Programming in K–6: Understanding Errors and Supporting Autonomous Learning

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
JACQUELINE STAUB

MSc ETH CS, ETH Zürich
born on 22.01.1990
citizen of Menzingen

accepted on the recommendation of
Prof. Dr. Valentina Dagienė (Vilnius University), co-examiner
Prof. Dr. Juraj Hromkovič (ETH Zürich), examiner
Dr. Dennis Komm (ETH Zürich), co-examiner
Prof. Dr. Jan Vahrenhold (University of Münster), co-examiner

2021
Acknowledgment

I would like to express my special thanks to the following individuals and institutions, such as ETH Zürich and Pädagogische Hochschule Graubünden PHGR, without whose support this dissertation would not have been possible.

My sincere appreciation and great thanks go to Prof. Dr. Juraj Hromkovič, my PhD supervisor, who has greatly shaped me and my way of thinking for more than a third of my life. I am very thankful for his continuous support and I will always admire his way of teaching, his tremendous knowledge, and his kind personality. I was only a young Bachelor student when I heard about his project and asked to join one of his first Logo school projects in Domat-Ems (a small village the canton of Graubünden). Without knowing it, this project also entailed my first encounter with a dear work colleague from PHGR: Pascal Lütscher, the teacher who welcomed me into his class, turned out to be a much-valued friend later on. Besides Pascal, I want to thank Dr. Gian-Paolo Curcio, Dr. Leci Flepp, Dr. Bernhard Matter, as well as Dr. Dennis Komm for giving me the unique opportunity to participate in the introduction of computer science to the teacher education at PHGR.

Throughout my PhD, I was surrounded by numerous wonderful personalities. To mention all of them by name, unfortunately, there are too many. Denise Spicher inspired me with her ability to keep track of the student body, even in the face of drastically increasing student numbers, and moreover to establish a personal connection with each student in the department. Angélica Herrera Loyo impressed me with her knowledge, commitment, and motivation to contribute to our group. Many thanks go to Serena Pedrocchi for our many valuable conversations, and to Urs Hauser and Giovanni Serafini who I admire for their profound talent to relate with students. You all have touched me in both my professional and personal life and I want to thank you all for your trust and friendship!

Over the past four years, I have had the pleasure of guiding more than 10 students through their Bachelor’s or Master’s theses to graduation. Working with these young people has been extremely rewarding for me and looking back, I consider the interaction with these students as the highlight of
my entire work. Their motivation, their thirst for knowledge, and their incomparable perseverance with which they joined me in pursuing our common goal can hardly be acknowledged by a simple “thank you”. I’ll try anyway and say: Nils, Martina, Renato, Michelle, Dominic, Anna Laura, Josua, David, Barbara, Larissa, Pascal, and Shi Me – you are my heroes!

Many thanks go to Dr. Dennis Komm as well as Dr. Zaheer Chothia for their invaluable support in the linguistic and scientific fine-tuning of the ideas contained in this thesis. I consider both of them as my mentors and I have the greatest respect for them. Thank you for taking time to assist and guide me through all the stages of my PhD and especially during the preparation of this dissertation. I learned a lot from you and any of my current or future research work will likely always contain your fingerprints.

Last but not least, I would like to thank my parents, Heidi and Beat Staub, who unconditionally supported and motivated me at every step of my development so far. They have put up with many limitations just to ensure my well-being and to enable me to live the life I have today. Thank you for all you have done for me!
Abstract

The research presented in this thesis aims at the development of a programming environment for primary schools that is focused on supporting students’ autonomous learning and independent troubleshooting. With the introduction of computer science as a new school subject, programming in primary school is currently experiencing a strong and promising rise in popularity. This creative form of learning can foster problem solving skills but it simultaneously requires a high level of precision. Making errors is inevitable and bugs show up in a variety of forms, from simple typos to highly complex misconceptions. Errors pose a significant problem for classroom implementation since students often struggle with troubleshooting and thus depend on external help.

Previous research has addressed this problem and presented various approaches that ease primary school programming. Suggestions reach from programming languages with specialized grammars to tangible application domains and block-based programming interfaces. While each of these approaches has its justification in easing the way into programming, still, primary school programmers eventually need to face their errors and acquire appropriate troubleshooting skills.

For more than fifty years, educational and professional programming languages have been treated as separate research fields which went through a distinct evolution in terms of error handling. Motivated by the enormous cost of a blunder in industry, numerous tools were developed for professionals to assist them in locating, analyzing, and fixing errors. Sophisticated tools reach from static program analysis all the way to post-mortem debuggers. In contrast, the educational community engaged with primary school programming focused during the last twenty years primarily on preventing errors. Yet, in order to become self-sufficient programmers, novices must be exposed to their own errors at some point. We developed a Logo learning environment with which primary school programmers aged 5 to 12 years can be introduced to programming and which provides novices with age-appropriate error recovery tools, e.g., an advanced syntax checker, static program analysis, type checking and inference, and a reverse debugger. All of these tools have been adapted to the use case in primary schools.
We evaluated our approach using a set of more than 2 million structurally-erroneous Logo programs that were collected in over 80,000 user sessions. Using the tools presented in this thesis, 97% of all structural errors can be detected proactively (i.e., at compile time) and visualized using in-line error markers. In this way, the majority of structural programming errors can be fixed on-the-fly. The remaining 3% are runtime errors that can be analyzed and examined alongside all logical errors using our debugger.

The proposed research has promising implications for primary school programming. Through the direct feedback loop with thousands of users, we gained an insight into their learning including problems and misconceptions. The analysis of structural errors allowed us to target the needs of students more effectively by adapting our teaching materials. In the long run, this improves the quality of programming lessons in primary school and students learn to handle errors on their own.
Zusammenfassung


In ihrer mehr als fünfzigjährigen Geschichte als separate Forschungsgegenstände haben sich Programmiersprachen aus dem frühschulischen Bildungsbereich immer mehr von professionellen Sprachen entfernt, besonders hinsichtlich der Handhabung von Fehlern. Motiviert durch die enormen Kosten eines Fehlers in der Industrie wurden in jahrzehntelangen Forschungsbemühungen zahlreiche Hilfsmittel entwickelt (z.B. statische Programmanalyse oder vielseitige Debugger), die professionelle Programmiererinnen und Programmrer darin unterstützen, Fehler einfacher zu lokalisieren, zu analysieren und zu beheben. Im Primarschulkontext besteht der Trend hingegen seit mehr als 20

Der verwendete Ansatz wurde durch eine Sammlung von mehr als 2 Millionen strukturell fehlerhafter Logo-Programmen evaluiert, welche in mehr als 80 000 User-Sessions anfielen. Mit unserem Ansatz können rund 97% aller strukturellen Programmierfehler bereits zur Compilezeit detektiert und mit Fehlermarkierungen im Programmtext visualisiert werden. Auf diese Weise gelingt es, einen Großteil aller strukturellen Fehler schnell und einfach on-the-fly zu beheben. Die restlichen 3% aller strukturellen Fehler zeigen sich zur Laufzeit und können gemeinsam mit logischen Fehlern dank bidirektionalem Debugger untersucht werden.

Contents

1 Introduction

1.1 Motivation .................................................. 1
1.2 Goals and Contributions ................................... 3
1.3 Theses ......................................................... 5
1.4 Structure ...................................................... 6
1.5 Related Publications ......................................... 6
  1.5.1 Conference Papers and Journals ......................... 7
  1.5.2 Bachelor and Master Theses .............................. 8

2 Background

2.1 A Programming Language Specifically for Novices .................. 11
  2.1.1 Syntax .................................................. 13
  2.1.2 Semantics .............................................. 21
2.2 Turtle Graphics – Logo’s Most Famous Application Domain ........... 32
  2.2.1 Turtle Graphics and Coordinates .......................... 32
  2.2.2 What is the Turtle? ...................................... 37
  2.2.3 Tangible Programming – Even Preschoolers can Interact with the Turtle 38
2.3 Programming Environments for Logo ................................ 40
  2.3.1 Back to the Roots – XLogoOnline’s Ancestors ................ 40
2.4 Practical Experiences with Programming Novices ...................... 52
2.5 Logo as a Philosophy of Education ................................ 53
  2.5.1 Learning by Doing – Constructivism ...................... 54
  2.5.2 Learning by Making – Constructionism .................... 55
3 How to Integrate Logo Into the Swiss School System

3.1 Starting Point Switzerland .................................................. 58

3.2 A Continuous Plan from Kindergarten to Grade 6 .................. 60
  3.2.1 Stage 1: Basic Commands and Command Sequences ........ 60
  3.2.2 Stage 2: Strategies and Testing ................................. 62
  3.2.3 Stage 3: Looping Single Instructions ......................... 64
  3.2.4 Stage 4: Parameters in Basic Commands ..................... 66
  3.2.5 Stage 5: Looping Over Multiple Instructions ............... 67
  3.2.6 Stage 6: Modularity and Parametrization .................. 69
  3.2.7 Stage 7: Conditional Statements .............................. 71

3.3 Teaching Programming Across Different Subjects ............... 73
  3.3.1 Logo in Mathematics Classes ................................. 73
  3.3.2 Logo in Language Classes .................................. 78

3.4 Student-Centered Learning and Teaching in Logo ............... 82

3.5 Age-Appropriate Input Methods ....................................... 83

4 Program Translation and Handling Structural Errors ........... 89

4.1 A Brief Overview – From Raw Text to Parse Tree ................ 89
  4.1.1 Lexer: Token Formation and Clustering .................... 90
  4.1.2 Parser: Retrieving Structural Properties .................. 92

4.2 Two Options to Define a Logo Grammar ......................... 93
  4.2.1 Keywords and Identifiers - Why Bother the Distinction? ...... 94
  4.2.2 Approach 1: Built-ins as Keywords ........................ 95
  4.2.3 Approach 2: Built-ins as Identifiers ....................... 96

4.3 Common Structural Errors in Novices’ Logo Programs .......... 98
  4.3.1 The Majority of Structural Errors .......................... 98

4.4 Detecting Structural Errors – Syntax Checking ................. 100
  4.4.1 A Methodology to Detect as Many Structural Errors as Possible .... 100
  4.4.2 Linguistic Ambiguities and Their Implications on Grammar Design ... 104

4.5 Detecting Structural Errors – Semantic Verification .......... 107

4.6 The Limits of Automated Error Detection in Logo ............ 113
5 Program Execution and Runtime Systems for Logo

5.1 Parse Tree Traversal

5.1.1 Many Roads Lead to Rome

5.1.2 Basic Parse Tree Traversal

5.1.3 Visiting Procedure Calls

5.1.4 Visiting Control Flow Structures

5.2 Physical and Virtual Environments

6 Debugging

6.1 Introduction

6.1.1 Making Mistakes: A Matter of Attitude

6.1.2 The Rift Between Expectation and Outcome

6.1.3 Students Do Not Know How to Cope with Logical Errors

6.1.4 Structural Errors vs. Logical Errors

6.2 A Glimpse Back in History

6.3 What Natural Coping Strategies do Novices Use?

6.3.1 Setup

6.3.2 Problem Decomposition: There is More than One Correct Solution

6.3.3 Not Only Beginners Face Logical Problems

6.3.4 Challenges in Tracing

6.4 Implementing a Reverse Logo Debugger

6.4.1 Logo's Language Constructs

6.4.2 Single Stepping: Understanding Program Execution

6.4.3 Reverse Debugging: Going Back in Time

6.5 Debugging is Relevant for Everyone

7 Evaluation

7.1 Demographics

7.1.1 Infrastructure and Programming Setup at Schools

7.1.2 Coverage in Switzerland
Chapter 1

Introduction

In recent years, more and more countries have decided to introduce computer science (and with it programming) into their public school curricula. Even primary school students now have the opportunity to learn how to program. This opens up many great opportunities, but it also raises unresolved questions of the following form: What should programming lessons in kindergarten look like when children are not yet able to read and write? How can a learning environment meet the needs of different age groups? Or even: How can a teacher be able to support students individually, when programming is known to be a buggy business and we can expect every child in the class to be constantly confronted with errors? In this thesis, we want to clarify these questions by presenting a new teaching concept for programming classes in K–6 in combination with a dedicated programming environment.

1.1 Motivation

With the introduction of computer science into public schools, even children now have the opportunity to explore the exciting world of algorithms. In order to teach fundamental concepts of informatics in an age-appropriate way, however, both teaching materials and learning environments must be made available to the population at large. In Switzerland alone, there is a market for more than five hundred thousand students aged five to twelve years who are enrolled in the Swiss K–6 primary school system [77]. All these children are supposed to develop the skills to solve problems algorithmically by the time they enter secondary school. Different methods were presented describing how algorithms can be explored by students either with or without the use of computers [31, 50].
Programming is a creative form of learning that allows teachers and students to explore and develop algorithms for a wide range of different problem cases. At the core, programming means to communicate with a computer in a language that the computer “understands.” Programming languages, much like natural languages, have their own vocabulary and their own grammar. In contrast to communication between humans, however, a computer is not able to interpret ambiguous statements and thus, programmers need to pay special attention to express their thoughts precisely (in respect of both syntax and semantics). Making errors is inevitable and most programmers consequently spend considerable amounts of time searching, analyzing, and fixing bugs [71, 82, 151].

The spectrum of typical programming errors reaches from simple structural errors like incorrect punctuation, missing or erroneous arguments, and unbalanced parentheses to more complex logical errors such as mixed-up loop conditions. While these two problem classes are the daily bread of almost all programmers independent of age and experience, there are also some problems that are specific to young programmers. Children as young as 5 years old are able to understand basic programming concepts and plan ahead [131], but they struggle to express themselves in a written language. Classically, programs are represented by text, but this requires that the programmers are fluent in both reading and writing. This is usually not the case for such a young age group and causes considerable difficulties.

Learning environments can, with the right tools, either eliminate some of these problems or at least simplify them to such an extent that novices are able to work autonomously and learn to cope with errors. This is a necessity for programming lessons in the context of public schools since most primary school teachers are inexperienced programmers who are largely outnumbered by their students. Unless children are able to cope with errors and resolve them autonomously, teachers are at risk of being overwhelmed by too many students requesting their help simultaneously. We consider autonomous troubleshooting to be an essential part of the skill set a young programmer needs to acquire by the end of primary school. For this reason, it is important that programming environments for novices are equipped with dedicated tools for error handling.

