work on a rather large tree with lots of nodes. The most special node in this experiment is the one belonging to the `wait` command, because it causes the execution to pause for one second before continuing. We expect all data points to lie beyond 1000ms due to this. If a data point has a performance of exactly 1 second, it implies, that this particular data point did not take any time other than the one used to wait.

6. Evaluation

Experiment 1: Execution time for drawing and parse tree traversal

In this first experiment we focus on the time it takes to traverse the parse tree and to draw onto the canvas in addition to the time the system spends waiting due to the users commands. In the following plot you see the results of our experiments:

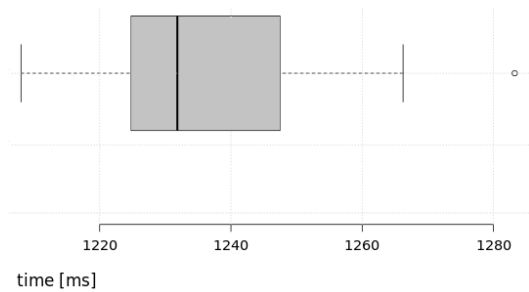


Figure 6.11.: Timing benchmark results, Experiment 1

	performance [ms]
upper whisker	1266.22
3rd quartile	1247.47
median	1231.84
1st quartile	1224.78
lower whisker	1208.01

We notice, that all of our data points lie within a spectrum of 1100ms to 1300ms, which implies that our system took roughly 100ms to 300ms to traverse the parse tree and to draw onto the canvas. We cannot make a better statement due to the fact that we implemented the asynchronous waiting using the JavaScript command `setTimeout`. This command adds operations to the queue as soon as some timeout is over and it is executed as soon as the main thread is idle. But it may happen that an operation stays in the queue even when the timeout is over, because the main thread is still busy.

Experiment 2: Cumulative execution time

In the second experiment we measured the time to prepare and visit the parse tree as well as drawing onto the canvas, in addition to the time we spend waiting due to the user's input. We observed the following results:

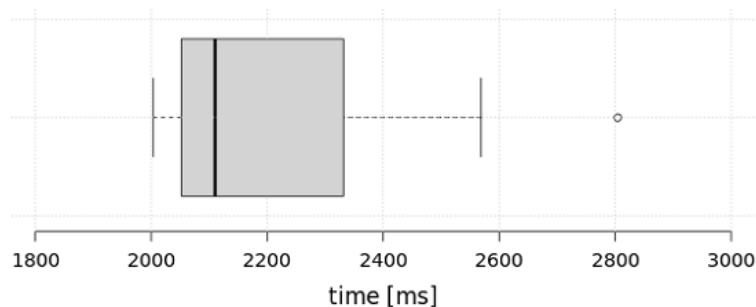


Figure 6.12.: Timing benchmark results, Experiment 2

	performance [ms]
upper whisker	2568.97
3rd quartile	2331.56
median	2110.17
1st quartile	2051.98
lower whisker	2003.00

In this experiment we observed the worst performance. In addition to the one second we spend waiting we also spend more than a second for building up the parse tree, traversing it and drawing onto the canvas.

These results comply with our expectations in terms of performance and overall correlation between the runtimes for building the parse tree and actually visiting it. We found out, that even the most elaborate programs in the booklet can be solved in under half a minute. We calculated the average number of drawing instructions, which can be performed in one second and we found values in the range of 18604 draw instructions per second for the small benchmark, 25352 draw instructions per second for our loop benchmark and 28739 draw instructions per second for the large benchmark. Thus we have roughly 20000 draw instructions per second for an average program. We saw that the building times for the parser vary in our benchmarks quite heavily. We see that building up a parse tree is very cheap in comparison to traversing it and drawing to the canvas. Only with small programs we even see a difference. We have to exclude the timing benchmark from those statements, though, because we want to measure the actual number of CPU cycles spent, rather than the wall-clock time (i.e. without waiting periods).

6.2. System Testing

In order to check for errors we implemented all 89 exercises in ABZs booklet. Some implementations were provided by Igo Schaller or Giovanni Serafini. For their support we are very thankful. We tested against all those programs and ensured that all of them are working properly in our system. A reference output is not needed due to the fact that solutions usually involve a graphical output, which can be checked easily with the expected output, which is given in a graphical form as well.

The test suit covers all topics from the booklet, which means the following topics:

- Basic commands
- Repetition
- modular design
- Polygons and circles
- Using parameters
- Passing parameters on to subprograms
- Animations

All concepts and basically all commands presented in the booklet are represented by at least one test program in our test suit.

The most interesting errors we found during the integration testing are the following two. We are shortly going to explain the errors and their implications for our system:

- *Erasure mode leaves traces on canvas*
Browsers are free how to implement the rendering of certain parts of the functionality which is provided by JavaScript. Especially on the canvas we found massive variations between different browsers. The following program turned out to leave a trace on the

6. Evaluation

canvas even though it is not supposed to:

```
1 rt 45 repeat 4[fd 100 rt 90] pe repeat 4[fd 100 rt 90] ppt
```

The problem is, that any pixel on the canvas can either be colored or not. If we draw a line diagonally, we need to decide which pixels to touch. Some browser decide to repaint pixels even if they are not perfectly covered by the line and thus those pixel traces are not eliminated properly in erasure mode. We solved the issue by increasing the pen-width by 1px when we are in erasure mode, and therefore guaranteeing that all old traces are eliminated properly. The implication of this solution is, that whenever we use erasure mode we are over-approximation and therefore sometimes deleting too much.

- *Performance with stroke*

We observed that drawing larger programs (like our large benchmark program) take a very long time to terminate, much longer than anticipated. A very simple program like the following took us 5741ms (almost 6 seconds) in average:

```
1 repeat 12[repeat 360[fd 1 rt 1] rt 360/12]
```

Moreover we noted the re-executing the same program without resetting the canvas in between increases the running time dramatically. In Figure 6.13 we demonstrate the effect for the same program as before:

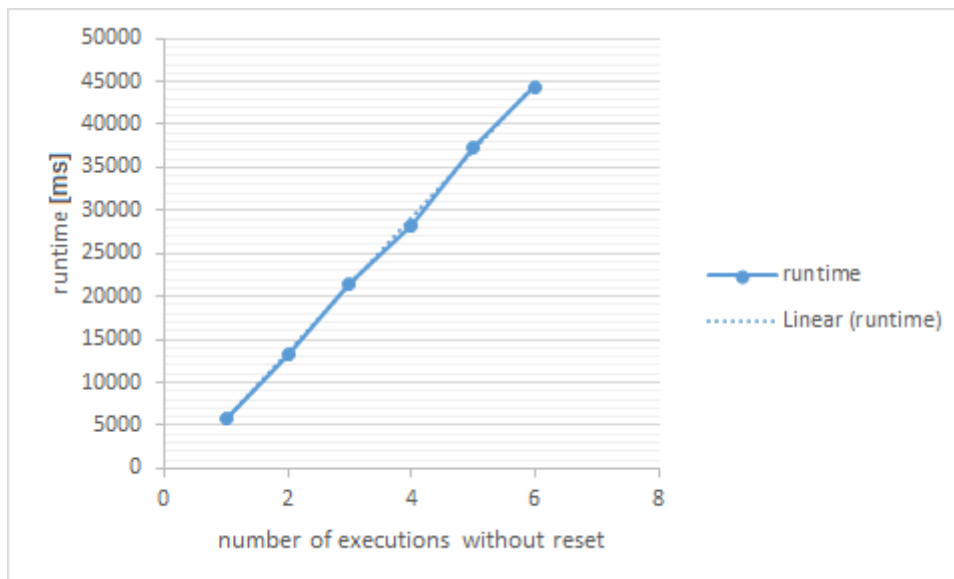


Figure 6.13.: Linear increase in runtime when re-executing the program without clearing the canvas

We observed that there was a linear increase in terms of runtime when re-executing the same program without resetting the canvas in between. We expected to observe a constant runtime, no matter how often the same program is executed. The reason for this surprising behavior is that the command *stroke*, which is used to draw onto the canvas, reprints every line in the current line-path on the canvas. Such a line-path is getting longer and longer

if we don't reset the canvas in between, which was the reason for the linear increase in runtime. We solved the problem by ensuring, that the stroke command is only executed once for every user input, in the end. If the user decided to run the program in slower execution mode to observe the progress or if he uses the wait command, we are forced to reprint the commands step by step however. Otherwise there would be no visible progress on the screen.

We appended all 89 programs which we used to test our system to Appendix A.2. The programs cover the vast majority of all XLogo commands. The system testing convinced us that the system is reliable enough to be used in a class room setting. In the next section we will explain what our usability study in a class room setting looked like and what results we got.

6.3. User study

Over the course of this thesis we conducted a user study at a primary school in Sempach. For our user study we were allowed to use our system in a usual XLogo class-room setting with 18 pupils, which were all between 11 and 12 years old. In this section we are going to present the goals, we had in mind when conducting the user study, as well as the setup, results and conclusions we gained from its realization.

6.3.1. Goals

The goal of this user study was to get an understanding of the children's reaction to the new environment and to check it in a realistic setting. In addition, due to the large number of users, we were able to use the pupils as indirect testers and we collected their feedback about design flaws and proposals for improvement.

6.3.2. Setup

The user study was conducted in a class-room setting, i.e. 18 netbook in the same room, using the same network and the same browser. The netbooks had a screen size of 10.1" and run Windows 10. We expected no connection to the internet, which is why we prepared 10 USB sticks with the offline version of our IDE. They used Microsoft's new default browser, Edge. The children themselves already knew XLogo, the predecessor version of the XLogo IDE, due to a previous course, which they took one year ago. In addition to that they had another session one week before our experiment to refresh their memories in terms of the language and the handling of the predecessor IDE. We introduced them within a 10 minutes session to the new environment and introduced the content of the day (chapter 4 in the ABZ booklet, about polygons and circles). They used the IDE during a two-hour programming session and reported bugs, if they found one. During the last 15 minutes we distributed a feedback form to check whether they understood the general structure of the new IDE in comparison to the structure of

6. Evaluation

the old IDE, to collect their feedback about individual components and a general measure for their satisfaction using our new IDE. The feedback form can be found in Appendix A.3.

6.3.3. Evaluation

First, we were unpleasantly surprised by the fact that in the setup we found in Sempach, we did not have write access to the file system on all the pupil's computers. Thus we were only able to run our offline version on 10 computers. But since we had 18 students in the class, we decided to use the online version instead.

In terms of general understanding (which component corresponds to which graphical user interface element) we got a 83.3% success rate. Only three pupils confused the purposes of some of the components, while 15 pupils understood their purpose correctly. The three mistakes were about which graphical user interface belongs to the editor and which belongs to the text input line.

In the feedbacks they gave us about the individual UI components we found the following recommendations for improvements. The students think that the right hand side of the website should be larger in comparison to the left hand side. They wish for a feature, which makes entries in the history clickable, such that they would reappear in the input line when they are clicked onto. They found a bug in the canvas. When moving the image around the canvas left a trace. They did not realize the feature, that the website can be resized by hand, and they did not figure out to use the arrow up key to pull old commands back up into the input line.

The following figure demonstrates the feedback in general terms about how they felt using our IDE:

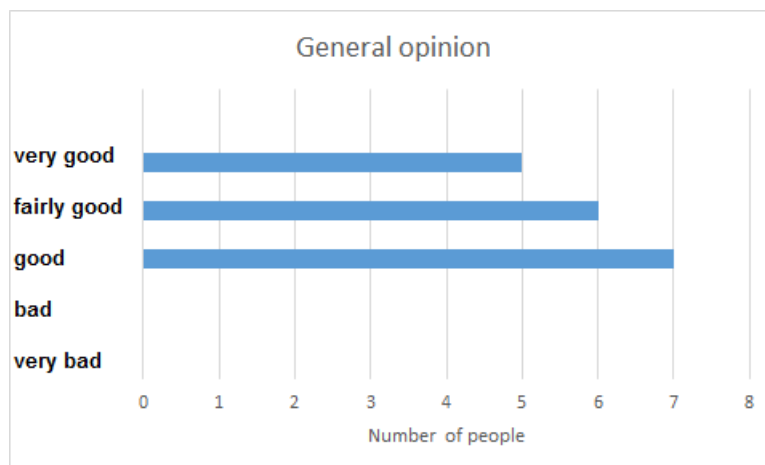


Figure 6.14.: Most students assessed the IDE to be between good or even very good

Out of 18 pupils, seven thought the system was good, 5 thought it was very good and 6 thought it was somewhere between good and very good.

6.3.4. Conclusions

From this user study we conclude that the default ratio between the left and the right hand side of the website is not satisfactory. Moreover we need to provide some visual hint that the website offers the feature to resize the two parts. Also they seem to like the entries in the history to be clickable, which was not yet implemented in our systems. We offer only the usage of the arrow up key with the same effect. In addition we found a few bugs, but in general the children reported to feel comfortable using the IDE.

6. *Evaluation*

7

Conclusion

7.1. Conclusion

We conduct programming courses for primary-school pupils using a dialect of the programming language Logo. The programming IDEs which were used for this purpose so far no longer work on a few platforms. Therefore we wrote a new programming IDE for XLogo, which is based on newest web-technologies. The program works on the client side and can thus be used both in online and offline mode. The system was designed to be platform-independent, easy to use, intuitive and, most importantly, supportive for newcomers who are learning to program the first time. Our programming IDE runs instantly in any web browser, without installation and offers an uncluttered interface, with visual cues directly in the user interface to guide an help pupils as they learn.

In comparison to previous versions of the programming IDE, which have multiple windows that obstruct the users view, we implemented the website as a single-page application, where all widgets are directly visible at a single glance. This reduces the cognitive load and facilitates the interaction with the programming environment – both important considerations for our target audience who have not been exposed to algorithmic thinking and programming before. The website provides all basic features which were possible with our predecessors, including the interpreter and the ability to load and store the editor's contents. Moreover, we implemented our canvas to be pannable for increased usability. To encourage interactive learning we developed teaching material, which incorporates group work where students practice their communication skills. As a further addition, we extend our IDE with a collaborative programming mode to synchronize editors in real-time. Our page is reactive, the editor UI element is resizable, provides syntax checking as well as syntax highlighting, these are all useful. We used a framework

7. Conclusion

called Angular2, which works on the basis of a composite and Model-View-Controller (MVC) design pattern; the UI is decomposed into a tree of independent components, which reflect the functional and visual structure of our website. Data and components are decoupled and are connected only when needed using services. We used JavaScript and TypeScript as developmental programming language and another library called ANTLR to generate the parser.

We conducted a user study with 18 pupils, which showed that the new programming environment is perceived positively and that the children feel at ease when working with our software. Feedback reflected that the structure of our IDE is intuitive and well-understood by most of the pupils. Our solution is purpose-built to fit with a pre-existing curriculum; we tested it against a master solution for all exercises in the booklet provided by ABZ. One limitation which emerged is that our system is unresponsive on long-running computations due to limitations of the browser. We resolved the issue by forcing the main thread to stop after a fixed number of calculation steps and then ask explicitly whether to continue. In our opinion the project fulfills its purpose as a system, which is intuitive, easily accessible, platform-independent and designed using newest web-technologies.

With the introduction of ‘Lehrplan 21’, Swiss teachers are in need of teaching material. In this thesis we tackle this problem by implementing a system, which integrates teaching material, exercises and a drawing environment, all designed to guide pupils while they programming. In the process of developing this work, several possibilities suggested themselves. The recent popularity of MOOCs is one such direction; our platform already provides the basic functionality to draw and include pre-defined programs; with the addition of a chat, the transition to and from videos and an automated tutorial, it could be used to learn without the aid of a teacher. A MOOC for Logo is ideally implemented on web-technologies, because they provide a good basis for interactivity and they are easily accessible without the need for specialized software. Our web-solution is accessible from anywhere, without installation and independent from the operating system, device type and screen size. We offer an approach with a streamlined design, which reduces the cognitive load, because everything the user needs is available on a single page. The editor and its content are visible at the same time as the input area and the canvas are. If the user decides not to use the editor, he can easily collapse it by resizing the corresponding parts of the website. We implemented syntax highlighting as well as syntax checking on the editor, which allows the user to see and correct his mistakes on the fly, which reduces the cognitive load for the user due to the direct feedback about the syntactic correctness of the editor. In addition, we improved on the usability of the canvas by allowing the user to pan it in any direction. This feature would be helpful if the user decides to draw a large picture or fears to waste space on the canvas otherwise.

7.2. Future Work

A programming environment is only one part of a CS curriculum; we will now propose further additions which are possible continuations for our work. We see five main ways to go: Improve the graphical user interface, implement cooperative programming, implement an in-

finitely movable canvas, make history elements clickable and resolve the separation between user interface and interpreter in a manner, which is more independent from the user.

7.2.1. Graphical user interface

The UI is the most vital component of a web application, due to its functionality of being the only interface, with which the user interacts. It must be easy to use. We propose to conduct a comprehensive user study focusing on the graphical user interface to determine whether it is intuitive, how children react to it and how they are learning with it (e.g. using eye-tracking). Web platforms allow easy instrumentation of such experiments. In our user study we found a few desired on the pupil's side, which could be further analyzed and implemented. Two of those features are to make the history resizable and its entries clickable. This would improve the usability and improve the graphical user interface to become even more intuitive.