In this dissertation we present both a concept and a corresponding learning environment that enable programming lessons from kindergarten up to the transition to secondary school. The environment allows students to start programming without any reading or writing skills, it offers useful tools for locating, analyzing, and fixing both structural and logical programming errors, while also addressing the constructivist and constructionist ideas of Jean Piaget [153] and Seymour Papert [84].
1.2 Goals and Contributions

The overall goal of this thesis consists of the conception, development and analysis of a system that can be used in programming activities in kindergarten and primary school. Students are a special audience for such a system and the analysis of common errors and misconceptions allows us to study the learning processes involved in programming. A number of research questions have been studied from two different domains: heterogeneity and error handling.

1. **Heterogeneity**:  
   What needs must be considered if the same programming environment is to be used by school children aged five to twelve years (i.e., kindergarten through grade six in the Swiss education system)?

2. **Error Handling**:  
   What common structural and logical programming errors and misconceptions do Logo novices experience?  
   What fraction of Logo programming errors can be detected proactively, (i.e., at compile time) while programmers are still typing?  
   Under what conditions and how is it possible to provide a reverse debugger for Logo?

With the aim of addressing the above-mentioned open questions, this dissertation makes contributions in the following points:

1. We created a comprehensive teaching concept for programming classes from kindergarten to sixth grade including a learning environment that can be used by all children in this age range. Learners can immerse themselves in the programming language Logo with concepts such as parameters, repetition, and modularity. Three different starting points are available for different age groups: (i) The first environment is designed for children aged five to eight years. It has a block-based interface and its commands come with pre-defined parameters that cannot be changed by the programmer. Using a symbolic interface, even children who cannot read and write yet are able to communicate with a computer. (ii) The second environment is used by children aged nine to ten years. It has a block-based interface but provides the possibility to parameterize all the available basic commands. (iii) The third version is a text-
based environment for Logo, which also allows users to define new custom commands, that may be parameterized.

2. A second contribution lies in the design and development of a static program analysis mechanism for Logo. This mechanism shifts the point of error detection for a large majority of all errors from runtime to compile time without introducing addition syntax. Our approach is based on the extension of the execution pipeline with an additional program verification step that allows several error classes (i.e., unresolved identifiers, missing or extraneous arguments, duplicate program or parameter declarations, invalid program identifiers, and incorrectly typed arguments for built-in commands) to be located and highlighted at compile time. Programmers are provided with in-line error markers that indicate problems while they are still typing.

3. Another contribution is our in-depth examination of structural errors committed by Logo beginners. A large number of errors can occur as programmers learn to write text-based programs. For almost a year, we collected all structural errors that were committed using our environment and we created a methodology to classify these errors. We found the vast majority of all structural errors to fall into the category of unresolved program identifiers as well as missing or extraneous arguments. This result confirms the findings of previous research which attributed few classes of errors to be responsible for the majority of all error occurrences [51, 160].

4. Not all errors are of structural nature. In fact, there is a whole class of errors that cannot be easily identified, namely logical errors. A logical error is characterized by the fact that it complies with the underlying grammar and can be parsed and executed correctly. However, the error causes the program to have unexpected effects. Usually, a debugger is used to locate such errors. A debugger allows a program to be executed step by step and pause after each command. Along the way, programmers can inspect the current state of the program. There are several Logo environments with debuggers of this type [168, 32], but none of them allows turtle graphics programs to be traversed in reverse order (i.e., stepping backwards in time). We developed such a stepping mechanism and explain how seemingly irreversible commands can be reverted.

Within this work, the author has developed the overall concept of how to achieve a spiral approach in programming lessons from kindergarten to sixth grade. The implementation of individual software
components was achieved via 13 Bachelor and Master theses (see Section 1.5.2) which built on top of a framework that was developed by the author as part of her Master’s thesis [173]. All student projects were directed and individually supervised by the author. The scientific studies presented in this thesis were planned, carried out, and evaluated by the author herself.

1.3 Theses

For the future of programming in school, we argue for the following three points:

1. For almost two decades, our community has been using block-based learning environments to give beginners a smooth start into programming. There are multiple reasons why block-based environments are used: some (like ourselves) see a potential to reach young children who would otherwise struggle with the handling of a keyboard. Others, meanwhile, consider structural programming errors a threat for all novices, independent of their age. Consequentially, opinions also diverge on the question when to transition away from block- into text-based programming. Some people suggest that blocks should be used well into tertiary education [88]. Conversely, we argue that there is no need to stick with block-based environments for so long. We show an approach of how students can successfully transition away from blocks and handle their own errors once they know how to read and write.

2. Most text-based learning environments for novices have a reactive approach to handle errors. That is, they report most errors only once a programmer decides to execute his or her code. Then, a long and frustrating process begins, until finally a result is visible on the screen: for each structural error, they try to execute their code, receive a red flag which then they need to fix before starting all over again. We argue that a proactive approach to error handling can help students to skip over this tedious phase more quickly. Our approach allows structural errors to be located and reported at compile time, as the programmer is still typing. We argue that a majority of all structural errors can be detected and reported before execution.

3. While simple imperative programs are just executed from top to bottom, control flow elements may cause a program to follow arbitrary long and complex patterns at runtime – some commands may be executed repeatedly and others not at all. Logical errors cause unexpected results which force programmers to mentally retrace program flow. This activity is hard for both novices and
1.4 Structure

This dissertation starts with a broad overview of the research field around the programming language Logo, its application domain turtle graphics, and experience on the topic of troubleshooting with novices. After this background chapter, the next chapter presents our didactic concept which is followed by three chapters pinpointing the process of execution at three different stages: (i) at compile time, (ii) at runtime, and (iii) while debugging. Finally, a last chapter follows in which we present an empirical study that outlines which kinds of errors occur how often in practice. The overall structure is displayed in Figure 1.1.

1.5 Related Publications

Some of the work presented in this thesis was previously published or evolved as part of Bachelor or Master theses at ETH Zürich. Here is the complete list of all publications related to this thesis.
1.5.1 Conference Papers and Journals

The following list shows several articles related to our programming environment or the underlying philosophy. In all except for the sixth entry, the writer of this dissertation held the role of the corresponding author.


1.5.2 Bachelor and Master Theses

In addition, many of the ideas and prototypes put forward as part of this thesis have been proposed as part of Bachelor or Master theses at ETH Zürich. The following students owe recognition for their efforts and great accomplishments towards the programming environment XLogoOnline.

Master Theses:

- David Eschbach (2019):
  A Computer-Based Examination System for XLogoOnline

Bachelor Theses (ordered chronologically):

- Marko Živković (2013):
  XLogo4Schools

- Nils Leuzinger (2017):
  Extension of the browser-based XLogoOnline IDE with a module for cooperative programming learning

- Martina Forster (2017):
  How to support children’s autonomous recovery from compilation errors in Logo

- Renato Menta (2018):
  Supporting Novice Programmers in Recovering from Semantic Errors

- Michelle Barnett (2018):
  Implementing block-based programming for XLogoOnline

- Dominic Weibel (2018):
  Well done! Keep it up! XLogoOnline Turtle is ready to judge your code

- Anna Laura John (2018):
  XLogoOnline Turtle emerges into real life

- Josua Cantieni (2019):
  Building a refined parser for XLogoOnline
1.5 Related Publications

- Barbara Isler (2019):
  From an On-Screen Rendering to the Whiteboard – Connecting XLogoOnline with a Physical Device

- Larissa Schrempp (2019):
  Implementing a tool to generate navigation tasks from kindergarten to second grade with direct integration into XLogoOnline

- Pascal Wacker (2021):
  Implementing a programming competition system for XLogoOnline

- Shi Me Henry (2021):
  Implementing a tool to generate turtle graphics tasks for third and fourth grades with direct integration into XLogoOnline
Chapter 2

Background

For several decades, educators and scientists have been asking themselves the questions “what makes good programming education?” There is still no definite and conclusive answer to this question, which indicates that we are facing a difficult query likely depending on numerous factors. At least the following four facets can be distilled as distinguished influences on the experience a novice programmer has in class [170]: (i) the programming language they use, (ii) its application domain, (iii) the programming environment they work with, and (iv) psychological aspects related to learning. All of these factors, summarized in Figure 2.1, contribute to the overall programming experience and need to be kept in mind when evaluating any given context.

In the course of this chapter, we will shed a light on each of these four areas and briefly discuss the most relevant insights researchers gathered in the past. Moreover, we provide a historical background of past work that we are building on top of.

2.1 A Programming Language Specifically for Novices

Before 1967, high-level programming languages like FORTRAN, ALGOL, COBOL, and LISP were widely spread among the professional community. These languages were popular choices among experts and allowed for various interesting applications once programmers got used to the syntactic and semantic details of the respective language. Many of these languages were, however, known to have an error-prone syntax that was difficult to get used to. Lisp, for example, was both popular for its expressiveness and notorious for its exuberant use of parentheses, which posed a risk for compile-time
errors due to mismatched parentheses. In particular novice programmers were prone to committing numerous errors which signified a rather tough start into the world of programming.

Researchers at Bolt, Beranek, and Newman (including well-known names such as Seymour Papert and Wally Feuerzeig) came to the conclusion that the programming languages used at the time were poorly suited for the needs of beginners and that a solution to this problem could only lie in the development of a new programming language. Through their special beginner-oriented programming language Logo, it should even be possible to introduce children to the world of programming, they claimed. This was a revolutionary assertion considering the fact that computers of the time were nothing like what they are today: a computer was a very large and expensive machine, and hardly any individual could afford one. The idea that computers could be an integral part of a classroom was by no means common.

Wally Feuerzeig mentions [65] three criteria Papert and him considered important during the design process of their new programming language: (i) First, they wanted a language that is simple enough for third-graders to use with very little preparation. (ii) Second, they wanted a language with a structure that embodies mathematically important concepts with minimal interference from
programming conventions. (iii) And third, the programming language should permit the expression of mathematically rich nonnumerical as well as numerical algorithms.

In the following two sections, we will present relevant information about the programming language Logo and in particular its syntactic and semantic attributes such as its type system and programming paradigms.

2.1.1 Syntax

Logo is a language with a long history and whose grammar has never been formally specified. As a consequence, many different dialects have emerged over time; all with slightly different vocabulary and syntax. In this work, we use a Logo dialect called XLogo. We deliberately limit our presentation to those language aspects that are connected to turtle graphics. (Be aware that Logo and turtle graphics are not the same thing.) Unless stated differently, all code examples in this thesis will be provided in the XLogo dialect.

**Syntactical Language Elements**

The process from raw, meaningless sequences of characters to the interpreted program code that can be executed by the computer is long and sophisticated. It passes through several stages, the first of which is concerned with converting a sequence of characters into a sequence of tokens (thereby attaching meaning to single characters or sequences of characters). This process is called *lexical analysis* and it is accomplished by defining a grammar. Using the rules defined by a regular expression, a given sequence of characters can be identified as an instance of a specific token class.

XLogo recognizes six types of tokens: (i) keywords, (ii) constants, (iii) identifiers, (iv) strings, (v) numbers, and (vi) operators. These six classes of tokens are identified as follows:

**Keywords**

Each programming language has a certain number of reserved words, which cannot be overwritten by the programmer. Those words are called *keywords*. Certain keywords have become so widely accepted that almost all programming languages have some form of these keywords integrated into their syntax (e.g., `if`, `while`, `true`). Other keywords, in contrast, are specific to certain languages and appear rarely in others. Logo contains representatives in both classes.
Reserved keywords like if, while, and print are not only used in Logo but also in many other programming languages. The same is true for the keywords to and end which are used to indicate the beginning and the end of a user-defined program. Although these exact keywords are specific to Logo, most languages use some built-in keyword for the same purpose (e.g., Python uses the keyword def to indicate the beginning of a procedure declaration instead of to. Besides using a different keyword, they are semantically identical).

There are some keywords that are specific to turtle graphics, such as built-in movement and rotation commands. Most of these come in pairs with one full (i.e., spelled out) version and one two-letter abbreviation as in clearscreem and cs. Both the full version and the abbreviation are reserved words and cannot be overridden. In fact, whether or not these built-in commands are included in the syntax or not is open to debate. In Chapter 4 we will present two approaches, one which does treat built-in commands as syntactic elements, whereas the other treats them as semantic elements just like any other user-defined command.

Some more examples of Logo keywords involve pen manipulation commands (e.g., penup, pendown, penerase, penpaint, and setpencolor), colors (e.g., red, green, and blue) and mathematical keywords (e.g., sin, sqrt, and mod).

**Constants/Literals**

Any data representation that corresponds to a fixed value that cannot be changed by the programmer is called a constant or literal. Logo has four types of these elements: (i) numerical literals, (ii) string literals, (iii) Boolean literals, and (iv) color literals.

Numerical constants are either represented by a sequence of digits 0 through 9 or as a mathematical symbol. A sequence of simple digits is interpreted as the corresponding integer value. Integers that contain exactly one punctuation mark (.) somewhere between the digits, are interpreted as floating point numbers (e.g., 12.625). Unlike other languages, Logo does not allow floating point numbers to start or end with a punctuation mark (e.g., .935 is not a legitimate floating point number in Logo). In addition to integers and floating point numbers, there is also a small number of irrational numbers, which are made available to the programmer via a textual acronym: \( \pi \) (standing for \( 3.1415 \ldots \)) and \( e \) (standing for \( 2.7182 \ldots \)). These two constants represent two of the most commonly used irrational numbers at machine precision, although they are relatively rarely used in primary school programming for obvious reasons.
Table 2.1 Logo uses three different syntactic forms to represent colors

<table>
<thead>
<tr>
<th>Color Name</th>
<th>Color Index</th>
<th>RGB Triplet</th>
</tr>
</thead>
<tbody>
<tr>
<td>black</td>
<td>0</td>
<td>[0 0 0]</td>
</tr>
<tr>
<td>red</td>
<td>1</td>
<td>[255 0 0]</td>
</tr>
<tr>
<td>green</td>
<td>2</td>
<td>[0 255 0]</td>
</tr>
<tr>
<td>yellow</td>
<td>3</td>
<td>[255 255 0]</td>
</tr>
<tr>
<td>blue</td>
<td>4</td>
<td>[0 0 255]</td>
</tr>
<tr>
<td>magenta</td>
<td>5</td>
<td>[255 0 255]</td>
</tr>
<tr>
<td>cyan</td>
<td>6</td>
<td>[0 255 255]</td>
</tr>
<tr>
<td>white</td>
<td>7</td>
<td>[255 255 255]</td>
</tr>
<tr>
<td>darkgray</td>
<td>8</td>
<td>[128 128 128]</td>
</tr>
<tr>
<td>lightgray</td>
<td>9</td>
<td>[192 192 192]</td>
</tr>
<tr>
<td>darkred</td>
<td>10</td>
<td>[128 0 0]</td>
</tr>
<tr>
<td>darkgreen</td>
<td>11</td>
<td>[0 128 0]</td>
</tr>
<tr>
<td>darkblue</td>
<td>12</td>
<td>[0 0 128]</td>
</tr>
<tr>
<td>orange</td>
<td>13</td>
<td>[255 200 0]</td>
</tr>
<tr>
<td>pink</td>
<td>14</td>
<td>[255 175 175]</td>
</tr>
<tr>
<td>purple</td>
<td>15</td>
<td>[128 0 255]</td>
</tr>
<tr>
<td>brown</td>
<td>16</td>
<td>[153 102 0]</td>
</tr>
</tbody>
</table>

Strings are represented by sequences of letters, numbers, and white spaces, surrounded by square brackets [...]. The content between the square brackets can consist of any characters except for closing square brackets. All of the following examples are legitimate strings in Logo [hello world], [fd 100], and [12]. The string [a ] b], however, is not legitimate.

There are exactly two Boolean literals in Logo. Like in many other programming languages, a Boolean literal can only take one of two values namely `true` or `false`.

The last category of literals in Logo is used to represent colors and was not available from the beginning – Logo had already been in use for ten years when Apple II introduced the first commercially available color screen. In order to represent colors, Logo nowadays uses three different forms: (i) `color name`, which is a set of keywords like red, blue, or yellow, (ii) `color index`, which uses the numbers 0 to 16 to refer to the same color palette as the color names, and (iii) `rgb values`, which are represented by three space-separated numbers between 0 and 255 that are surrounded by square brackets. In Table 2.1 we show the 17 colors that can be represented in all three ways. Note that there are numerous more colors which can, however, only be represented by RGB triplets.
Logo’s syntax is context-sensitive. For example, an RGB color literal cannot be distinguished from a string literal in a purely syntactical way. This means that a compiler can only answer the question whether \([0\ 0\ 0]\) is a color or a string when it is aware of the context in which the token is used. For instance, in \(\texttt{setpc \ [0\ 0\ 0]}\), the token is used as a color whereas \(\texttt{print \ [0\ 0\ 0]}\) uses the same token either as a string or as a color.

**Identifiers**

Identifiers are found in two use cases: (i) program names and (ii) variable and parameter names.

In XLogo, *program name identifiers* consist of a letter followed by any number of upper- or lowercase characters, underscores and digits such as \(a\), \(A123\), \(fd_\_100\), and \(BANANA\). But not the following: \(5eck\), \(my\ name\) and \(fd-100\). The first instance does not start with a character, the second and third contain characters that are not allowed in identifiers (spaces and dashes).

XLogo *variable name identifiers* are almost identical to program name identifiers: they consist of a letter followed by any number of upper- and lowercase characters, underscores, and digits, too. They do, however, require the prefix \(:\) to indicate that the identifier will be used as a variable. The only exception to this rule is when a variable is declared (or re-declared) using the \(\texttt{make}\) command. In this case, the identifier is used with the prefix \(\"\) to indicate that the variable is used as a reference to write to. What at first glance may look like an arbitrary syntactical rule does indeed have a historical background. The first Logo version was written in Lisp [81], which resulted in some of the terminology and syntax to be adopted from Lisp. The idea of distinguishing between the lookup of a name and the value of a variable was originally coined by Lisp [170].

A last point to note in the context of identifiers is that programming languages are often oriented towards English speakers. Special characters like German umlauts (ä, ö, ü) and French accents (é, è, â, ï, etc.) are common in the international community. Professional programming environments, however, often do not allow them as part of their identifiers. When used with novices, this could cause problems since some beginners consider these special characters to be a natural part of their language and they therefore expect to be able to use them anywhere. For this reason, we have designed our syntax rules such that special characters are allowed in identifiers. There are several other Logo versions in our genealogy which also support special characters in identifiers.
2.1 A Programming Language Specifically for Novices

Listing 2.1 The two programs have a fundamentally different meaning although they differ superficially only in one space. This, however, causes the minus operator once to be considered as a unary operator (on the left), and once as a binary operator (on the right). The consequence is that once, \texttt{setxy} is not provided with enough arguments while the other time it does receive the expected number of arguments.

**Operators**

Operators are core components of a programming language. In Logo there are three categories of classical operators:

1. arithmetic operators (e.g., \texttt{+} as is \texttt{2+3})
2. relational operators (e.g., \texttt{<} as in \texttt{2<3})
3. logical operators (e.g., \texttt{&&} as in \texttt{true&&false})

These three categories represent the general scope of what can be done with an operator. An operator can be used to express mathematical operations, to compare elements with one another, and to evaluate logical statements. For this, in addition to the operator itself, we also need elements on which the operator can work. These elements are called *operands* and can take on one of the following three forms:

1. the operator works on constants (e.g., \texttt{2} as in \texttt{2+3})
2. the operator works on variables (e.g., \texttt{:x} as in \texttt{:x+:y})
3. the operator works on composite expressions (e.g., \texttt{2*:x + 3*:y})

Most of all operators in Logo are *binary operators*, i.e., they operate on exactly two operands. Several examples of this form have been mentioned above. Logo also offers some *unary operators*, i.e., operators which do not work on two operands but only on one. An example is the operator \texttt{!}, representing a logical *not*. Note that some operators can be interpreted as both unary and binary, as for instance the \texttt{+} and \texttt{-} operators. Tables 2.2 to 2.4 give an overview over all the classical operators used in Logo.
<table>
<thead>
<tr>
<th>Op</th>
<th>Kind</th>
<th>Effect</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>binary</td>
<td>Return the sum of both operands</td>
<td>1+2</td>
</tr>
<tr>
<td>+</td>
<td>unary</td>
<td>Return the operand as is</td>
<td>+1</td>
</tr>
<tr>
<td>-</td>
<td>binary</td>
<td>Return the difference between the first and the second operand</td>
<td>2-1</td>
</tr>
<tr>
<td>-</td>
<td>unary</td>
<td>Return the operand with a negated sign</td>
<td>-1</td>
</tr>
<tr>
<td>*</td>
<td>binary</td>
<td>Return the product of the first and the second operand</td>
<td>2*3</td>
</tr>
<tr>
<td>/</td>
<td>binary</td>
<td>Return the quotient of the first operand divided by the second</td>
<td>2/3</td>
</tr>
</tbody>
</table>

Table 2.2 Logo’s arithmetic operators.

<table>
<thead>
<tr>
<th>Op</th>
<th>Kind</th>
<th>Effect</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>binary</td>
<td>Return a Boolean value stating whether the first operand is smaller than the second</td>
<td>1&lt;2</td>
</tr>
<tr>
<td>&lt;=</td>
<td>binary</td>
<td>Return a Boolean value stating whether the first operand is smaller than or equal to the second</td>
<td>1&lt;=&gt;2</td>
</tr>
<tr>
<td>=</td>
<td>binary</td>
<td>Return a Boolean value stating whether the first and the second operand are equal</td>
<td>1=2</td>
</tr>
<tr>
<td>&gt;=</td>
<td>binary</td>
<td>Return a Boolean value stating whether the first operand is greater than or equal to the second</td>
<td>1&gt;=2</td>
</tr>
<tr>
<td>&gt;</td>
<td>binary</td>
<td>Return a Boolean value stating whether the first operand is greater than the second</td>
<td>1&gt;2</td>
</tr>
</tbody>
</table>

Table 2.3 Logo’s relational operators

On the basis of the syntactic properties of the Logo programming language, general principles can be derived. One of them is that spaces carry meaning and that programmers must be careful when using them. The underlying rationale is as follows: Logo uses relatively few syntactical elements that determine how a program is interpreted. The whitespace character is used to separate both consecutive arguments and statements which means that a single whitespace character can change the meaning of an entire program. Listing 2.1 shows two Logo programs side by side, which differ only just in the use of a whitespace. Yet, the two programs are interpreted differently by the computer and hence have a different meaning. Note that \texttt{setxy} is a command that requires two arguments and the minus character can be used as either a unary or a binary operator. In Listing 2.1a, the minus is interpreted as a unary operator, which means that the command \texttt{setxy} can be executed without errors. In Listing 2.1b however, the minus is interpreted as a binary operator which leaves \texttt{setxy} with too few arguments. This example illustrates how vulnerable Logo’s syntax can be in certain cases. It is a problem that does not happen outside the realm of languages that use whitespace as a separator between arguments.
2.1 A Programming Language Specifically for Novices

<table>
<thead>
<tr>
<th>Op</th>
<th>Kind</th>
<th>Effect</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>binary</td>
<td>Return a Boolean value stating whether both operands are true</td>
<td>true &amp;&amp; true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>binary</td>
</tr>
<tr>
<td>!</td>
<td>unary</td>
<td>Return a Boolean value stating whether the operand is false</td>
<td>! true</td>
</tr>
</tbody>
</table>

Table 2.4 Logo’s logical operators.

Assignment versus Equality:

Most imperative programming languages use a fourth class of operators to indicate variable assignment. The symbol chosen for assignment is = in many famous languages like Python, Java, or C. This symbol has striking similarities to the operator used for equality checks == which is why the two operators = and == are often confused by novices [55].

Unlike Python, Java, or C, Logo does not use the equality symbol for variable assignment (e.g., x = 1). Instead, Logo uses the dedicated command make. This command has no visual resemblance to the equality symbol and leaves the = symbol unused and available as a relational operator to perform equality checks.

The Syntax of Procedure Invocation

Procedure invocation is a concept that nowadays is offered across virtually all programming languages. The associated syntax, however, differs from one language to another. In some cases, a large number of parentheses is required to uniquely bind a procedure and its arguments together. In other cases, such as Logo, parentheses can be omitted almost completely. In this section, we show the range of parenthesis rules related to procedure calls across different languages, using three representatives, namely Lisp, Python, and Logo.

At the one end of the spectrum, we find Lisp (see Listing 2.2) which makes extensive use of parentheses to such an extent that, in fact, its fully-parenthesized notation reveals how the program is internally parsed. Arguments are separated by whitespace while all program invocations along with their arguments have to be encapsulated by opening and closing parentheses. The structure of the parentheses suffices to unambiguously convert a Lisp program into the corresponding abstract syntax tree.

In Python, every program invocation requires one set of opening and closing parentheses around its arguments. Unlike Lisp, Python uses commas to separate consecutive arguments (see Listing 2.3). In
addition, parentheses are used to indicate precedence rules that would otherwise not be pre-determined by the language (e.g., the expression \((2+3)*4\) requires parentheses in Python since otherwise Python’s default precedence rules would apply and execute multiplication before addition). In Lisp, the same problem does not occur since the order of precedence is implicitly given through the chosen syntax.

Logo, finally, is a language that uses parentheses only sparingly. Indeed, Logo programmers need parentheses exclusively to indicate precedence rules. Procedure invocations do not require parentheses, and instead, the program name is just followed by its arguments directly, separated by whitespace (see Listing 2.4). While Lisp, Python, and many other languages need parentheses to unambiguously identify the binding between arguments and their respective procedure, this is not the case in Logo. Let us have a look at the example shown in Listing 2.4 and examine the implications of this choice: Without a precise specification of \(\text{foo}\) and \(\text{bar}\), it is not possible to tell whether the number 3 is intended as an argument for \(\text{foo}\) or for \(\text{bar}\). This question can only be answered when the context is fully known, including the specification of how many arguments the two procedures take respectively. Through the grammar or the user-defined specification of a Logo command, it is clear how to parse and execute the program. Logo proves that parsing is possible without a strict requirement for parentheses except for priority rules in arithmetic expressions.

**Line Endings and Statement Separators**

Logo is a lightweight language in terms of its syntax – not only considering program invocation but also when it comes to the use of statement separators and line endings. Different languages use different symbols to separate consecutive statements. In general, we observe two common techniques used in modern programming languages: (i) semicolons and (ii) newlines.

Languages that use semicolons as statement separators (e.g., Java, Pascal or C) allow multiple statements to be written in one line. The semicolon is used by the compiler to infer where the one statement ends and where the next statement starts. In the past, many educators have voiced their concern that semicolons are syntactic clutter which should be avoided for the purpose of novices’
productivity [89]. With the susceptibility of novice programmers to make errors of all kinds, a long argument has started on whether to choose a programming language with or without semicolons [155].

A second class of programming languages replaces semicolons with newline characters as the typical separator between consecutive statements. While this leads to less syntactical clutter, it also means that programs which could previously be written with just a few lines of code now take considerably more vertical space on the screen. Some languages that are compromised by this rule allow both newline characters and an alternative character (typically semicolon, colon, or comma) to still write several statements in one line (e.g., Python, FORTRAN, or Scala), leaving the trade-off between syntactical clutter and wasted space to the programmer.

Logo is one of very few languages that neither suffers from syntactical clutter through semicolons nor excessive vertical space consumption. It uses whitespace as separator between statements which allows for an arbitrary number of commands to be written in one single line without any separating character other than whitespace. This choice, however, has direct implications: Without knowing the program signature (i.e., name, number of arguments, and whether or not a program provides a return value), it is not possible to tell where the one statements stops and where the next statement starts. In the example shown in Listing 2.4, for instance, it is unclear whether \texttt{bar} is used as an argument of \texttt{foo} or whether the two procedures are independent of one another. By providing the signature of all built-in and user-defined programs, it is possible to identify consecutive statements despite just using whitespace separators.

### 2.1.2 Semantics

In order to understand the implications and trade-offs of different languages, it is essential to also consider the semantics of a given language. We cover several topics (namely type system, programming paradigms, function semantics, and scoping) that show the most important semantic properties characterizing Logo. Many of these properties have a historical background and sometimes show relating properties between Logo and Lisp.

#### Understanding Logo’s Type System

Logo is a language that has a number of built-in types that can be used in different contexts. There are four primitive data types (i.e., numbers, Booleans, strings, and colors) which can be used as arguments for primitive commands. Together, they form combinations which may be correct or incorrectly typed.
Figure 2.2 Logo has four classes of primitive types: numericals, booleans, strings, and colors.