We implemented a basic means of reactive design on our application, in terms of the turtle's size, which adjusts to the size of the window as well as the header component whose buttons and text shrink and grow according to the windows size, similar to the absolute sizes of all basic graphical user interfaces on our website. Though the implementation of this reactive design was written manually, it might still contain design flaws. Some libraries (like Bootstrap) provide a structure, which helps to build websites with a reactive design.

7.2.2. Cooperative Programming

Pupils learn more effective in groups [Gokhale 1995] and programming is less errorprone if it is performed in groups. We have built a first prototype and added a cooperative coding feature into our existing IDE. Cooperative coding would force the children to work together on one large programming exercise. This leads the children to think about proactive problem managements strategies, to break problems into smaller problems, to work in groups, to agree upon strategies to finally reach a common goal of solving a large exercise as a team. To demonstrate how group work might be used in a classroom we also developed a set of more challenging exercises which feature an introduction to monospaced text and more sophisticated geometry A.4. We implemented a prototype providing this feature using a library called *firebase*. This library uses the ACE editor, shares the content of the editor with every user who uses the same id and handles all the synchronization for us. We implemented a prototype with this library, but currently encounter an issue. Sending and receiving messages from and to the firebase server is not possible, because we are confronted with a problem in the firebase library. We receive a silent error which is thrown in a library method (push) when we try to instantiate the editor's content in the ACE editor (see README.txt). We encapsulated the bug by building a minimal non-working code sample, which can be found as a documented comment in our code. More information about firebase itself can be found here: <https://firebase.google.com/docs/web>. For this reason we disabled the feature at the moment.

7. Conclusion

7.2.3. Movable Canvas

Using a canvas in web-browsers directly implies that we are given a fixed number of pixels which can be manipulated. Logo programs are inherently infinitely-large; they offer commands to draw arbitrarily large images, which is why we would like to give the user the impression of an infinitely-large canvas. Towards this we implemented a prototype version which features a movable canvas, but not an infinite one. The limitation of a movable canvas over an infinite one is that we are still not able to show infinitely large images, but only a larger part of them. We could implement the functionality using a backing buffer, onto which all the drawing operations performed. A separate canvas is used to show a specific scene from the backing-buffer which the user is currently looking at. Our use case is a bit more difficult than this simple illustration, because we need to support resizable parts of the window and thus the actual size and the origin of the coordinate system of the canvas may be dynamically changing.

7.2.4. Responsiveness

In a programming IDE the user can introduce heavy computational overheads to the programming environment. It can even happen that he launches an infinitely-long running sequence of commands, by running an infinite loop. Users who ‘break’ the program by doing so have an interest in interrupting their program during its computation. For this reason we need a possibility to interrupt a computational from the user interface directly. Unfortunately it is not possible to statically analyze programs and detect infinite loops, because of the halting problem. A dynamic approach is necessary. Browsers only offer a single main thread, which has exclusive access to the DOM and coordinate all accesses to the graphical user interface. The UI is event-driven, which means that the main UI thread runs an infinite loop which processes a queue of asynchronous tasks, which is set up in advance in the form of a callback. Having only a single main thread is problematic, especially if we have long computations, because the XLogo interpreter is forced to share the same main thread which can block UI elements because interactions with the user interface are queued up until the main thread is idle again (which might take a very long time for long-running programs). The UI thread may become unresponsive, and thus the user does not see any reaction to an interaction with the user interface. The main thread is blocked by the long computation which blocks further pending callbacks unhandled until the potentially long-running computation runs to completion.

We found the following three approaches to the above-mentioned problem:

1. *Web workers:*

Browsers already offer a limited form of parallelism. Web workers are background threads, which are standardized in HTML5 and which can run simultaneously to the main thread. If a long computation does not involve the user interface, we can pass the computation to a background thread. Workers and the main thread do not share memory and are forced to communicate via messages. In our case this approach is not suitable because our long tasks involve operations on the UI, furthermore XLogo programs are primarily about drawing, which is UI-intensive. If we wanted to use workers, we would also need to proxy all UI operations from the web worker to the main thread. Common libraries such as Angular2 are not intended to be used in this way and would require extensive

redesign to use messaging.

2. *Chunking computations down into multiple small pieces:*

The second approach is to break the single long computation down into many smaller ones. This means each chunk is bounded to a limited number of node traversals in the parse tree and so the XLogo program does not monopolize the main UI thread. Whenever the main thread is not working on one of these smaller chunks of operations, it is responsive and reacts to the user. In JavaScript this is done using the *setTimeout* command, which takes a delay and a callback which will be executed at some point after this interval. This strategy involves some redesign too due to the way *setTimeout* forks one operation into two. If we decide to use *setTimeout* on a large operation which is supposed to be run sequentially, we need to interrupt the whole operation and restart the remaining part after the timeout. This approach has an impact on the performance of our website and, moreover, we need to find a good ratio between the amount of time we spend waiting (and being responsive to the user) and executing our computation.

3. *Set timeout and ask user how to proceed:*

A last option is to actually block the user interface and let the main thread perform its operations, but to set a maximum amount of time it is allowed to spend working on our computation (by setting a threshold on the number of computation steps it is allowed to perform in one chunk). After this timeout, the user is asked for a confirmation whether the execution should still be continued or not. This solution is optimal in terms of its performance but, on the downside, it is some of the time not responsiveness to interactions on the user interface.

We think approaches 2 and 3 are promising. We decided to take solution 3. As a threshold we chose one million operations, which is just about enough to complete every exercise from ABZ's booklet. If the user decides to continue with the computation, the threshold is increased by a factor of two. Browsers have had limited support for parallelism; this is dealt with better in Android's *asynTasks* which shifts the burden of managing the communication between background thread and main thread using messages away from the programmer by providing a better framework, which takes care of this communication for us.

7. Conclusion

A

Appendix

A.1. XLogo grammar

```
1 grammar Logo;
2
3 prog: (line? EOL) + line?;
4 line: cmd + comment?| comment| print comment?| procedureDeclaration;
5 cmd: repeat| fd| bk| rt| lt| cs| pu| pd| ht| st| pe| ppt| home| ↔
   ↔ setx| sety| setxy| setpc| setsc| make| procedureInvocation| ↔
   ↔ ife| stop| wait| whilee| wash| setpw| circle| setheading;
6 procedureInvocation: name expression*;
7 procedureDeclaration: 'to' name parameterDeclarations* EOL? (line? ↔
   ↔ EOL) + 'end';
8 parameterDeclarations: ':' name (parameterDeclarations)*;
9 func: random| sin| cos| tan| arcsin| arccos| arctan| abs| sqrt| ↔
   ↔ mod| power| log10;
10
11 repeat: 'repeat' number block;
12 whilee: 'while' '[' expression ']' block;
13 block: '[' cmd + ']';
14 ife: 'if' expression block| 'if' expression block block;
15 expression: sExpr (comparisonOperator sExpr)?;
16 comparisonOperator: '<' | '>' | '=' | '<=' | '>=' | '!=';
17
18 sExpr : term (('+'| '-'| '|' | '|') term)*;
19 term: factor (('*' | '/' | '&&') factor)*;
20 factor: ('+'| '-')? nsFact;
```


A. Appendix

```
21 nsFact: (number | deref | func) | LPAREN expression RPAREN | '!' ↵
    ↵ factor;
22 value: STRINGLITERAL | expression | deref;
23
24 deref: ':' name;
25 colorname: 'black' | 'red' | 'green' | 'yellow' | 'blue' | 'magenta' | ↵
    ↵ 'cyan' | 'white' | 'gray' | 'lightgray' | 'darkred' | ↵
    ↵ 'darkgreen' | 'darkblue' | 'orange' | 'pink' | 'purple' | 'brown';
26 rgb: '[' expression expression expression ']';
27 colornumber: expression;
28 make: 'make' STRINGLITERAL value;
29
30 print: ('print' | 'pr') value | ('print' | 'pr') '[' value+ ']' | ↵
    ↵ ('print' | 'pr') quotedstring;
31 quotedstring: '[' (quotedstring | ~ ') '* ']';
32 name: STRING;
33
34 fd: ('fd' | 'forward') expression;
35 bk: ('bk' | 'backward') expression;
36 rt: ('rt' | 'right') expression;
37 lt: ('lt' | 'left') expression;
38 cs: 'cs' | 'clearscreen';
39 pu: 'pu' | 'penup';
40 pd: 'pd' | 'pendown';
41 pe: 'pe' | 'penerase';
42 ppt: 'ppt' | 'penpaint';
43 ht: 'ht' | 'hideturtle';
44 st: 'st' | 'showturtle';
45 setpc: ('setpc' | 'setpencolor') colornumber | ↵
    ↵ ('setpc' | 'setpencolor') rgb | ('setpc' | 'setpencolor') ↵
    ↵ colorname;
46 setsc: ('setsc' | 'setscreencolor') rgb | ('setsc' | ↵
    ↵ 'setscreencolor') colorname | ('setsc' | 'setscreencolor') ↵
    ↵ colornumber;
47 home: 'home';
48 stop: 'stop';
49
50 setxy: 'setxy' expression expression;
51 setx: 'setx' expression;
52 sety: 'sety' expression;
53
54 random: ('random' | 'rnd' | 'ran') expression;
55 sin: 'sin' expression;
56 cos: 'cos' expression;
57 tan: 'tan' expression;
58 arcsin: 'arcsin' expression;
59 arccos: 'arccos' expression;
60 arctan: 'arctan' expression;
61
```

```

62 abs: 'abs' expression;
63 sqrt: 'sqrt' expression;
64 mod: 'mod' expression expression;
65 power: 'power' expression expression;
66 log10: 'log' expression expression;
67
68 waite: 'wait' expression;
69 wash: 'wash';
70 setpw: ('setpw' | 'setpenwidth') expression;
71 circle: 'circle' expression;
72 setheading: 'setheading' expression;
73
74 number: NUMBER| pie| ee;
75 comment: COMMENT;
76 pie: 'pi';
77 ee: 'e';
78
79 STRINGLITERAL: '"' STRING;
80 STRING: [a-zA-Z] [a-zA-Z0-9_]*;
81 LPAREN: '(';
82 RPAREN: ')';
83 NUMBER: [0-9] +;
84 COMMENT: ';' ~ [\\r\\n]*;
85 EOL: '\\r'? '\\n';
86 WS: [ \\t\\r\\n] -> skip;

```

A.2. Master Solution to the Exercises in ABZ Booklet

Aufgabe 1:

```

1 fd 100 rt 90 fd 150 rt 90 fd 50 lt 90 fd 150 rt 90 fd 50

```

Aufgabe 3:

```

1 a) fd 200 rt 90 fd 200 rt 90 fd 200 rt 90 fd 200 rt 90
2 b) rt 90 fd 100 lt 90 fd 50 rt 90 fd 100 rt 90 fd 500 lt 90 fd 100 ←
   ↪ lt 90 fd 50 rt 90 fd 100 rt 90 fd 50 lt 90 fd 100
3 c) lt 90 fd 10 lt 90 fd 20 lt 90 fd 30 lt 90 fd 40 lt 90 fd 50 lt ←
   ↪ 90 fd 60 lt 90 fd 70 lt 90 fd 80 lt 90 fd 90
4 d) fd 100 rt 90 fd 100 rt 90 fd 100 rt 90 fd 100 rt 90 rt 90
5 fd 100 rt 90 fd 100 rt 90 fd 100 rt 90 fd 100 rt 90 rt 90
6 fd 100 rt 90 fd 100 rt 90 fd 100 rt 90 fd 100 rt 90 rt 90
7 fd 100 rt 90 fd 100 rt 90 fd 100 rt 90 fd 100 rt 90 rt 90

```

Aufgabe 4:

A. Appendix

```
1 ) fd 50 rt 90 fd 50 rt 270 fd 50 rt 90 fd 50 rt 270 rt 50 rt 90 fd ←  
   ↷ 50 rt 90 fd 50 rt 270 fd 50 rt 90 fd 50 rt 270 fd 50 rt 90 ←  
   ↷ fd 50
```

Aufgabe 5:

```
1 fd 50 rt 90 fd 50 rt 90 fd 50 rt 90 fd 50 rt 90 rt 90 rt 100 rt 90 ←  
   ↷ fd 50 bk 50 lt 90 fd 50 lt 90 fd 50 rt 90 fd 50 rt 90 fd 50 ←  
   ↷ rt 90 fd 50 rt 90
```

Aufgabe 6:

```
1 repeat 4[fd 75 lt 90]
```

Aufgabe 7:

```
1 repeat 6[fd 50 rt 60]
```

Aufgabe 8:

```
1 repeat 4[fd 200 rt 90]
```

Aufgabe 9:

```
1 repeat 3[fd 100 rt 120]
```

Aufgabe 10:

```
1 a) repeat 10[fd 20 rt 90 fd 20 lt 90]  
2 b) repeat 5[fd 50 rt 90 fd 50 lt 90]  
3 c) repeat 20[fd 10 rt 90 fd 10 lt 90]
```

Aufgabe 11:

```
1 a) repeat 8[fd 150 bk 150 lt 45]  
2 b) repeat 16[fd 100 bk 100 lt 22.5]
```

Aufgabe 12:

```
1 a) repeat 10[repeat 4[fd 20 rt 90] rt 90 fd 20 lt 90]  
2 b) repeat 4[repeat 4[fd 30 rt 90] fd 30]
```

Aufgabe 13:

```
1 repeat 4[repeat 4[fd 100 rt 90] fd 100 rt 90 fd 100 lt 90]
```

Aufgabe 14:

```
1 repeat 4[repeat 4[fd 100 rt 90] rt 90]
```

Aufgabe 15:

```
1 repeat 4[fd 100 bk 100 rt 90 pu fd 100 pd lt 90]
```

Aufgabe 16:

```
1 repeat 7[repeat 4[fd 20 bk 20 rt 90] rt 90 pu fd 70 lt 90 pd]
```

Aufgabe 17:

```
1 to quadrat30
2 repeat 4 [fd 30 rt 90]
3 end
4
5 to ausrichten30
6 bk 30
7 end
8
9 to reihe4
10 repeat 4 [QUADRAT30 AUSRICHTEN30]
11 end
```

Aufgabe 18:

```
1 to quadrat100
2 repeat 4 [fd 100 rt 90]
3 end
4
5 to ausrichten100
6 fd 100 rt 90 fd 100 lt 90
7 end
8
9 to afg18
10 repeat 4 [QUADRAT100 AUSRICHTEN100]
11 end
```

Aufgabe 19:

```
1 to treppe20
2 fd 20 rt 90 fd 20
3 end
4
5 to ausrichtentreppe
6 lt 90
7 end
8
9 to afg19
10 repeat 10 [TREPPE20 AUSRICHTENTREPPE]
11 end
```

Aufgabe 20:

A. Appendix

```
1 to line
2 fd 150 bk 150
3 end
4
5 to afg20
6 repeat 8 [LINE lt 45]
7 end
```

Aufgabe 21:

```
1 to strahl
2 fd 100 bk 200 fd 100
3 end
4
5 to strahlausrichten
6 lt 60
7 end
8
9 to afg21
10 repeat 4 [STRAHL STRAHLAUSRICHTEN]
11 end
```

Aufgabe 22:

```
1 to line100
2 fd 100
3 end
4
5 to line100move
6 pu bk 100 rt 90 fd 100 lt 90 pd
7 end
8
9 to afg22a
10 repeat 4 [line100 line100move]
11 end
```

```
1 to kreuz
2 repeat 4 [fd 20 bk 20 lt 90]
3 end
4
5 to kreuzmove
6 pu rt 90 fd 70 lt 90 pd
7 end
8
9 to afg22b
10 repeat 7 [KREUZ KREUZMOVE]
11 end
```

Aufgabe 23:

```
1 to quadrat20
2 repeat 4 [fd 20 rt 90 ]
3 end
4
5 to ausrichten20
6 rt 90 fd 20 lt 90
7 end
8
9 to reihe10
10 repeat 10 [QUADRAT20 AUSRICHTEN20]
11 end
12
13 to afg23
14 REIHE10 fd 20 lt 90 fd 200 rt 90
15 end
```

Aufgabe 24:

```
1 to quadrat20
2 repeat 4 [fd 20 rt 90 ]
3 end
4
5 to ausrichten20
6 rt 90 fd 20 lt 90
7 end
8
9 to reihe10
10 repeat 10 [QUADRAT20 AUSRICHTEN20]
11 end
12
13
14 to afg23
15 REIHE10 fd 20 lt 90 fd 200 rt 90
16 end
17
18 to afg24
19 repeat 3 [afg23]
20 end
```

Aufgabe 25:

```
1 to quadrat50
2 repeat 4 [fd 50 rt 90]
3 end
4
5 to quadrat75
6 repeat 4 [fd 75 rt 90]
7 end
8
```