1. **Primitive Elements**

All programming languages are composed of a number of so-called *primitives*. These semantic atoms represent the smallest units of meaning in a given language and they cannot be decomposed further. Programmers assemble primitives to form new units of meaning which then are added to the language and facilitate the design of more complex structures. We discuss two kinds of primitives, namely *primitive data types* and *primitive commands*.

**Primitive data types** are one of the default components of a typed language. They usually involve numerical data (like *integers* and *floats*), Boolean data (to represent truth values), and textual data (like *strings* and *characters*). Logo’s turtle graphics additionally has a dedicated data type to represent colors. As explained in Section 2.1.1, there are three syntactic forms in which a color can be represented. Figure 2.2 shows an overview over all of Logo’s primitive types.

Some languages choose a lenient approach to typing where the type of a given value can be changed dynamically to fit the context. One such example concerns the treatment of Booleans. Lisp and JavaScript, for instance, allow programmers to use any data and interpret it as a Boolean. They distinguish between a special symbol to represent *false* (e.g., *Nil* in Lisp and 0 in JavaScript) while everything else is implicitly interpreted as *true*. Under these conditions, the dynamic type system accepts various elements as *true* (e.g., the number 12 or the string *hello*).
2.1 A Programming Language Specifically for Novices

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ht</td>
<td>Hide the turtle icon on the screen</td>
<td>0</td>
</tr>
<tr>
<td>st</td>
<td>Show the turtle icon on the screen</td>
<td>0</td>
</tr>
<tr>
<td>home</td>
<td>Reset to turtle to position 0 0</td>
<td></td>
</tr>
<tr>
<td>fd</td>
<td>Move the turtle forward a certain distance</td>
<td>1 (numerical)</td>
</tr>
<tr>
<td>bk</td>
<td>Move the turtle back a certain distance</td>
<td>1 (numerical)</td>
</tr>
<tr>
<td>rt</td>
<td>Rotate the turtle a certain angle to the right</td>
<td>1 (numerical)</td>
</tr>
<tr>
<td>lt</td>
<td>Rotate the turtle a certain angle to the left</td>
<td>1 (numerical)</td>
</tr>
<tr>
<td>setheading</td>
<td>Change the turtle’s global orientation</td>
<td>1 (numerical)</td>
</tr>
<tr>
<td>setx</td>
<td>Set the x-position of the turtle on the screen</td>
<td>1 (numerical)</td>
</tr>
<tr>
<td>sety</td>
<td>Set the y-position of the turtle on the screen</td>
<td>1 (numerical)</td>
</tr>
<tr>
<td>setxy</td>
<td>Set the x- and y-position of the turtle on the screen</td>
<td>2 (numerical)</td>
</tr>
</tbody>
</table>

Table 2.5 Eleven of the most common turtle graphics commands related to the turtle specifically.

Logo does not support such conversions to Booleans, but a similar approach can be used to handle colors. Some values can be interpreted in different ways. For instance, the numbers 1–16 can both be interpreted as integers and as colors. Similarly, a subset of all strings (namely those which correspond to an RGB triplet as in \([12 \ 100 \ 213]\)) can be interpreted both as a string and as a color, depending on context.

As a dynamically-typed language, Logo typically performs type tests at runtime [180]. This means that type errors are usually only detected during execution and result in runtime errors. This is different in languages with static type checking (e.g., Java and C) where type errors are typically detected at compile time.

**Primitive commands** come as a built-in part of the language as well. Primitive commands are used to perform standard operations on data. Typical examples are arithmetic operations like \(+\), \(-\), \(*\), and \(/\) for numerical data, logical operations like \(\text{and}\), \(\text{or}\), and \(\text{not}\) for Boolean data, and operations such as \text{print} and string concatenation on textual data. One special category of primitive commands belongs to the application domain of *turtle graphics*. There are a number of primitive commands (see Tables 2.5 to 2.8) that are used for navigating the turtle, manipulating the pen, and interacting with canvas and history. All of these primitives are tailored to accept specific data type arguments and reject all inputs whose data type does not fit.

2. **Combination**

Primitive commands can be combined with primitive data to create a *combination*. Logo’s built-
### Command Meaning Arguments

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>pu</td>
<td>Lift the pen</td>
<td>0</td>
</tr>
<tr>
<td>pd</td>
<td>Lower the pen</td>
<td>0</td>
</tr>
<tr>
<td>pe</td>
<td>Use an eraser when moving around</td>
<td>0</td>
</tr>
<tr>
<td>ppt</td>
<td>Use a pen when moving around</td>
<td>0</td>
</tr>
<tr>
<td>setpc</td>
<td>Exchange the pen color with a given value</td>
<td>1 (color)</td>
</tr>
<tr>
<td>setpw</td>
<td>Change the pen width to a given size</td>
<td>1 (numerical)</td>
</tr>
</tbody>
</table>

Table 2.6 Five of the most common turtle graphics commands related to the pen.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs</td>
<td>Reset the screen content</td>
<td>0</td>
</tr>
<tr>
<td>wash</td>
<td>Remove all previous traces</td>
<td>0</td>
</tr>
<tr>
<td>setsc</td>
<td>Exchange the background color with given color</td>
<td>1 (color)</td>
</tr>
</tbody>
</table>

Table 2.7 Three of the most common turtle graphics commands related to the canvas.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ct</td>
<td>Remove all text from the history</td>
<td>0</td>
</tr>
<tr>
<td>pr</td>
<td>Print content into the history</td>
<td>1 (string</td>
</tr>
</tbody>
</table>

Table 2.8 Two of the most common turtle graphics commands related to the history.
2.1 A Programming Language Specifically for Novices

Listing 2.5 Three Logo combinations. Only the combination (c) is correct, while (a) and (b) are flawed due to too extraneous and ill-typed arguments respectively.

in commands may expect a certain number of arguments of a certain type. Only combinations that are correctly assembled can be executed.

A combination is correct if and only if

a) the number of arguments fits the specification of a given command, and
b) all arguments have the type indicated by the specification of the respective command

In Listing 2.5, we show three Logo combinations. Only one of them is correct while the other two are flawed. Listing 2.5a is incorrect due to the number of arguments. According to the specification, the primitive command setpc requires exactly one argument. Listing 2.5b is problematic since the provided argument (i.e., a string) does not have the correct type. According to specification, setpc requires a color. Only the combination in Listing 2.5c is providing both the correct number of arguments and the correct type.

Most of turtle graphic’s primitive commands do not have a return value other than their visual effect on the screen. Arithmetic, trigonometric, relational and logical commands, on the other hand, usually do return a result to the context they were called from. Such functions allow for creating nested call chains. The program fd random 100, for instance, contains two combinations which are nested into one another. The inner function random 100 returns a result that can be used as an argument for the outer combination with fd. Tables 2.9 to 2.12 capture all functions that are expected to return a value to the context they were called from.

Nested combinations can, just like non-nested ones, be tested for correctness. Nested combinations need to fulfill two requirements to be correctly typed, namely (a) the number of arguments of each combination in the nested structure needs to fit the specification and (b) all arguments of each combination in the nested structure need to have the correct type. To check whether a nested combination fulfills the requirements, type information needs to be propagated up through all the layers of nesting.
### Table 2.9 Arithmetic functions in Logo with their respective input and output types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>Return the absolute value of argument x</td>
<td>1 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>sqrt</td>
<td>Return $\sqrt{x}$ where x is a numerical argument</td>
<td>1 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>log</td>
<td>Return $\log_{10}(x)$ where x is a numerical argument</td>
<td>1 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>random</td>
<td>Return a random number between 0 and the argument -1</td>
<td>1 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>+</td>
<td>Return $x + y$ where x and y are numerical arguments</td>
<td>2 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>-</td>
<td>Return $x - y$ where x and y are numerical arguments</td>
<td>2 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>*</td>
<td>Return $x \cdot y$ where x and y are numerical arguments</td>
<td>2 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>/</td>
<td>Return $\frac{x}{y}$ where x and y are numerical arguments</td>
<td>2 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>power</td>
<td>Return $x^y$ where x and y are numerical arguments</td>
<td>2 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>mod</td>
<td>Return $x % y$ where x and y are numerical arguments</td>
<td>2 (numerical)</td>
<td>numerical</td>
</tr>
</tbody>
</table>

### Table 2.10 Trigonometric functions with their respective input and output types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>Return $\sin(x)$ where x is a numerical argument</td>
<td>1 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>cos</td>
<td>Return $\cos(x)$ where x is a numerical argument</td>
<td>1 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>tan</td>
<td>Return $\tan(x)$ where x is a numerical argument</td>
<td>1 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>arcsin</td>
<td>Return $\sin^{-1}(x)$ where x is a numerical argument</td>
<td>1 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>arccos</td>
<td>Return $\cos^{-1}(x)$ where x is a numerical argument</td>
<td>1 (numerical)</td>
<td>numerical</td>
</tr>
<tr>
<td>arctan</td>
<td>Return $\tan^{-1}(x)$ where x is a numerical argument</td>
<td>1 (numerical)</td>
<td>numerical</td>
</tr>
</tbody>
</table>

### Table 2.11 Logical functions with their respective input and output types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>Return $x &amp; &amp; y$ where x and y are Boolean arguments</td>
<td>2 (Boolean)</td>
<td>Boolean</td>
</tr>
<tr>
<td>or</td>
<td>Return $x \mid \mid y$ where x and y are Boolean arguments</td>
<td>2 (Boolean)</td>
<td>Boolean</td>
</tr>
<tr>
<td>not</td>
<td>Return $!x$ where x is a Boolean argument</td>
<td>1 (Boolean)</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

### Table 2.12 Relational operators with their respective input and output types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Return $x &lt; y$ where x and y are numerical arguments</td>
<td>2 (numerical)</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Return $x \leq y$ where x and y are numerical arguments</td>
<td>2 (numerical)</td>
<td>Boolean</td>
</tr>
<tr>
<td>=</td>
<td>Return $x = y$ where x and y are numerical arguments</td>
<td>2 (numerical)</td>
<td>Boolean</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Return $x \geq y$ where x and y are numerical arguments</td>
<td>2 (numerical)</td>
<td>Boolean</td>
</tr>
<tr>
<td>&gt;</td>
<td>Return $x &gt; y$ where x and y are numerical arguments</td>
<td>2 (numerical)</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
2.1 A Programming Language Specifically for Novices

Listing 2.6 Loop and conditional statements cause the creation of a new block scope. In this case, a new block scope is started with the body of the respective statement. The block scope automatically closes once the statement has finished.

**Variable Passing and Scoping**

One important aspect of semantics in a programming language is the question of how name bindings with variables are resolved and thus what model of *scope* the language uses. In this section, we examine the concept of scope with a specific focus on how the scoping rules work in Logo. The discussion will be held on two levels of granularity: First, name binding in block scope. Second, scoping in the context of function scopes.

**Block Scope**

Logo is not the only language that offers the possibility to group one or several statements into one logical block [25]. Each block automatically instantiates a new level of nomenclature (i.e., a context where variables “live” in), which is commonly called a block scope. Both loops and conditional statements work on individual block scopes and are affected by this mechanism. The code snippets in Listing 2.6 show three examples where new block scopes are created in Logo.

At runtime, a new block scope is automatically created when the block (i.e., the body of a statement like *if*, *repeat*, or *while*) is entered. Consequently, once execution leaves a block, the respective block scope is deleted and with it all of its local variables. All identifiers that are instantiated within a block are considered local to the block and have no existence outside. Consequently, every attempt of accessing a local variable from outside of a block scope results in failure. This effect is illustrated in Listing 2.7a, where *x* is a local variable that lives in the block scope of the *if* statement. It does not exist outside of that block and can hence not be accessed – as a consequence, line 6 results in an error. We call variables non-local if they were not created within the scope of a block. These variables do exist even after the block has finished. Listing 2.7b shows an example of such a variable. It is
Listing 2.7 $x$ is a local variable in (a) and a non-local variable in (b). As a consequence, on the left, line 3 succeeds but line 6 fails, whereas on the right, both line 3 and 6 succeed.

Listing 2.8 Two programs with the same effect, once written in Logo (which uses block scope, on the left) and once in Python (which does not use block scope, on the right). Note that Logo requires the variable $x$ to be instantiated before the if-statement which is not necessary in Python. Otherwise, line 5 of the Logo code could fail.

A block scope offers a useful mechanism for recycling unneeded identifiers. Not all variables need to be available throughout program execution. In general, the concept of temporary variables includes loop iterators (which do not exist in Logo) and variables that are used as temporary storage. Without block scope, the variable space in a program can easily pollute and unneeded identifiers can make it hard to keep an overview over all variables used. Not all languages, however, decided to implement block scope. The restrictions imposed on the programmer are not always wanted and so some languages (e.g., Python) decided against block scopes. This means that certain programs that execute fine in Python, have to be re-written in order to still work under the additional restrictions in Logo. Listing 2.8a shows how a piece of Logo code has to be adapted to achieve the same effect as the equivalent Python code in Listing 2.8b.
Block scope is one of the scoping mechanisms used in programming languages to recycle unneeded identifiers, on the level of individual blocks. Next, we will discuss the purpose of scoping on the level of functions.

**Function scope**

Orthogonal to block scopes, most languages (including Logo) also use individual scopes for functions. These scopes are called *function scopes* and they are used to keep the variable space small and tidy. Keeping track of all name bindings across the entire program would be an immense challenge for programmers, especially if programs grow longer and many variables are used. This problem can be resolved thanks to the concept of function scopes.

*Local variables* are variables that are defined within a function. They live in the scope of that particular function and are destroyed when the lifetime of the function has reached its end. The same variable name can be used in different function scopes without any interference. Listing 2.9 shows such a case where the variable name \( x \) is used in two different functions. We highlight that in this case the two instances are not connected to one another in any way other than naming.

One of the concepts which allow variables to be shared among different functions is called the *global scope*. It corresponds to the scope of the overall program execution including all variables that are not defined in any function scope. Variables that are defined in the global scope are not bound to any function. Instead, they are available throughout execution until the program terminates. Listing 2.10 shows how the same program as in Listing 2.9 could be re-written with a global variable that is accessed in both \( \text{foo} \) and \( \text{bar} \). The two examples differ from one another in regard of the chosen scope for the variable \( x \); either \( x \) is part of the global scope (as in Listing 2.10) or there are two variables named \( x \) in two different function scopes (as in Listing 2.9).