A. Appendix

```
9 to quadrat125
10 repeat 4 [fd 125 rt 90]
11 end
12
13 to quadrat150
14 repeat 4 [fd 150 rt 90]
15 end
16
17 to afg25
18 quadrat50 quadrat75 quadrat100 quadrat125 quadrat150
19 end
```

Aufgabe 27:

```
1 to haus
2 rt 90
3 repeat 4 [fd 50 rt 90]
4 lt 60 fd 50 rt 120 fd 50 lt 150
5 end
6
7 to hausreihe
8 repeat 5 [HAUS rt 90 pu fd 50 lt 90 pd]
9 end
10
11 to afg27
12 repeat 3 [hausreihe lt 90 pu fd 500 rt 90 fd 150 pd]
13 end
```

Aufgabe 29:

```
1 to fett
2 fd 100 rt 90 fd 1 rt 90 fd 100 rt 180
3 end
4
5 to afg29
6 repeat 100 [fett]
7 end
```

Aufgabe 31:

```
1 to fett40
2 fd 40 rt 90 fd 1 rt 90 fd 40 rt 180
3 end
4
5 to schwarz40
6 repeat 40 [fett40]
7 end
```

Aufgabe 32:

```
1 to fett40
2 fd 40 rt 90 fd 1 rt 90 fd 40 rt 180
3 end
4
5 to schwarz40
6 repeat 40 [fett40]
7 end
8
9 to afg32
10 repeat 4 [schwarz40 rt 90 fd 40 lt 90]
11 end
```

Aufgabe 33:

```
1 to fett40
2 fd 40 rt 90 fd 1 rt 90 fd 40 rt 180
3 end
4
5 to schwarz40
6 repeat 40 [fett40]
7 end
8
9 to afg33
10 repeat 4 [schwarz40 fd 40]
11 bk 40
12 repeat 3 [bk 40 schwarz40]
13 end
```

Aufgabe 34:

```
1 to fett40
2 fd 40 rt 90 fd 1 rt 90 fd 40 rt 180
3 end
4
5 to schwarz40
6 repeat 40 [fett40]
7 end
8
9
10 to afg34
11 repeat 4 [schwarz40 rt 90]
12 rt 90
13 schwarz40
14 end
```

Aufgabe 35:

```
1 to quad100
2 repeat 4 [fd 100 rt 90]
```


A. Appendix

```
3 end
4
5 to fett100
6 fd 100 rt 90 fd 1 rt 90 fd 100 rt 180
7 end
8
9 to schwarz100
10 repeat 100 [fett100]
11 end
12
13 to afg35
14 repeat 2 [schwarz100 quad100 rt 90 fd 100 lt 90]
15 end
16
17 to afg35a
18 repeat 2 [schwarz100 quad100 rt 90 fd 100 lt 90]
19 end
20
21 to afg35b
22 repeat 2 [quad100 rt 90 fd 100 lt 90 schwarz100]
23 end
```

Aufgabe 36:

```
1 to unterteil
2 rt 90
3 repeat 4 [fd 50 rt 90]
4 lt 90
5 end
6
7 to dach
8 rt 30
9 repeat 3 [fd 50 rt 120]
10 lt 30
11 end
12
13 to haus1
14 unterteil
15 dach
16 end
```

Aufgabe 37:

```
1 to unterteil
2 rt 90
3 repeat 4 [fd 90 rt 90]
4 lt 90
5 end
6
7 to dach
```

```
8  rt 30
9  repeat 3 [fd 90 rt 120]
10 lt 30
11 end
12
13 to tuer
14 repeat 2 [fd 70 rt 90 fd 30 rt 90]
15 end
16
17 to fenster
18 repeat 4 [fd 30 rt 90]
19 end
20
21 to haus2
22 unterteil
23 dach
24 bk 90 rt 90 fd 10 lt 90
25 tuer
26 rt 90 fd 40 lt 90 pu fd 40 pd
27 fenster
28 pu fd 50 rt 90 fd 80 lt 90 pd
29 end
30
31 to hauszeile2
32 repeat 4 [haus2]
33 end
34
35 to wohnsiedlung
36 repeat 3 [hauszeile2 pu fd 150 lt 90 fd 580 rt 90 pd]
37 end
```

Aufgabe 38:

```
1  to afg38a
2  repeat 5 [fd 180 rt 72]
3  end
4
5  to afg38b
6  repeat 12 [fd 50 rt 30]
7  end
8
9  to afg38c
10 repeat 4 [fd 200 rt 90]
11 end
12
13 to afg38d
14 repeat 6 [fd 100 rt 60]
15 end
16
```

A. Appendix

```
17 to afg38e
18 repeat 3 [fd 200 rt 120]
19 end
20
21 to afg38f
22 repeat 18 [fd 20 rt 20]
23 end
24
25 to afg38g
26 repeat 7 [fd 100 rt 360/7]
27 end
```

Aufgabe 39:

```
1 to afg39a
2 repeat 360 [fd 1 rt 1]
3 end
4
5 to afg39b
6 repeat 180 [fd 3 rt 2]
7 end
8
9 to afg39c
10 repeat 360 [fd 2 rt 1]
11 end
12
13 to afg39d
14 repeat 360 [fd 3.5 rt 1]
15 end
```

Aufgabe 40:

```
1 to afg40a
2 repeat 360 [fd 0.3 rt 1]
3 end
4
5 to afg40b
6 repeat 360 [fd 5 rt 1]
7 end
```

Aufgabe 41:

```
1 to afg41a
2 repeat 180 [fd 2 rt 1]
3 end
4
5 to afg41b
6 rt 180 repeat 180 [fd 1.5 rt 1]
7 end
8
```

```
9 | to afg41c
10 | repeat 4 [afg41a rt 90]
11 | end
```

Aufgabe 42:

```
1 | to dreieck
2 | repeat 3 [fd 200 rt 120]
3 | end
4 |
5 | to quadrat
6 | repeat 4 [fd 200 rt 90]
7 | end
8 |
9 | to kreis
10 | repeat 360 [fd 2.5 rt 1]
11 | end
12 |
13 | to afg42a
14 | quadrat
15 | fd 200
16 | rt 30
17 | dreieck
18 | end
19 |
20 | to afg42b
21 | rt 30
22 | dreieck
23 | rt 60
24 | fd 100
25 | kreis
26 | end
```

Aufgabe 43:

```
1 | to eck12
2 | repeat 12 [fd 70 rt 30]
3 | end
4 |
5 | to afg43
6 | repeat 18 [eck12 rt 20]
7 | end
```

```
1 | to afg43pre
2 | repeat 36 [repeat 7 [fd 100 rt 360/7] rt 10]
3 | end
```

Aufgabe 45:

```
1 | to kreis3
```

A. Appendix

```
2 repeat 360 [fd 3 rt 1]
3 end
4
5 to kreis1
6 repeat 360 [fd 1 rt 1]
7 end
8
9 to must3
10 repeat 36 [KREIS3 rt 10]
11 end
12
13 to must1
14 repeat 18 [KREIS1 rt 20]
15 end
16
17 to afg45pre
18 setpc 2
19 MUST3 rt 2
20 setpc 3
21 MUST3 rt 2
22 setpc 4
23 MUST3 rt 2
24 setpc 5
25 MUST3 rt 2
26 setpc 6
27 MUST1 rt 2
28 setpc 15
29 MUST1 rt 2
30 setpc 8
31 MUST1 rt 2
32 setpc 9
33 MUST1 rt 2
34 end
35
36 to afg45
37 setpc 13
38 MUST3
39 end
```

Aufgabe 46:

```
1 to kreis
2 repeat 360 [fd 2 rt 1]
3 end
4
5 to afg46
6 lt 180
7 setpc 15
8 kreis
```

```
9 lt 180
10 setpc 13
11 kreis
12 end
```

Aufgabe 47:

```
1 to kreis
2 repeat 360 [fd 1 rt 1]
3 end
4
5 to afg47
6 repeat 4 [setpc 0 fd 200 lt 135 setpc 1 kreis lt 135]
7 end
```

Aufgabe 48:

```
1 to quadrat :gr
2 repeat 4[fd :gr rt 90]
3 end
```

Aufgabe 49:

```
1 to kreise :gr
2 repeat 360[fd :gr rt 1]
3 end
```

Aufgabe 50:

```
1 to fettelinie :gr
2 fd :gr rt 90 fd 1 rt 90 fd :gr lt 180
3 end
```

Aufgabe 51:

```
1 to dreieck :gr
2 repeat 3[fd :gr rt 360/3]
3 end
```

Aufgabe 52:

```
1 to reihe :anz
2 repeat :anz[repeat 4[fd 40 rt 90] rt 90 fd 40 lt 90]
3 end
```

Aufgabe 53:

```
1 to vierfachquad :gr
2 repeat 4[repeat 4[ fd :gr rt 90] rt 90]
3 end
```

A. Appendix

Aufgabe 54:

```
1 to sechseck :gr
2 repeat 6[fd :gr rt 60]
3 end
```

Aufgabe 55:

```
1 to haus :gr
2 repeat 4[fd :gr rt 90] fd :gr rt 90 repeat 3[fd :gr lt 120]
3 end
```

Aufgabe 56:

```
1 to muster :gr1 :gr2
2 rt 90 repeat 360[fd :gr1 rt 1] bk :gr2/2 repeat 3[fd :gr2 lt 120]
3 end
```

Aufgabe 57:

```
1 to recht :breite :hoehe
2 repeat 2[fd :breite rt 90 fd :hoehe rt 90]
3 end
```

Aufgabe 58:

```
1 to parallelogramm :gr1 :gr2
2 repeat 2[rt 45 fd :gr1 rt 45 fd :gr2 rt 90]
3 end
```

Aufgabe 60:

```
1 to fettelinie :gr
2 fd :gr rt 90 fd 1 rt 90 fd :gr lt 180
3 end
```

Aufgabe 61:

```
1 to aufgabe61 :anz
2 repeat :anz [kreise 1 rt 360/:anz]
3 end
```

Aufgabe 62:

```
1 to aufgabe62 :n
2 repeat :n [kreise 2 rt 360/:n]
3 end
```

Aufgabe 64:

```
1 to blumen :gr1 :gr2
2 #Aufgabe 64
3     setpc 3 blume :gr1
4     setpc 4 blume :gr2
5 end
```

Aufgabe 65:

```
1 to schoeneblume :n :gr
2 #Aufgabe 65: Weiterentwicklung des Programms blume
3     repeat :n [kreise :gr rt 360/:n]
4 end
5
6 to schoeneblumen :n :gr
7 #Aufgabe 65: Weiterentwicklung des Programms blumen
8     setpc 3 schoeneblume :n :gr1
9     setpc 4 schoeneblume :n :gr2
10 end
```

Aufgabe 66:

```
1 to aufgabe66
2     repeat 36 [blatt 100 rt 20]
3 end
```

Aufgabe 69:

```
1 QUAD100 wait 4 pe QUAD100 ppt
```

Aufgabe 70:

```
1 to QUADLAUF
2 repeat 50 [QUAD100 wait 4 pe QUAD100 fd 4 ppt]
3 end
```

Aufgabe 71:

```
1 to quadlauf
2 #Aufgabe 71
3     repeat 120 [quad100 rt 90 fd 4 lt 90]
4 end
```

Aufgabe 72:

```
1 to quadlauf2
2 #Aufgabe 72
3     repeat 120 [quad100 fd 4]
4 end
```

Aufgabe 73:

A. Appendix

```
1 to aufgabe73
2   repeat 15 [fd 100 bk 100 rt 20]
3 end
```

Aufgabe 74:

```
1 #Aufgabe 74: Es wird kein Programm verlangt.
```

Aufgabe 76:

```
1 to aufgabe76
2   quad100 wait 4 pe quad100 ppt
3 end
```

Aufgabe 77:

```
1 to quadlauf4
2 #Aufgabe 77
3   repeat 120 [quad50 wait 4 pe quad50 fd 4 ppt]
4 end
5
6 to quadlauf3
7 #Aufgabe 77
8   repeat 120 [quad100 wait 4 pe quad100 rt 90 fd 4 lt 90 ppt]
9 end
```

Aufgabe 78:

```
1 to quadlauf5
2 #Aufgabe 78
3   repeat 120 [quad100 wait 2 pe quad100 rt 90 fd 4 lt 90 ppt]
4 end
```

Aufgabe 79:

```
1 to quadlauf6
2 #Aufgabe 79
3   repeat 120 [quad100 wait 8 pe quad100 rt 90 fd 4 lt 90 ppt]
4 end
```

Aufgabe 80:

```
1 to quadlauf7
2 #Aufgabe 80
3   repeat 120 [quad100 wait 2 pe quad100 lt 90 fd 4 rt 90 ppt]
4 end
```

Aufgabe 81:

```
1 to quadlauf81
2 #Aufgabe 81
3   ht
4   repeat 50 [quad100 wait 5 pe quad100 fd 3 rt 90 fd 3 lt 90 ppt]
5   quad100
6   st
7 end
```

Aufgabe 82:

```
1 to aufgabe82
2 #Aufgabe 82
3   ht
4   repeat 360 [quad100 wait 4 pe quad100 fd 5 rt 1 ppt]
5   quad100
6   st
7 end
```

Aufgabe 83:

```
1 to aufgabe83
2 #Aufgabe 83
3   ht
4   repeat 360 [quad100 wait 1 pe quad100 fd 5 rt 1 ppt]
5   quad100
6   st
7 end
```

Aufgabe 84:

```
1 to aufgabe84
2   repeat 6 [aufgabe83]
3 end
```

Aufgabe 85:

```
1 to erde
2 #Aufgabe85 (Erde ist kleiner)
3   repeat 45 [fd 1 rt 8]
4 end
```

A.3. Questionnaire for User Study

Fragebogen

1. Im folgenden Bild sind vier Elemente (a,b,c und d) markiert. Wozu werden diese Elemente in Xlogo verwendet?

Weise die folgenden Begriffe zu:

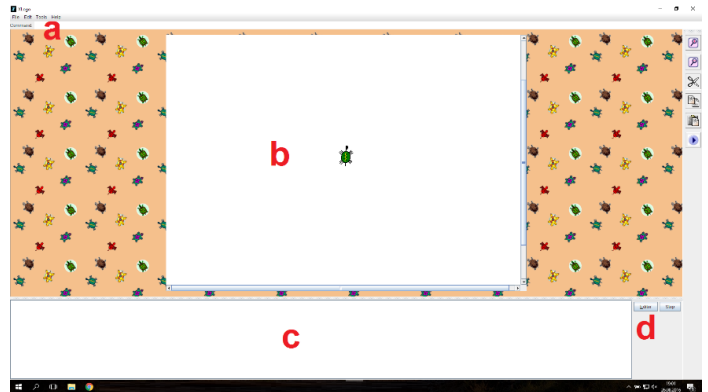
- *Programme definieren,*
- *Zeichenfläche,*
- *vorherige Befehle,*
- *Eingabezeile für Befehle*

a)

b)

c)

d)



2. Im folgenden Bild sind vier Elemente (a,b,c und d) markiert. Wozu werden diese Elemente in Xlogo online verwendet?

Weise die folgenden Begriffe zu:

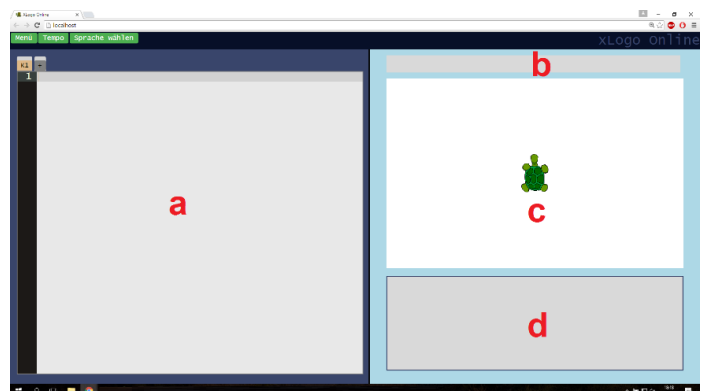
- *Programme definieren,*
- *Zeichenfläche,*
- *vorherige Befehle,*
- *Eingabezeile für Befehle*

a)

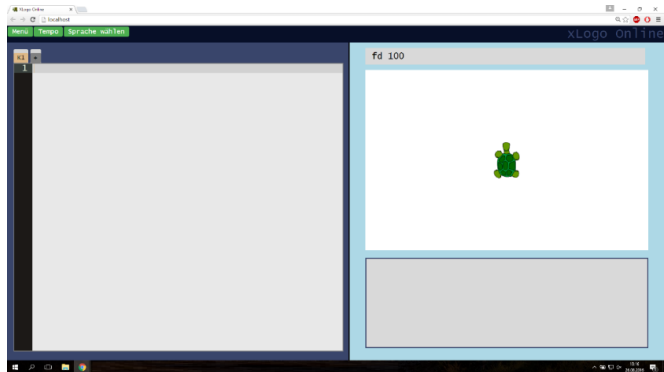
b)

c)

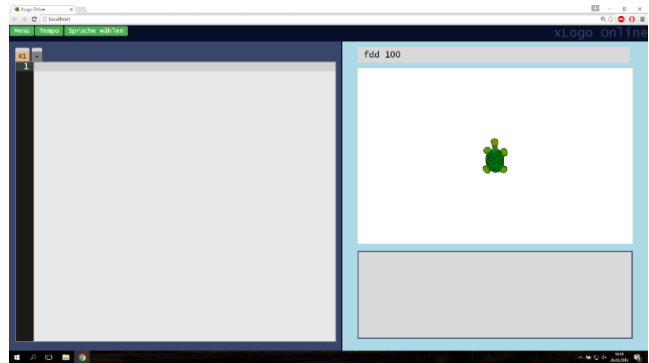
d)



3. Was passiert, wenn du den Befehl *fd 100* ausführst? Wie sieht die Situation danach aus? Zeichne die Veränderung direkt ins untenstehende Bild.