If we understand a program as a structure of nested scopes, then all inner functions have access to variables in their outer scope (i.e., the scope of their parent), as in Listing 2.10. In contrast, the parent has no access to any variables in the local scope of their children. This fact is visualized in Figure 2.3, where the variable \( x \) is located either in the inner scope or the outer scope and accessing it either succeeds or fails.

There exist different implementations of the idea of scope. The two dominant models that are used by the vast majority of all programming languages are called *static scoping* and *dynamic scoping*. One of the main differences between the two models is whether static or dynamic program properties are chosen to determine the program’s underlying structure of scope.
Listing 2.9 foo and bar have their own local instance of x. The two instances are completely independent of one another.

Listing 2.10 foo and bar share the same variable x which is declared in the global scope and exists independently of the two functions.

Figure 2.3 A variable x that is located in the local scope of an inner function cannot be accessed from the parent. If, however, the variable is located in the outer scope of a function, it can be accessed from within the function scope.
Listing 2.11 Static and dynamic scoping can lead to different results. Note that bar defines its own local version of x (which is shadowing the variable in the global scope with the same name) and hence line 7 does not touch the global variable but rather uses the local version of x. Function foo, on the other hand, does not have its own local version of x. So, in a statically scoped universe, foo is linked to the global scope and hence line 2 consistently prints the value 0. In a dynamically scoped universe, recursively tracing up the call chain leads once to the global variable (line 12) with value 0 and once to bar’s local variable (line 8) with value 1.

- **Static scoping**
  Static scoping relies on the lexical structure (e.g., the location of function definitions within the code) to resolve variable lookups. Name bindings are determined on a purely statical basis. Since all information is available at compile time, the precise name bindings are known from the beginning and do not change during execution.

- **Dynamic scoping**
  Dynamic scoping purely relies on dynamic properties (e.g., the order of function invocations) to resolve variable lookups. To do so, the call chain is recursively traced upwards until a suitable candidate is found. If no candidate can be found even in the global scope, the variable lookup fails and an error is reported. Due to its dynamic nature, name bindings may change in different contexts – depending on the caller of a function, different values may be retrieved for the same variable. This effect is shown in Listing 2.11, where the variable lookup in line 2 changes depending on whether foo was called from the global scope or from bar’s function scope.

Most languages use static scoping while arguing that it is simpler to understand and allows errors to be detected at compile time already. Logo, however, is one of the few languages that uses dynamic
scoping. This is attributable to historical reasons: Lisp, one of the most famous languages in use at the time Logo was invented, was one of the earliest languages that introduced dynamic scoping. Logo adopted the same scoping strategy as Lisp.

The concept of scope and variables in general have proven to be difficult for novices [70, 115]. For this reason, we decided to simplify the concept by not providing global variables and by defining a simplified function scope which is neither dynamic nor static – instead we provide a flat binding as proposed by Tanter [177]. In a flat finding, each function has its own local scope and can only access variables that are defined in the respective local scope. A flat binding causes the example shown in Listing 2.11 to report an error instead of a response. The reason for this is that foo contains a local variable that was neither defined in the body of foo nor is it passed to foo from outside via a parameter. In a flat binding, only these two possibilities are considered and so the variable lookup in line 2 results in an error.

2.2 Turtle Graphics – Logo’s Most Famous Application Domain

Sometimes, Logo and turtle graphics are used as synonyms. This is an incorrect conclusion [65] since there are many application domains for the Logo language – turtle graphics is just one of them. Yet, turtle graphics undoubtedly is the most popular and famous of all application domains for Logo. The turtle has been used in different circumstances and for different purposes: Sometimes, it has had a physical appearance (as a robot moving in the real world), other times it has had a virtual appearance (as an abstract object on a computer screen). Sometimes, the turtle has been involved in drawing activities, other times it was involved in navigation activities without the need for a pen. All of these use cases suit the same purpose: the turtle was intended to be an object-to-think-with [144]. It is a metaphor that is supposed to help students build strong mental models of how computing concepts work. In the following, we will give an overview of what turtle graphics is, how it came into existence, and what varieties of it exist.

2.2.1 Turtle Graphics and Coordinates

Turtle graphics is the main application domain Logo is nowadays known for. It involves an actor (i.e., the “Turtle”) which understands a predefined set of movement and rotation commands which leave a visible trace on a two-dimensional surface and thus allows programmers to draw. Logo is
not the only language that provides a vocabulary for drawing – the same was already possible in BASIC and is still possible nowadays with languages like SVG or Processing. One difference between these examples and turtle graphics, however, lies in the chosen coordinate system. For drawing purposes, both the Cartesian and the polar coordinate system can be used. Languages like BASIC, Processing, and SVG [185] all use Cartesian coordinates whereas Logo and Turtle graphics use polar coordinates [106]. In this subsection, we explain why polar coordinates are better suited for primary school programming due to the order and timing of the mathematics curriculum in Swiss schools which usually does not include Pythagorean arithmetic before grade 7.

Both Cartesian and polar coordinates can be described either in global or local terms. Global coordinate systems generally assign each point an absolute coordinate that depends only on one common point – the origin. In the global Cartesian coordinate system, for instance, all coordinates are expressed in relation to their horizontal and vertical distance to the origin (i.e., a dedicated point at coordinate \((0,0)\)). The coordinate of any other given point \((x,y)\) is expressed as the horizontal distance \(x\) and the vertical distance \(y\) to this dedicated origin. In Figure 2.4, four points are depicted and labeled by their global Cartesian coordinates. The point indicated in orange has coordinates \((2,1)\) due to its horizontal distance (2 units) and vertical distance (1 unit) from the origin. In Figure 2.4, four points are depicted and labeled by their global Cartesian coordinates. The point indicated in orange has coordinates \((2,1)\) due to its horizontal distance (2 units) and vertical distance (1 unit) from the origin.

Listing 2.12 shows an example of how Processing’s global Cartesian coordinates can be used to draw a square using a command called `line`. This command has the following specification: `line(start_x, start_y, destination_x, destination_y)` where the first and the second argument are used as the starting point of a line and the third and the fourth argument are used as the end point of the line. The first line of code in Listing 2.12 therefore connects the two points \((1,1)\) and \((1,2)\) in a line. All arguments in this program are interpreted as global Cartesian coordinates.

Listing 2.12 Program to draw a square in global Cartesian coordinates

```plaintext
# link (1,1) with (1,2)
line(1,1,1,2)
# link (1,2) with (2,2)
line(1,2,2,2)
# link (2,2) with (2,1)
line(2,2,2,1)
# link (2,1) with (1,1)
line(2,1,1,1)
```
A local coordinate system, on the other hand, usually describes all transitions from one point to another through relative changes rather than absolute ones. This means that programmers do not have to keep track of absolute coordinates and instead focus just on the local transitions from one point to the next. One way of doing so is to use vectors. Vectors are mathematical objects that express transitions between points in space. One advantage of vectors lies in the fact that they are not bound to a specific location and can be applied to any point. This always results in the same transition which does not depend on the starting point. For instance, vector $\vec{a} = (1, 1)$ can be used to transition from point (0,0) to (1,1). Yet, the exact same vector can also be used to express the transition from point (3,3) to (4,4). Figure 2.5 and listings 2.13 and 2.14 show how the same square can be described through the application of four vectors. In order to walk the same path as in Figure 2.4, the four vectors can be applied to any starting point. We chose (1,1) in this example. If the square was supposed to be re-drawn at location (3,3), the same four vectors could be applied to another starting point (namely (3,3)), and the square would be correctly drawn without the need to further adapt the vectors themselves (see Listing 2.14). Doing the same in global coordinates would require all coordinates to be adapted or the concept of variables to be used. This concept, however, has been shown to be hard to understand for novices [113] and is thus usually only introduced at later stages of the curriculum.

Turtle graphics uses a local coordinate system where the turtle’s position represents the origin at all times. Instead of using Cartesian coordinates, however, turtle graphics uses polar coordinates – a system that is more adapted to working with angles. Geometry and angles are a crucial part of turtle graphics: Figure 2.6 shows a typical task that can be solved by primary school students. The
2.2 Turtle Graphics – Logo’s Most Famous Application Domain

Figure 2.6 Drawing equilateral triangles in Cartesian coordinates is hard for students.

Listing 2.15 Students require a basic understanding of Pythagoras in global Cartesian coordinates...

Listing 2.16 ... and local ones, too.

The same problem does not occur when working in the polar coordinate system: polar coordinates are assigned to points in space by determining their distance and angle from a reference point. Similar to the Cartesian coordinate system, there is a dedicated point which is used as a reference to express locations around it. Instead of using the vertical and horizontal distance to said point (like in the Cartesian coordinate system), the polar coordinate system uses the radius and the polar angle. In Figure 2.7, three points are positioned around a reference point in the center of the coordinate system. All three points are labeled in polar coordinates which can be decomposed into two parts: The first component is the radius between the point and the center, the second component is the angle between the point and the reference direction (i.e., the 0 degree mark at the top). The coordinate (1, 30), for instance, describes a point that is located one unit away from the reference point in the center, which has a 30-degree deflection from the reference direction indicated as 0 degrees at the top. The two points highlighted in orange have the same radius from the center. The two points highlighted in yellow share a common deflection of 30 degrees from the reference direction.
Figure 2.7 An example of three points and their coordinates in the polar coordinate system. The point in the middle is the reference point. All three points have a coordinate that is expressed as a pair of the radius and angle to the reference point.

Figure 2.8 An equilateral triangle with four indicators at interesting time points

Listing 2.17 A Logo program that draws an equilateral triangle with four indicators corresponding to Figure 2.8

# T0
fd 2 rt 120
# T1
fd 2 rt 120
# T2
fd 2 rt 120
# T3

In order to draw the equilateral triangle in Figure 2.8 using polar coordinates, a sequence of just six turtle graphics commands suffices (see Listing 2.17). Note that, unlike in Cartesian coordinates, the program does not require Pythagorean calculations. The four labels A, B, C, and D indicate four different points during execution.

Once point $T_3$ is reached, the turtle performed an overall rotation of 360 degrees. All a programmer needs to know in this implementation is the total angle of a full revolution. In order to draw regular polygons, a full revolution of 360 degrees can be split into any number of identical outer angles. This empowers young students to draw triangles (and, in fact, any regular polygon) without having to go into advanced mathematics.
2.2.2 What is the Turtle?

The first time tortoises were mentioned in the context of computing, the term was not tied to education but rather to modelling brain cells inside a moving floor robot. This research was conducted by neurophysiologist, cybernetician, and robotician William Grey Walter, who introduced two cybernetic animals called Elmer and Elsie between 1948 and 1949 [34]. Collocually, the two robots were often just called “tortoises”. Papert and his colleagues honored the work of William Grey Walter by adopting the term “turtle” for their own work in the context of education [146].

Despite using the same name, Papert’s turtle has fewer similarities with Elmer and Elsie than it might seem at first glance. Papert explained the difference as follows: “Grey Walter’s turtle had life-like behavior patterns built into its wiring diagram. Our [turtles] have no behavior except the ability to obey a few simple commands.” [146] The intention was not to create a device with complex behavior and a life of its own. In contrast, the turtle was meant to be a loyal but passive communication partner with drawing and movement capabilities which does not do anything unless it is explicitly told to. In order to make the turtle draw, people have to learn its language (i.e., the programming language Logo). Papert said: “The Turtle […] exists within the cognitive minicultures of the LOGO environment, LOGO being the computer language in which communication with the Turtle takes place. The Turtle serves no other purpose than of being good to program and good to think with.” [144]

In order to implement such a communication partner, both virtual and physical approaches are possible. Papert pointed out: “Some Turtles are abstract objects that live on computer screens. Others […] are physical objects that can be picked up like any mechanical toy.” [146] From the beginning, both screen turtles and physical floor robots were developed and tested with children. Logo’s first floor turtle was created by Tom Callahan at MIT [170]. This robot was connected to a computer via a hardwire line and whenever the robot moved, the cable was dragged behind. From a terminal, programmers could instruct the robot to move and once execution started, the commands were sent through the wire to the wheeled floor robot [64]. In 1972, Paul Wexelblat developed the first wireless version of a floor turtle called Irving that could be communicated with via radio frequency [170]. Still later, robots could be steered via Bluetooth technology [35, 101].

Around 1972, the development of the first virtual screen turtles started at MIT. Ever since, there were many different implementations that rely on the fundamental idea of turtle graphics. Some of these versions rely on drawing functionality [86, 15, 100], others just on navigation [2]. The same is true for floor robots as well: there were numerous different robots that implemented the vocabulary
of turtle graphics [184, 101, 35, 170]. Some of them are designed for drawing purposes [184, 101], others just implement the navigation features, which typically replicate the classical turtle graphics functionality without drawing [35].

Overall, there is a plethora of different physical and virtual implementations of the turtle which all have been invented during the last fifty years. As technology progressed, new and innovative applications were found for the same fundamental idea: what Papert initially intended to be an object to think with, found its use in various different scenarios – sometimes the turtle is a physical object that can be navigated only, other times it is a virtual object that can be used for drawing as well. The core of the approach is the question of what can be achieved using the turtle, by using a language like Logo.

2.2.3 Tangible Programming – Even Preschoolers can Interact with the Turtle

Although the turtle can manifest as both physical or virtual device, its language stays mostly the same. The most common way to communicate with the turtle is by typing commands on a physical keyboard. This, however, proves difficult for young programmers. Papert and colleagues claim that programming can be taught to children of all ages – even preschoolers [144]. In order to teach such a young audience, they experimented with different input techniques. Papert’s student Radia Perlman invented a new way to interact with young children using physical buttons. Her idea paved the way to what we call tangible programming nowadays [138]. In this section, we present Perlman’s approach and we indicate how she intended to bridge preschoolers with older students.

Between 1974 and 1976, Perlman developed a special input method called “TORTIS” (Toddler’s Own Recursive Turtle Interpreter System) that was specifically targeted at the capabilities of preschoolers [149]. TORTIS is a physical device that decomposes into four independent modular parts. It is possible to use one module alone or multiple modules combined, depending on the proficiency level of a child. Each part is a box with a number of buttons on top where every button has a characteristic functionality that is sent to and executed by a physical floor robot. The teaching approach is based on the idea that the modules are introduced separately and in accordance with the child’s mental development. The four modules comprise the following:

1. The first module is a box with just a few simple commands whose functionality is related to Logo’s basic movement and rotation commands. All commands have static behavior, that is: movement commands make the robot travel a fixed distance forward or back, and rotation
commands make the turtle change its direction at a fixed angle, namely 5 degrees. Once a button has been pressed, a signal is sent to the robot, which executes the corresponding functionality immediately. In order to achieve rotations of more than 5 degrees, the user has to press the rotation command repeatedly.