4. Was passiert, wenn du den Befehl *fdd 100* ausführst? Wie sieht die Situation danach aus? Zeichne die Veränderung direkt ins untenstehende Bild.



5. Was würdest du an Xlogo online ändern bezüglich...

	Dein Kommentar	Bewertung: +/-
Aussehen des Editors		
Verhalten des Editors		
Aussehen des Canvas		
Verhalten des Canvas		
Aussehen des Felds für die vorherigen Befehle		
Verhalten des Felds für die vorherigen Befehle		
Aussehen der Eingabezeile		
Verhalten der Eingabezeile		
Allgemein		

6. Allgemeiner Eindruck von Xlogo online:

Sehr schlecht	schecht	gut	Sehr gut

A.4. Supplementary Exercises for Collaborative Coding

Namen schreiben

In den letzten Kapiteln haben wir viel darüber gelernt wie man mit Logo einfache Muster zeichnen kann. Nun möchten wir dieses Wissen verwenden um beliebige Wörter zu schreiben. Konkret geht es darum in Logo unseren Namen zu schreiben. Alle werden ihren eigenen Namen schreiben, daher werden wir am Schluss Bilder wie beispielsweise das folgende sehen:

LORENZ

Bevor wir uns daran machen ganze Wörter zu schreiben, wollen wir uns jedoch zuerst mit den kleineren Komponenten eines Wortes beschäftigen - den Buchstaben. Wie du weißt, ist jedes Wort aus Buchstaben aufgebaut. Wenn wir für sämtliche Buchstaben des lateinischen Alphabets in Logo ein Programm hätten, wäre es kein Problem damit beliebige Wörter zu schreiben.

ABC ... XYZ

Wir versuchen also zunächst einzelne Buchstaben zu zeichnen. Aber wie sollen diese Buchstaben denn aussehen? Wenn wir uns in der Welt umschauen, erkennen wir sofort, dass es sehr viele verschiedene Arten gibt, wie man einen Buchstaben schreiben kann. Betrachten wir nur einmal die folgenden drei Beispiele:

ABC ... XYZ

ABC ... XYZ

ABC ... XYZ

-
1. Vergleiche die obigen drei Schriftarten und markiere fünf Eigenschaften, in welchen sich die Schriftarten voneinander unterscheiden.
-

Wir erkennen, dass die Schriftarten für denselben Text manchmal mehr oder weniger Platz benötigen. Wie viel Platz er genau einnimmt hängt von der Breite der Buchstaben,



und der Grösse der Lücke ab, die wir zwischen zwei benachbarten Buchstaben finden:



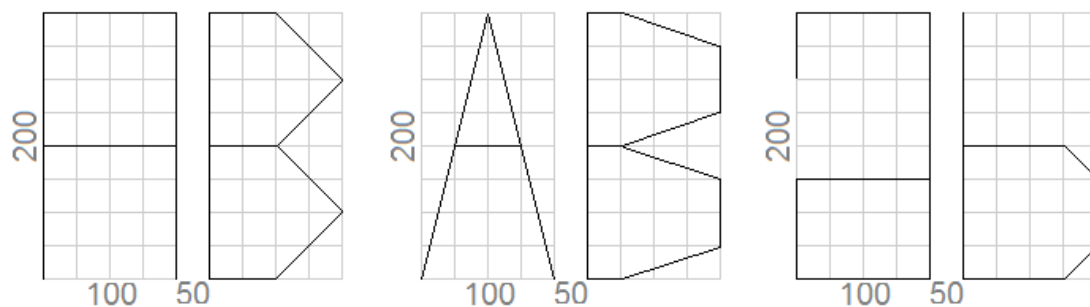
Wir fordern von der Schriftart, die wir in diesem Kapitel entwickeln, dass alle Buchstaben gleich viel Platz einnehmen, dass die Lücke zwischen zwei Buchstaben immer dieselbe Breite hat und dass sämtliche Vorkommen eines Buchstabens identisch aussehen sollen. Wir setzen diese Forderung um, indem wir die folgenden Eigenschaften definieren, an welche sich unsere Schriftart halten muss:

- Jeder Buchstabe darf maximal 100 breit sein
- Jeder Buchstabe darf maximal 200 hoch sein
- Die Lücke zwischen zwei Buchstaben ist 50 breit

2. Fülle die folgende Tabelle aus, in der du berechnen sollst, wie lange ein Wort mit einer gewissen Anzahl von Buchstaben wird:


Anzahl Buchstaben	Beispiel	Wortlänge
1	„A“	$100 + 50 = 150$
2	„W O“	
3	„I C H“	
4	„V O L L“	

Bei der Gestaltung der Buchstaben sind wir grundsätzlich frei, solange wir uns an diese Regeln halten. Die Buchstaben A und B könnten also eine beliebige der folgenden Formen annehmen:



Nun ist es an der Zeit dein eigenes Design der Buchstaben zu entwerfen...

3. *Entwirf für jeden Buchstaben dein eigenes Design und zeichne es jeweils ins Feld rechts neben den Buchstaben:*

A		B		C		D		E	
F		G		H		I		J	
K		L		M		N		O	
P		Q		R		S		T	
U		V		W		X		Y	
Z									

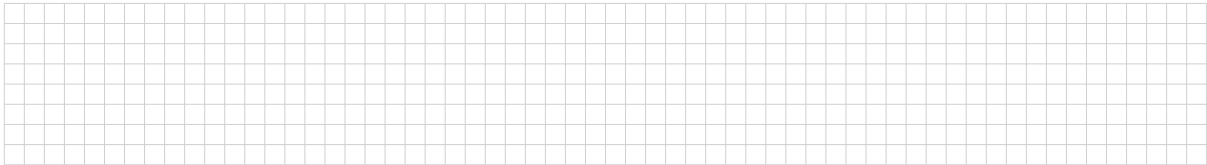
Da wir nun wissen, wie wir die einzelnen Buchstaben des Alphabets zeichnen wollen, ist es endlich möglich aus diesen Buchstaben ein Wort zusammen zu setzen. Achte darauf, dass du die Buchstaben so schreibst, wie sie in deiner selbst entworfenen Schriftart geschrieben werden.

4. *Schreibe die Worte HAWAII, XLOGO und SCHABERNACK in deiner selbst-entwickelten Schriftart.*

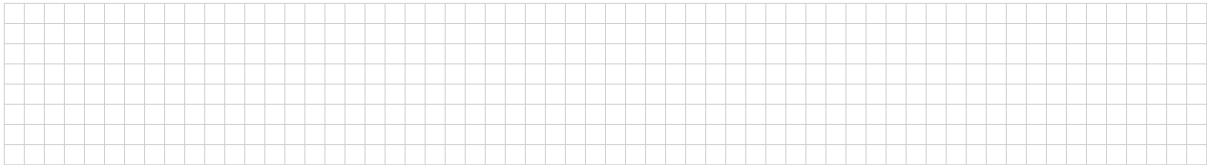
a) HAWAII:



b) XLOGO



c) SCHABERNACK

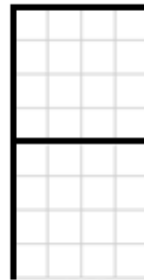


5. *Schreibe deinen Namen in deiner selbst-gestalteten Schriftart.*



Nun geht es darum die von uns entworfenen Buchstaben auch in Logo zu programmieren. Wir erstellen für jeden Buchstaben ein neues Programm im Editor, wie zum Beispiel hier:

```
to a
  fd 100 rt 90
  fd 100 bk 100 lt 90
  fd 100 rt 90
  fd 100 rt 90
  fd 200 lt 90
  pu fd 50 pd lt 90
end
```



Wichtig ist, dass wir uns nach vollständigem Zeichnen eines Buchstabens um 50 nach rechts bewegen, sodass wir uns anschliessend exakt an derjenigen Position befinden, an welcher wir den nächsten Buchstaben zeichnen können.