2. The second module is a box with buttons numbered 1 till 10. This box can be combined with the buttons from the previous box in order to execute single commands repeatedly. For instance, the turtle can be made to move forward nine steps by pressing button “9” before pressing button “forward”. Note that this has the same effect as pressing the button *forward* nine times. However, it takes fewer buttons to reach the same goal. In order to make the turtle rotate 45 degrees, the children previously had to press the *right* button nine times. Now, they can simply press the number nine followed by button *right*. Still, all movement and rotation commands are executed instantaneously.

3. A third module allows sequences of commands to be stored and executed all at once. For this, the third box has four buttons named *start remembering*, *stop remembering*, *do it*, and *forget*. In addition, the module comes with an external display that shows which commands are currently stored in memory and prepared for later execution. Once the button *start remembering* is hit, each action is stored in memory and shown on an external display (rather than being executed immediately). The button *do it* executes all commands that have previously been stored. The button *stop remembering* stops the memorization process and leads back to instantaneous execution. The button *forget it* finally clears the memory and leaves the screen empty for a new sequence of commands to be remembered. With this mode, it is possible to repeat a sequence of commands by first storing it in memory and then executing it repeatedly by pressing a number along with the *do it* button.

4. The last module is called “four procedure box” and it allows students to store and retrieve up to four sequences of commands (whereas before only one single sequence of commands was available). This fourth box comes with four buttons in four colors (red, green, blue, and yellow). A sequence of commands can be memorized under the name “red” by pressing *start remembering* first and then pressing the red button. From that moment on (until *stop remembering* is pressed) all subsequently pressed buttons are stored in memory and are associated with the red button. Such a command sequence can be executed by pressing *do it* along with a color (such
as red). The combination do it red can also be used as part of other procedures such as blue or green which matches the concept of sub-procedures. In fact, do it red can even be used as a sub-procedure of itself which thereby results in an infinite recursion.

With her work, Perlman showed that children as young as four years are able to program. One important aspect is the question of how to teach programming in a spiral curriculum. Perlman’s approach allows students to incrementally explore topics such as basic commands, sequences of commands, repetition, procedures, sub-procedures, and recursion. One topic that is still missing in this list is the concept of parameters. In Chapter 3 we will present our own (related) approach to programming and discuss similarities and differences to the approach Perlman proposed almost fifty years ago.

2.3 Programming Environments for Logo

Logo’s history dates back more than half a century; time enough for several hundred Logo implementations to be developed in the meantime. Most environments that were developed in the past decades build on one another either in terms of their design, dialect, or their underlying philosophy. Researchers from the university of Sofia have collected a list of more than 300 Logo implementations whose ancestry they traced back to build a genealogy tree [37]. In this section, we present the history of our own implementation and how it has taken on ideas and concepts from previous Logo versions.

2.3.1 Back to the Roots – XLogoOnline’s Ancestors

The programming environment presented in this thesis is a descendant of a series of several previous environments which lead back to the first version of Logo that was implemented in 1966–1967 at Bolt, Beranek and Newman (BBN). According to the Logo genealogy project [37], there are about ten significant and well-known ancestors which connect our work with the very first implementation of Logo. We present these ten versions of Logo and highlight the merits of each of them in order to finally elaborate on the significance of our own work in comparison. The basis of the information provided in the following subsections stems largely from the article History of Logo written by Cynthia Solomon, Brian Harvey, Ken Kahn, Henry Lieberman, Mark L. Miller, Margaret Minsky, Artemis Papert, and Brian Silverman [170].
TO NOUN
  OUTPUT PICK [CATS DOGS MICE]
END

TO VERB
  OUTPUT PICK [CHASE [PLAY WITH]]
END

TO ADJECTIVE
  OUTPUT PICK [YOUNG OLD]
END

PRINT (SENTENCE ADJECTIVE NOUN VERB ADJECTIVE NOUN)

Listing 2.18 This program creates correct English sentences such as YOUNG DOGS CHASE OLD CATS or YOUNG CATS PLAY WITH YOUNG MICE. The program uses three lists which declare words belonging to either of the following three word forms: (1) nouns, (2) verbs, (3) adjectives. The last line of code creates a sentence that picks and concatenates words from these lists in a way that fits the structure of English sentences. Note how this program uses the standard Logo syntax (e.g., for procedure declarations), yet it does not produce graphical output. The first version of Logo came without turtle graphics.

BBN Logo – A Language for Everyone

The first versions of Logo were all written in a time when personal computers had not yet been invented. Instead, time-sharing systems were used, which meant that each machine was intended to be used by multiple users simultaneously. John McCarthy invented the first time-sharing system around 1962 at BBN [134].

Several years before, McCarthy had also designed the first version of Lisp which had become one of the most-used languages at the time, especially for AI purposes. So, when Feuerzeig and his team at BBN collaborated with Papert, they thought of a language similar to Lisp (that is equally expressive), however with a lighter syntax (especially in terms of parentheses). The goal was to design a language that is targeted at students of all ages, and even professionals.

As a motivating application domain for students, they came up with word games. One such example is shown in Listing 2.18 where students are asked to generate random sentences using word lists containing a number of adjectives, verbs, and nouns that can be concatenated into sentences. Apart from this example, students were also asked to implement a Pig-Latin converter or even math quizzes [170].
Daniel Bobrow implemented the first draft of such a language on one of BBN’s time-sharing machines using Lisp as the host language. For simple word games, as illustrated in Listing 2.18, many of Lisp’s built-in commands like FIRST and BUTFIRST (that is CAR and CDR) proved useful and were directly offered to the user through the Logo interface. Indeed, while this first version of Logo was supposed to be a beginner-friendly programming language for kids, it could also be used as a full-fledged Lisp dialect which allowed everything a normal Lisp would allow as well. This was a necessary requirement in order to design a language that could be used by the full spectrum of users – from beginners to experts. Some later implementations stuck with the choice to integrate an entire Lisp into their grammar, despite being hosted in other languages. The discussion sometimes goes so far as to suggest that all Logo implementations which do not contain a full Lisp implementation are “only Logo in name” [170]. We do not agree with this opinion. While Lisp is certainly one option to continue with, it is definitely not the only one. By now, Python is a viable alternative and it comes by default with a turtle graphics extension that can be used to bridge from Logo into Python.

In contrast to more recent implementations of Logo that can usually be run on many different machines and different operating systems, early versions were produced for one specific system only. Machines like SDS 940, PDP-1, and PDP-10 were available at BBN and were thus convenient platforms to be used in the first three implementations of Logo. All these systems relied on a time-sharing mechanism and could be accessed by children. For this, they used computer terminals that were located at schools and which were connected to one of these bigger machines at BBN.

MIT Logo – Introducing the Turtle

In 1969, Solomon and Papert founded a new group at MIT (so-called Turtle Group) that specifically focused on the development and continuation of the Logo idea. They added the turtle as a new microworld to Logo. At the time, displays were still rare and very expensive, and thus Papert and his colleagues strived for a “tangible” experience. Inspired by William Grey Walter’s robots Elmer and Elsie, the Turtle Group decided to build their own version of such a floor robot. Two floor robots were crafted at MIT in the year 1970, both of which could be steered using Logo.

Harold Abelson implemented the first display turtles as a virtual counterpart of floor robots. In later implementations, the support for physical floor robots was completely omitted since most schools did not have any robots at hand. Only in 1987 was the potential of floor robots rediscovered thanks to a collaboration with Lego in LEGO TC Logo for the Apple II.
Between 1973 and 1976, Perlman proposed an approach to make even young children learn to program as explained earlier in Section 2.2.3. Still, one commonality between all these different approaches (robots, screen turtles, and tangible programming interfaces) was that children interacted with a terminal that was linked to a central computing instance located at MIT. The machine at the university would then execute all given commands in a time-shared way.

**Pascal Logo – A Shift Towards Microcomputers**

With the advent of microcomputers around 1980, Logo experienced a rise in popularity and coverage. Over the next 40 years, several hundred different Logo implementations were invented. The first such implementation on a microcomputer was Pascal Logo in the year 1977. Among others, Texas Instruments had started to work on home computers which were considerably cheaper than the previously-used mainframe computers that could usually only be afforded by larger companies and universities. Cecil Howard Green, the founder of Texas Instruments, was planning to create a new generation of home computers (the TI 99/4) that was also supposed to be shipped to schools. For this purpose, he wanted Logo to be part of the system.

The TI 99/4 was a machine that came with its own individual sprite chip that would support the drawing functionality of turtle graphics. One significant constraint it had, however, was its limited memory. In its default configuration, the machine came with only 256 bytes of RAM which could be extended to at most 32kB using external expansion cards (which, however, led to a massive slowdown of the system). The only high-level programming language that was supported by the TI 99/4 was Pascal, which was therefore chosen as the host language for this new Logo implementation.

Pascal Logo was not the only Logo implementation of its era that struggled with memory constraints. The developers of Music Logo had to take drastic measures to offer a music extension and they decided not to offer turtle graphics instead. The topic of memory management was a topic in Logo long before other mainstream languages started to offer automatic garbage collection. Logo pioneers around the globe seemed to agree that `malloc()` and `free()` were not something young programmers should have to bother with – even more so on systems with tightly limited memory.

**LCSI Apple Logo – Redesigning the Syntax**

One of the most successful home computers around the eighties was the *Apple II*. It contributed to the movement away from large and expensive mainframe computers and towards smaller, more-affordable
home computers that were much more wide-spread and accessible for the broad public. Seeing that the Apple II was used more and more widely, two companies decided to implement their own version of Logo for this computer: First, Terrapin Logo was implemented, later LCSI followed with Apple Logo. Most later Logo versions are descendants of either of these two implementations.

One of the main differences between Terrapin Logo and LCSI Apple Logo lies in their different understanding of the conditional statement if. Terrapin decided for a verbose syntax of the form if-then-else (see Listing 2.19a) with the argument that this notation would favor beginners who can relate the syntax to spoken English. With this adaptation, however, conditionals become syntactic special cases. LCSI decided not to adopt Terrapin’s syntax proposal and instead went for a notation that follows the proceduralization approach known from Lisp (that is, the approach tries to treat all linguistic elements like procedure calls). Listing 2.19b shows the notation LCSI chose for conditionals. Note that the keywords then and else have been replaced by two instruction lists, the first corresponding to the then-case, the second to the else-case.

Following Lisp’s code-as-data concept, LCSI Apple Logo tried to describe the conditional statement if as a procedure with three arguments: the first argument evaluates to a Boolean, while the second and third arguments are interpreted as instruction lists. An instruction list is a dedicated data structure for representing code – in its raw form this data structure is not interpreted but just treated as a piece of information (like any other data type). Using the special command run, it is possible to parse the content of an instruction list and execute it. This corresponds to Lisp’s above-mentioned code-as-data concept.

Using the duality between code and data, even control structures like repeat and if can be treated as simple procedure calls. Listing 2.20 demonstrates how Logo’s looping construct repeat can be implemented in Logo using recursion. Once written, such a command can be used in exactly the same way as the traditional repeat statement (e.g., myrepeat 4 [fd 100 rt 90]).

Both Terrapin Logo and LCSI Apple Logo chose the same syntax for repeat, while their choices for if differ. Both sides had arguments for their respective decision: Terrapin argues that a simplified syntax would make the conditional statement more accessible for novices, while LCSI argued that
2.3 Programming Environments for Logo

Listing 2.20 Using a dedicated data representation for code, it is possible to treat even control structures like \texttt{repeat} as procedure calls. All it takes to implement \texttt{repeat} in Logo is the availability of instruction lists, the command \texttt{run}, conditional execution, and recursion.

\begin{verbatim}
 to myrepeat :n :instructionlist
 if :n>0 [ run :instructionlist
 myrepeat :n-1 :instructionlist ] end
\end{verbatim}

Listing 2.21 A simple Logo program to draw a right-angled triangle. This program does not involve object-oriented features but it is still valid syntax in ObjectLogo.

\begin{verbatim}
 fd 100 rt 90 fd 100 rt 135 fd 141
\end{verbatim}

to myrepeat :n :instructionlist
if :n>0 [ run :instructionlist
myrepeat :n-1 :instructionlist ]
end

Listing 2.21 A simple Logo program to draw a right-angled triangle. This program does not involve object-oriented features but it is still valid syntax in ObjectLogo.

their syntax was more uniform and thus especially useful for advanced programmers who wish to implement their own Logo interpreter. Both choices are sensible, although they aim at different user groups with different experience levels.

\textbf{ObjectLogo – Combining Object-Oriented Programming with Logo}

As Logo grew older, new programming paradigms evolved around it, and so some Logo versions specialized in some of these new directions. One such example is \textit{ObjectLogo}, a Logo implementation from the year 1986 which incorporated the ideas and concepts from object-oriented programming into Logo [52]. The implementation is designed to be an extension of LCSI Apple Logo and all previous teaching materials and common Logo notations could still be used. New features related to object-oriented programming were meant for advanced Logo programmers while beginners could still write the same programs as before, as the program in Listing 2.21 which draws a triangle without making use of object-oriented programming.

The same triangle can also be drawn using object-oriented programming. ObjectLogo allows programmers to create multiple instances of the Turtle which all have their own internal state and behavior. New actors can be instantiated using the common \texttt{make} command and a special command \texttt{something} that is used to create a new object (see Listing 2.22, lines 1 and 2), and each object has its own local fields and methods which can be invoked using a special \texttt{tell} command. The \texttt{tell} command takes two arguments: first, the name of an object whose state or behavior shall be changed,
Listing 2.22 In ObjectLogo, multiple turtles can be instantiated and they can be addressed individually. In this program, two turtles are used to draw a right-angled triangle in collaboration. A first agent called turtle1 is used to draw the first two sides of the triangle, another agent (named turtle2) is used to finish the work.

and second an instruction list. The object reacts by executing all instructions that were provided in the instruction list.

The example in Listing 2.22 shows how two agents (turtle1 and turtle2) collaborate to draw the same triangle as shown in Listing 2.21. In line 3, the first agent is told to draw the first two sides of a triangle. Then, turtle2 finishes the drawing by connecting the last line. Note that turtle2 is aware of the other agent’s position. The command setheading towards :turtle1 causes it to face turtle1.

In contrast to non-object-oriented versions of Logo, ObjectLogo allows each actor to have its own local methods that can be used to overload pre-existing global commands. For instance, among three turtles each could have its own local version of the command fd. One turtle may draw a thick line, another one draws the line in red, while the last turtle draws a dotted line – all reacting to the same command fd. Listing 2.23 shows how such an implementation would look like. Three actors (turtle1, turtle2, and turtle3) have their own local versions of fd. The keyword howto indicates the beginning of a local procedure definition and corresponds to the global keyword to. In all three cases, a local command fd is defined which internally relies on the global implementation of the command fd (i.e., the keyword usual invokes the global version of fd). Each actor has their own version of fd and will react differently to the statement tell :turtleX [fd 100].

ObjectLogo was an advanced implementation of Logo that aimed at teaching modular design by using the object-oriented programming paradigm. The environment allowed programmers to use several turtles which each have to their own local environment with encapsulated variables and methods. Different objects can inherit features from one another and specialize them on demand. In addition to programming turtles, ObjectLogo also allowed most UI elements to be programmed such as menus, windows, files, and dialogues. All those elements were implemented as objects.
Listing 2.23 Local procedures can be used to overload global definitions. Different objects can be specialized on demand and each will use its own local version of procedure calls during execution.

ObjectLogo was platform-dependent and could only be used on Macintosh machines. It was updated until the year 2000 but is no longer maintained and cannot be used on modern computers any longer.

**Berkeley Logo – Aiming for Standardization and Platform Independence**

By the late eighties, the computer world had stabilized around four common operating systems: DOS, Windows, Mac, and Unix. The majority of all machines ran one of these four systems and, yet, most Logo implementations were not designed to be portable between different operating systems. The consequence was a large variety of different Logo implementations, all crafted for different platforms and all with slight differences in terms of their internal implementation.

In 1988, Brian Harvey and his team at UC Berkeley decided to develop a Logo version that was portable across all common operating systems of the time. Their goal was to design a system that could be run among DOS, Windows, Unix, and Mac [170] and to standardize the graphical output by scaling it such that the graphics fit on all screens [85].

Berkeley Logo was designed as a console application, a window with only one interface for the programmer to interact with (see Figure 2.9a). This interface was a text-based interface similar to a typical command-line interpreter. Even though the application only contained one window, it still had more than one functionality this window could be used for. Specifically, the console window in Berkeley Logo was used for three different purposes:
1. **Procedure invocations**: Firstly, the window can be used to execute built-in or user-defined procedures. The user is provided with an interactive programming experience in a so-called **REPL** (i.e., read–eval–print loop); an environment that allows programmers to piecewise execute code snippets. Each command may change the program state, which can be observed and inspected by the programmer. Unlike the LISP and Python REPL however, Logo’s REPL does not print results to arithmetic calculations such as 1+2, but instead the outcome of such an arithmetic expression has to be printed explicitly using the `print` command.

2. **Procedure definitions**: New procedures can be defined using the common Logo notation `to . . . end`. Once the keyword `to` has been detected, the environment enables multi-line input which can be closed again upon entering the `end` keyword. Lines which have already been executed cannot be revoked but an existing procedure can be edited and corrected. All versions of previously-executed procedures and procedure declarations remain visible by scrolling back through the history of previous activity.

3. **Output/history/error messages**: Finally, the window can be used as an output mechanism and as a history of previous activities. Besides the text related to procedure invocation, declaration and their respective output, the window can also be used to issue error messages in case the input was not correct.

For turtle graphics applications, the window is split in two parts as shown in Figure 2.9b. While the bottom part of the window is the same as before, the top part is used to illustrate the corresponding graphical output. The top part shows a canvas with a turtle. All movement and rotation commands leave a visible effect and can be observed on this screen, just as in all other implementations that make use of turtle graphics.

Berkeley Logo is one of the oldest Logo versions that is still running today and works on modern computers. It has influenced multiple environments between the nineties and today. One main point that has been improved in more recent versions is the graphical user interface, as we will see in the next section.

**MSWLogo/XLogo/XLogo4Schools – Advanced Graphical User Interfaces**

In the eighties, graphical user interfaces slowly started to evolve and during the nineties this progress continued. Step by step, the previously used command-line interfaces, as known in DOS, were
replaced by graphical ones. Windows played an important role in this process. While MS-DOS was still widely used in the eighties, Windows started to move away from DOS with the introduction of Windows 3.0 – a system that came with an advanced graphical user interface which managed to appeal the population. Windows 95 was the first Windows version that had MS-DOS fully integrated into the system (in contrast to the previous standalone DOS systems which could be booted without a graphical user interface). Microsoft swiftly moved away from DOS and towards a more graphical interface which made the computer more accessible to the wider public.

In 1993, a new Logo version called MSWLogo (Microsoft Windows Logo) was published specifically for Microsoft Windows. This Logo version used Berkeley Logo as its core, but the designers enhanced it with a couple of additional features – first and foremost an improved user interface (see Figures 2.10 and 2.11). Instead of just one command-line window, MSWLogo provided three windows for three different functionalities: (i) a canvas (used for graphical output), (ii) an editor (used for displaying procedure declarations) and (iii), an input line with included history.

Debugging features were previously hidden features a programmer would only know of if she or he carefully read the user manual. In MSWLogo, debugging features such as pause and trace were added as visual buttons on the user interface and they allowed users to make use of debugging features if they experienced an error in their code.

MSWLogo was widely used among Windows users but unfortunately it could not be used on operating systems other than Windows. For this reason, two more Logo versions were developed
called XLogo and XLogo4Schools. Both of them were platform independent due to building on top of Java. Both XLogo and XLogo4Schools were widely used in programming courses across Switzerland.

XLogoOnline

In 2016, the development of XLogoOnline started as the main contribution of a master thesis at ETH Zurich [173]. XLogoOnline is the last Logo version in the genealogy presented in this chapter. It builds on top of most Logo versions that were presented in this chapter, and it adapted various aspects like language syntax and semantics, design, and portability considerations from previous versions.

The following four attributes characterize the environment’s most important features:

- **Focus on K–6**: Logo was designed as a language that suits both the needs of beginners and experts. That is why many early versions included highly advanced features from the Lisp universe. But in the end, Logo was established mostly as a language for novices and it could not gain the same popularity among experts as it had among novices. Nowadays, advanced programmers use languages such as Python which by now has its own history as an educational language. In our work, we decided to focus on the age spectrum from kindergarten to sixth grade of primary school. Once children enter secondary school, they slowly transition away from Logo into the world of Python using suitable environments. For this reason we have focused our work on teaching programming to children. The programming environment XLogoOnline is not designed for teaching advanced users and instead focuses heavily on teaching certain basic concepts such as sequential execution of commands, repetition,
modularity, and parameterization. Other concepts such as variables and recursion can be implemented, but they are not the main focus of this work.

- **Platform independence**: Having software that is not bound to one platform but that can be used from many different devices and operating systems was already considered an important feature thirty years ago when Berkeley Logo was designed. The same feature is still relevant today and it is especially important in the context of schools. Note that IT administrators at Swiss schools invest an average of 2.5 hours per computer and school year in the installation and maintenance of software [76]. This is a substantial amount of time considering that usually teachers have to perform this work by themselves, in addition to their preparation for school. In other words, the easier it is to maintain software, the more time a teacher can spend on the preparation of lessons instead. We hence decided for JavaScript as our host language. JavaScript is a core part of web browsers and a majority of all devices come with a browser installed. In addition to portability between platforms, the choice of web technologies also allows for simple deployment without the requirement to installing or maintaining any software on the client side.

- **Proactive error handling**: Learning to cope with errors is one of the most crucial aspects of a young programmer’s experience. Bugs cannot be prevented and every programmer eventually needs to be able to identify, locate, and fix programming errors by her- or himself. From the beginning, Logo had built-in error detection mechanisms. These tools were however comparatively crude considering the great progress the professional community has made in terms of debugging support throughout the last decades. In our work, we explore the area of debugging and error detection. We present several tools that support Logo novices in detecting and fixing programming errors autonomously.

- **Support for physical and virtual turtles**: Although the first versions of Logo usually came with an integrated support for both physical floor turtles and virtual screen turtles, most environments in the eighties and nineties did not include support for floor robots any longer since schools “did not typically have any floor turtles they could connect with” [170]. Nowadays, physical robots have regained popularity and many Swiss schools bought physical floor robots for educational purposes. We have decided to provide potential connections to several floor robots such that each age group can choose between the screen turtle or a floor robot with the same functionality to accomplish the same tasks.
As just seen, XLogoOnline is part of a long and extensive history of how the programming language Logo and its corresponding learning environments have evolved over time. Contrary to the voices claiming that Logo’s Turtle was extinct just twenty years ago [178], Switzerland has a constantly growing number of Logo programmers every year thanks to our contribution.

2.4 Practical Experiences with Programming Novices

Learning to program is hard. From issues in the problem domain to issues in the programming domain to issues in the conversion between the two, a wide variety of problems can and do occur on a regular basis when novices learn to program [38, 109]. As teachers and educators try to understand which difficulties novices experience, a number of problem areas have been reported in previous research. Several literature reviews [53, 161, 128] provide useful insights in where this research domain is currently standing and what tangents it is likely to experience in the next decades. We highlight the diverse field of programming errors by discussing five specific domains where potential sources of errors can be found.

1. Problems in general orientation: Some programmers struggle to understand the purpose of programming. They are not aware of the kind of problems which can be solved using programming and why it is worth investing time and effort into learning how to program. This problem is one of the first problems novices may encounter. The occurrence of such problems also depends on the choice and way of how a problem domain is motivated [46]. Papert’s turtle provides a problem domain that is familiar and motivating for even young children of all ages and ability levels [156].

2. Problems in understanding the notional machine of program execution: A second problem is connected to novices not understanding how to steer the notional machine they are working on. Du Boulay explains the semantics of a programming language as an abstract machine. Each programming language comes with its own notional machine and Du Boulay claims that a large class of problems in novice programming is based on poor mental models of how program execution works. Moreover, he claims that young programmers struggle to understand the connections between the notional machine they work with when they program and the physical machine they sit in front of [53].
3. **Problems in the notation of programs:** The third problem is connected to the syntactic precision required when communicating with a computer. Humans communicating in natural languages are usually more forgiving in terms of syntactic precision than a computer. Novices learning to program, among other things, have to learn how to express themselves in a formal language that was designed to communicate with a machine [169]. Learning the syntax and the underlying semantics of such a language is hard and often results in various programming errors [61, 133, 102, 28, 130].

4. **Problems in planning a strategy:** The next problem domain is not specifically limited to programming; *problem-solving* (i.e., a key aspect of *computational thinking*) is considered one of the core contributions of computer science to general education [193]. Learning to solve problems algorithmically is one of the inherent difficulties programmers face and it takes time and effort to become a proficient problem-solver who is able to express his or her ideas in a programming language [194].

5. **Problems in mastering the pragmatics of programming:** Programmers need to learn a number of skills that are required to develop code using block- or text-based programming interfaces [166], test and debug their code [140] using whatever tools they are given. Morales, Rusu, Botella, and Quinones point out that one of the most significant influences on the experience of a programmer is related to the tools they use and in the choice of the programming environment [137].

These five areas are non-exclusive and most programmers struggle with issues from more than one domain at once. These issues need to be kept in mind when designing programming languages, application domains, teaching materials, and error diagnosis tools for novices.

### 2.5 Logo as a Philosophy of Education

Logo is not only a programming language but also a philosophy of education that evolved with and around the actual programming language [14]. The theory was greatly influenced by the constructivist and constructionist understanding of education. In this section, we explore the two psychological theories which mark the core of the Logo philosophy.
2.5.1 Learning by Doing – Constructivism

Constructivism is a famous psychological learning theory which suggests that learners have to actively engage with the learning content in order to construct their own knowledge. At the core of this theory is the idea to make learners explore mental models actively at first hand. The first efforts in this area were conducted by Jean Piaget whose work received major attention during the second half of the twentieth century.

Piaget’s theory of constructivism explains how children grow gradually from being concrete and sensomotoric thinkers into young adults with the ability to perform coherent logical and abstract thinking. He describes four relevant stages, which are passed along the way [152]:

1. Sensomotoric stage (age 0-2 years): In the first stage, children are primarily focused on learning through their own bodies. Children react to sensory events and gain experience through physical movement and activity. At this early stage, children are not yet able to put themselves in the position of others.

2. Preoperational stage (age 2-7 years): In the second stage, logical conclusions are still rather weak, but learners discover the symbolic representation of information. Even at this stage, it is still difficult for children to adopt the perspective of another person and to look at situations from that person’s point of view. It is difficult for children to perform operations on a purely mental level and to predict cause and effect.

3. Concrete operational stage (age 7-11 years): In the third stage, the children develop the ability to draw logical conclusions and understand the relation between cause and effect. Children now develop the ability to put themselves in the position of others and adopt their perspective.

4. Formal operational stage (age 11 and older): In the final stage of a child’s cognitive development, the child develops the ability to think abstractly. Children are able to work through tasks in their minds. Children now continuously develop into competent problem solvers.

Logo allows various skills, summarized by Piaget in these four stages, to be trained from early on. We developed a concept for teaching programming that supports the cognitive developments of children according to the principles of constructivism – as well as the related principles of constructionism which will be explained in the next section.
2.5 Logo as a Philosophy of Education

2.5.2 Learning by Making – Constructionism

A second learning theory, which is closely related to constructivism, was developed by Seymour Papert between the sixties and seventies of the last century. Papert had previously worked together with Piaget and then developed his own learning theory based on their common understanding. He playfully called it constructionism. Papert explains the differences and similarities between the two approaches as follows [145]:

“Constructionism […] shares constructivism’s view of learning as ‘building knowledge structures’ through progressive internalization of actions […] It then adds the idea that this happens especially felicitously in a context where the learner is consciously engaged in constructing a public entity, whether it’s a sand castle on the beach or a theory of the universe.”

That is, while both constructivism and constructionism claim that learners should construct new knowledge actively, they set different focus points. In his theory, Piaget mostly focuses on what children can or cannot do at different development stages, and he consequently built a model of how the human mind develops overall. However, he does not propose a specific technique on how to introduce constructivism to schools. This is one of the points where Papert put his main attention.