6. Welcher buchstabe wird hier gezeichnet?

```
to unbekannt  
  repeat 2[fd 200 rt 90 fd 100 rt 90]  
  rt 90 pu fd 150 pd lt 90  
end
```



6. Schreibe je ein Programm für die ersten zwei Buchstaben deines Vornamens.



Dein Name, wie auch der deiner Klassenkameraden ist aufgebaut aus maximal 26 verschiedenen Buchstaben. Da viele deiner Klassenkameraden dieselben Buchstaben programmieren wie du, werden wir die Arbeit ab hier aufteilen, sodass jeder am Schluss ein vollständiges Alphabet hat aber nicht sämtliche 26 Buchstaben programmieren muss.

-Anleitung...

A. Appendix

Bibliography

- A. ROBINS, J. ROUNTREE, N. R., 2003. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13:137–172.
- ABZ, 2016. Logo. abz reference: <http://www.abz.inf.ethz.ch/primarschulen-stufe-sek-1/unterrichtsmaterialien/>. website, April.
- AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. 2006. *Compilers. Principles, Techniques, and Tools*. Pearson, Addison Wesley.
- ANDERSON, J. R., 2000. Learning and memory, an integrated approach.
- ARTO VIHAVAINEN, M. L., AND KURHILA, J., 2012. Multi-faceted support for mooc in programming, October.
- ARUNKUMAR PALANISAMY, M. S. P. F. A. P. P. A. P., 2014. Modelica based, parser generator with good error handling. *Proceedings of the 10th International Modelica Conference S. 567–575*.
- ATOM, 2016. <https://atom.io/>. website, April.
- BAU, D. A., 2015. Droplet, a blocksbased editor for text code. Thesis.
- BELL, T., 2016. Cs unplugged: <http://csunplugged.org/>. website, April.
- BEN-ARI, M., 2001. Constructivism in computer science education. *Jl. of Computers in Mathematics and Science Teaching*.
- BONK, C. J., 2009. *The world is open: How web technology is revolutionizing education*. Jossey-Bass.
- CLOUD9, 2016. <https://c9.io/>. website, April.
- CODIAD, 2016. <http://codiad.com/>. website, April.
- CONOLE, G., 2015. Moocs as disruptive technologies: strategies for enhancing the learner experience and quality of moocs. Thesis.
- DONALD A, N., 1983. Design rules based on analyses of human error. *Communications of the ACM*, April.
- ET AL., H. M., 2014. Functional programming for all! scaling a mooc for students and professionals alike. thesis, May.
- ET AL., A. C. R., 2015. Maps and the geospatial revolution: teaching a massive open online course (mooc) in geography.
- ET AL., C. D. Z., 2015. Direct instruction of metacognition benefits adolescent science learning, transfer, and motivation: An in vivo study.
- FLANAGAN, D., 2014. *Javascript: The definitive guide: Activate your web pages*. O’Reilli.
- GABI REINMANN, H. M., 2006. *Unterrichten und lernumgebungen gestalten*.
- GAVIN BIERMAN, MARTIN ABADI, M. T., 2014. *Understanding typescript*. Springer-Verlag

BIBLIOGRAPHY

Berlin Heidelberg.

- GOKHALE, A. A. 1995. Collaborative learning enhances critical thinking.
- GROSS, T., 2012. Course notes: Compiler design. course, September.
- GROSSECK, G., HOLOTESCU, C., BRAN, R., AND IVANOVA, M., 2015. A checklist for a mooc activist. Conference on eLearning and software for Education, April.
- HAEFLIGER, L., 2014. xlogo mobile. Bachelor Thesis, August.
- HALIMAHTUN KHALID, MARTIN HELANDER, C.-Y. M., 2015. Mooc on human factors in user interface design.
- HROMKOVIC, J. 2010. *Einfuehrung in die Programmierung mit LOGO*. Viewweg+Teubner.
- HROMKOVIC, J., 2013. Homo informaticus. Magazine, October.
- ICECODER, 2016. <https://icecoder.net/>. website, April.
- INFORMATIK, D., 2010. Informatik und allgemeine bildung. course.
- INGMAR DASSEVILLE, G. J., 2015. A web-based ide for idp. Paper, November.
- J. BENNEDSEN, M. E. C., 2008. Reflections on the teaching of programming. chapter Exposing the Programming Process, pages 6–16, Springer-Verlag.
- J. KURHILA, A. V., 2011. Management, structures and tools to scale up personal advising in large programming courses. In Proceedings of the SIGITE '11. ACM.
- JOHN, M., MITCHEL, R., NATALIE, R., BRIAN, S., AND E, E., 2010. The scratch programming language and environment. Trans. Comput. Educ. 10, 4, Article 16, November.
- LE COQ, L., 2009. Xlogo: Reference manual: <http://downloads.tuxfamily.org/xlogo/downloads-en/manual-html-en/>. Website, July.
- LE COQ, L., 2014. Xlogo: <http://xlogo.tuxfamily.org/de/index-de.html>. website, February.
- LEFRANCOIS, G., 2008. Psychologie des lernens. Springer.
- MANDLER, G., 2007. A history of modern experimental psychology: From james and wundt to cognitive science. MIT Press 2007.
- MARCUS HASSELHORN, A. G. 2006. *Paedagogische Psychologie*. Verlag W. Kohlhammer.
- MATTER, B., 2011. Kluger spass: Programmieren in der primarschule. Magazine.
- MEYER, H. 2004. *Was ist guter Unterricht?* Cornelsen.
- MICHAEL FELTEN, E. S. 2012. *Lernwirksam unterrichten*. Cornelsen.
- PAPERT, S. Introduction: What is logo? and who needs it?
- PARR, T. 2013. *The Definitive ANTLR 4 Reference*. The Pragmatic Booksheir.
- PÉREZ, F., AND GRANGER, B. E. 2007. Ipython: a system for interactive scientific computing. *Computing in Science & Engineering* 9, 3, 21–29.
- PERKINS, D. 1999. The many faces of constructivism. *Educational leadership* 57, 3, 6–11.

- PIAGET, J. 1952. *When thinking begins, the origins of intelligence in children*. New York: International University press.
- PIETERSE, V., 2013. Automated assessment of programming assignments. Paper, April.
- PIETRA, P., 2015. An xlogo programming environment to steer a wheeled robot. Bachelor Thesis, April.
- PÖPPER, C., AND ALTENHOFF, A. What drives young women to study computer science in switzerland?—experiences on promoting computer science studies for female high school graduates. *women* 6, 8, 10.
- RAIMOND REICHERT, J. N., AND HARTMANN, W., 2001. Programming in schools — why, and how? *Enseigner l’informatique*, pp 143-152.
- REUSSER, K., 2005. Problemorientiertes lernen — tiefenstruktur, gestaltungsformen, wirkung, February.
- S. TOBIN-HOCHSTADT, M. F., 2008. The design and implementation of typed scheme. Proceedings of POPL.
- SAVKIN, V., 2016. Website: <http://victorsavkin.com/post/118372404541/the-core-concepts-of-angular-2>. Web, April.
- SCHULMEISTER, R., 2001. Taxonomy of multimedia component interactivity a contribution to the current metadata debate. Paper, November.
- SEEL, N. M. 2012. *Encyclopedia of the Sciences of Learning*. Springer US, Boston, MA, ch. Ebbinghaus, Hermann (1850–1909), 1069–1070.
- SERAFINI, G., 2014. Course notes: Fachdidaktik informatik 1+2. course, September.
- STERN, E., 2014. Course notes: Menschliches lernen. course, September.
- STIFTUNG, H., 2008. Das image der informatik in der schweiz. Umfrage, April.
- TERENCE PARR, S. H., 2016. Antlr4. reference manual: <http://www.antlr.org/index.html>. website, April.
- TYPESCRIPT, 2016. Typescript. reference manual: <http://www.typescriptlang.org/>. website, April.
- VANIEA, K., BAUER, L., CRANOR, L. F., AND REITER, M. K., 2012. Out of sight, out of mind: Effects of displaying access-control information near the item it controls. Tenth Annual International Conference on Privacy, Security and Trust.
- VOSNIADOU, S. 2009. *International Handbook of Research on Conceptual Change*. Routledge.
- W3SCHOOLS, 2016. www.w3schools.com. Website, April.
- ZIVKOVIC, M., 2013. Xlogo4schools. Bachelor Thesis, September.



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):

xLogo online - a web-based programming IDE for Logo

Authored by (in block letters):

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

Name(s):

Staub

First name(s):

Jacqueline

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

Zürich, 29.09.2016

Signature(s)

J. Staub

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