Papert proposes some concrete approaches on how constructivism can be used in teaching and learning. One example is his idea of using an “object-to-think-with” [144]. A concrete tool, he claims, can support people to construct their own knowledge even about abstract and difficult concepts, like mathematics. Ackermann argues [16] that this aspect makes Papert’s constructionism both more situated and more pragmatic than Piaget’s constructivism, although seeking the same overall goals.

Papert argues that schools should take a student-centered approach, where the unique interests of individuals can be explored and new knowledge can be acquired according to the ideas of classical constructivism, rather than merely learning to apply well-known procedures. In fact, everyday schooling and especially the use of computers was initially so far away from Papert’s idea that he understood Logo as an “anti-schooling project” [21].

In the meantime, Logo has found its way into thousands of schools around the world. From curricula [94, 80, 87], to learning environments [173, 32, 168, 179, 112] and field tests [127, 171, 93], Logo has become the object of much research around the globe over the past fifty years. In the remaining part of this thesis, we present our contribution to the success story of the programming
Background

language Logo. In concrete terms, we present a learning environment that enables programming lessons from kindergarten to sixth grade and which provides various tools to facilitate autonomous learning in all age groups.
Chapter 3

How to Integrate Logo Into the Swiss School System

Logo represented a critique against the traditions of the US school system during the seventies and eighties [21]. The background of this story is that, at that time, technology consumption in US schools started ramping up and computer-aided instruction gained widespread acceptance. During this phase, computers were installed in many classrooms around the country, however, they were rarely used for constructive activities such as programming. Papert opposed this practise and proposed computers to be used as a tool for creative problem solving instead of drill-and-practice routines. That is, rather than putting students into the role of technology consumers, Papert envisioned them to become creative inventors of technology. Seeing the path computer-aided instruction had taken, Papert considered Logo an “anti-schooling project,” which in many ways contradicted the common trend that the school system seemed to be adopting at the time [21].

After several years of pushing both the technological and the political frontlines, Papert and his colleagues reached a significant number of schools which were actively working with Logo and which finally laid the foundation for what we still consider the basis of modern computer science education and programming classes. Amazingly, it still took many more decades until the educational community finally attributed the same importance to educational programming as Papert. Even today, the idea of teaching programming in schools has not yet fully been established in all developed countries. Initiatives like the “Programma il futuro” in Italy [11], the “Computer Science For All” in the USA [4], or Japan’s “Minna No Code” [10] show that there is progress and in Switzerland,
too, there has been a change of mindset. With the latest school reform called Lehrplan 21, computer science finally has become a fundamental part of the curriculum and will be taught from an early age on to all Swiss students attending public school in any of the 21 German-speaking cantons of the country.

In this chapter, we present the current situation in Switzerland in regard to programming education in primary schools. We discuss our approach of how Logo can be taught in a spiral curriculum from kindergarten to sixth grade and we highlight the connections between the presented approach and the general learning objectives of Lehrplan 21. Several examples will be given of how programming lessons can be designed in a cross-curricular context and how the specific challenges of teaching in the lower grades can be tackled with programming concepts being taught despite most children not being literate yet.

3.1 Starting Point Switzerland

Switzerland is a country that is shaped by its federative regime – a fact that can also be recognized in its educational system. Most of the responsibilities for the overall educational system in this country lie with the individual cantons which comprises 26 distinct administrative divisions. The result is a highly diverse overall structure in which the experiences of individuals can vary greatly depending on which canton they go to school to. In 2007, the population voted in favor of harmonizing the educational system and, as a result, three cross-cantonal curricula were devised: (i) The “Lehrplan 21” which covers the German-speaking parts of the country, (ii) the “Plan d’études romand” for the French-speaking parts of the country, and (iii) the “Piano di studio” for the Italian-speaking territories. The three curricula reach different percentages of the overall population: more than 60% of the population speak German, more than 20% speak French and, only just under 10% of the population speak Italian [78]. A consequence of these demographics is that a majority of students in Switzerland are taught by the rules of Lehrplan 21.

Lehrplan 21 divides the 11 school years from kindergarten to the end of compulsory education (i.e., the 9th grade) into a total of three cycles: the first cycle comprises two years of kindergarten and the first two years of primary school. The second cycle follows on and covers grades 3 to 6, and the third cycle continues with three years of secondary school (grades 7 to 9). Of the three cycles, the
first one is mainly aimed at the interdisciplinary development of the children, whereas the second and third cycles are largely aimed at subject-specific teaching.

Specifically for the first cycle, there are a number of topics defined as learning objectives which are subject-independent and thus ought to be achieved by all subjects jointly. These learning objectives are fostered through nine so-called development-oriented approaches that ensure the high-level development of each child and which cover the following topics: (i) body, health, and motor skills, (ii) perception, (iii) temporal orientation, (iv) spatial orientation, (v) relationships and patterns, (vi) imagination and creativity, (vii) learning and reflection, (viii) language and communication, and (ix) independence and social interaction.

The learning objectives in the second and third cycle are meant to be achieved in both subject-specific and interdisciplinary manners. The former is primarily established through dedicated subjects such as (i) languages, (ii) mathematics, (iii) nature, people, and environment, (iv) arts, (v) music, and (vi) sports. All of these subjects have their own fixed time allocation and extend over the entire duration of compulsory schooling. For interdisciplinary teaching across the mentioned subjects, the curriculum provides a small number of modules. One such module is called “Medien und Informatik” which is aimed (at least partly) at providing computer science competencies. Modules are generally assigned a more interdisciplinary character and, in contrast to subjects, they do not have a continuous time allocation and they have a more restricted overall time budget [49]. From kindergarten to grade 4, there is no dedicated time vessel allocated for computer science, but the curriculum requires computer science competencies to be taught in a cross-curricular manner. From fifth grade on, there is an additional time budget of one hour per week allocated to the module “Medien und Informatik.”

The competencies in Lehrplan 21 are generally clustered in competence fields which cover specific topics or domains. One such competence field in the module “Medien und Informatik” addresses competencies around problem solving and programming. It counts six objectives (see Table 3.1) that students ought to achieve by the end of sixth grade [59]. In order to achieve these goals despite the tight time constraints and to ensure a sustainable learning effect, it is worthwhile to start early and to use a curriculum which aims at a continuous acquisition of competencies – a spiral curriculum. In the following sections, we present such a curriculum that sequentializes the learning goals listed in Table 3.1 in a continuously evolving way.
Table 3.1 Six objectives related to problem-solving and programming must be tackled within the first six years of primary school.

### 3.2 A Continuous Plan from Kindergarten to Grade 6

Hromkovič et al. propose a spiral curriculum that aims at the mentioned learning objectives and shows how these goals could be reached with fifth and sixth graders [95, 94]. We extend this idea and show how the same concepts can be sequentialized such that an earlier intervention is possible and even kindergarten children can start programming. To this end, we break down some of the learning goals mentioned in Lehrplan 21 into smaller sub-goals, which we address individually in seven stages.

#### 3.2.1 Stage 1: Basic Commands and Command Sequences

Two learning objectives mark the core of the first steps in our approach. Learners explore the concept of a computer and build an intuition of what it can and cannot do. They understand programming as a form of communication and they gain an understanding of how programs can be constructed from basic commands. In this stage, we focus on the following two learning objectives:

*The students understand that a computer can only execute predefined instructions. (§5a)*

*The students understand that a program is a sequence of instructions. (§5b)*

Students often have striking misconceptions about what capabilities a computer has and how those capabilities differ from those of a human being [147]. Fundamentally, a computer is just a machine and it does not have any human intellect. Consequently, it is also not capable of interpreting ambiguous or imprecise statements and it typically is oblivious of what a programmer wants to achieve. One of the first things a young programmer thus needs to learn is the fact that a computer is
a reliable but meticulous conversational partner and that, in order to make this conversation work, they need to learn a language that the computer “understands.” A computer’s behavior is entirely determined by the content of the computer program it is provided with – a piece of text written in a programming language.

Programs decompose into a sequence of linguistic elements called *commands*. In the context of turtle graphics, four commands permit programmers to navigate a computerized turtle to move around on a two-dimensional surface. The turtle reacts by either (a) moving forward or backward or (b) by rotating on the spot to the left or to the right. In the context of programming classes in the first cycle, we chose these four commands not to be parameterized, meaning that commands such as `forward` and `right` (which ordinarily require a parameter) can be used as standalone commands without parameters. Distances in movement commands are measured in terms of a *unit distance* where one unit is defined through the concept of a *field*. Similarly, the two commands `right` and `left` also do not require a parameter in the first stage and instead just cause the turtle to rotate ninety degrees to the right or to the left, respectively.

Using just these four non-parameterized commands, students can already explore a large variety of interesting behaviors. Consider the example shown in Figure 3.1, where a programmer is asked to navigate the turtle to a strawberry located one field ahead and one field to the left of the turtle’s current position. Since the turtle cannot move diagonally from one field to the other, it has to pass through the one field that connects the starting point with the destination. While the strawberry cannot be reached in just one step, it is possible to reach it using three commands: first, the turtle moves to the adjacent field in a `forward` movement. Once arrived, the turtle turns to the `left` such that it faces the strawberry. After a last `forward` movement, it finally arrives at its destination.

In order to build an intuition for the effect a command has, young programmers first execute one command after the other to see each effect immediately. Doing so allows them to correct for errors on the fly. This is especially useful for beginners of a young age who are likely to confuse left and right [118].

Trying to solve this task, students learn that it is virtually impossible to “jump” from the starting point to the destination in just one step. There is no basic instruction available to do so and, consequently, young programmers experience the limits of the language they work with – they see that the computer can only execute predefined instructions. The fact that the used programming language only contains four commands may seem like a massive limitation, but by combining multiple commands,
How to Integrate Logo Into the Swiss School System

Find the strawberry.

Figure 3.1 The turtle cannot move diagonally across fields and, in order to reach the strawberry, it must pass through the connecting intermediate field. This can be achieved using three commands: (i) forward, (ii) left, (iii) forward. In the beginning, programmers execute all three commands individually to see their effect instantaneously.

much more than just four behaviors can be achieved. A series of commands can be arranged in a program to create a multitude of different behaviors; all it takes is a clever algorithm.

The size of a programming language is not representative for its expressiveness and, although our approach provides young programmers with a rather small vocabulary, they still achieve to create a large number of complex behaviors. The example in Figure 3.1 illustrates this point: in order to make the turtle move diagonally, children learn to assemble multiple commands and create a program. They learn that a program is just a sequence of (basic) commands which allows them create behaviors that are not defined in the language’s basic vocabulary. Rather than searching for a specific command within a large toolbox, students learn to create new behaviors by writing their own programs.

3.2.2 Stage 2: Strategies and Testing

Once students gained an understanding of the vocabulary and they know how to create longer programs for custom behaviors, they are ready for the next step aiming towards algorithm design. In this stage, students learn about the diversity of the solution space to a given problem, they reflect what impacts different solutions have, and they compare various strategies against one another. Different turtle graphics algorithms typically have distinct attributes (e.g., readability, descriptive length, and traveled distance). Children learn to adapt their algorithms to given restrictions and retrace their solutions in case of errors. In this stage, we focus on the following learning objectives:

The students are able to recognize and follow formal instructions. (§1)
The students are able to find solution strategies to simple problems. (§2a)

The students compare different approaches to solve the same problem. (§2b)

The students check their strategy for correctness. (§2c)

School classes are known to be heterogenous groups of individuals, and, in order to take these personal differences into account, teachers need rich tasks that are accessible to all learners, regardless of their prior knowledge. For mathematics, such tasks are called low treshold high ceiling tasks [13, 45]. In the context of programming, the same approach can be used simply by introducing additional restrictions on a given task. That is, by adapting the task, some children might try to find a simple path from A to B, others might look for the shortest path, and still others solve the same task using only a reduced instruction set. On the surface, all children work on the same task, but on closer consideration, they all face individual challenges and work at their individual level of expertise. After some explorations and group discussions, students are able to find multiple solutions to the same problems and they compare different approaches.

Programming obviously does not only allow for multiple correct solutions but also countless incorrect ones. Thus, to verify their approach, children need to test their solutions for correctness. Turtle graphics is characterized by its ease of program verification that can be directly performed by the students themselves. Neither a teacher nor a solution sheet is necessary to check the correctness of a given solution and all it takes to verify a strategy is a computer. By executing a program on a computer, learners can easily verify whether their program works as intended: if the resulting effect matches their expectation, the students can simply continue with their work; but if it does not, young programmers encounter yet another important aspect of programming, namely troubleshooting.

Making errors is a natural and unavoidable part of programming, as even small logic errors can cause obvious graphical defects. In order to detect such flaws at an early stage, programmers initially execute commands individually, i.e., they observe the effect of every single command immediately and individually. This allows them to know exactly where a flaw was introduced. Fitzgerald et al. [71] have shown that programmers generally seem to struggle more with error localization than the actual repair. In fact, once an error has been located, it generally can be resolved autonomously most of the times. Consequently, if all commands are executed individually, the challenge of troubleshooting is simplified drastically. Only once learners begin to write longer programs and plan ahead, they need more advanced debugging skills. From that moment on, it is important that students are able to
recognize and follow formal instructions, just as if they had to execute their program mentally, step by step.

3.2.3 Stage 3: Looping Single Instructions

Next, learners deal with the concept of repetition for the first time. In this stage, the concept of repetition is not yet used to draw complex patterns, but it is simply aimed at repeating individual instructions, e.g., the movement command forward. This approach suits two purposes: (a) it allows programmers to shorten long programs by combining identical consecutive commands, while (b) it also serves as an introduction to the concept of parameters which will be explored in even more depth in stage 4 and 6. The following learning goals mark the core of this stage:

The students recognize, describe, and structure processes with loops. (§3a)

The students can read and manually execute program sequences with loops. (§4a)

The students can read and manually execute program sequences with parameters. (§4c)

The students write and test programs with loops. (§6a)

The students write and test programs with parameters. (§6c)

Up to this point, programmers only used non-parametrized commands, which arguably might be easier for beginners; however, it can also affect the length of a program in a negative way. This effect is illustrated in Figures 3.2 and 3.3 where three squares are drawn at different sizes. For a small square of one field per side, eight commands are required (see the topmost program in Figure 3.3). If the square is supposed to count two fields per side, the program grows to twelve commands (see the second program of Figure 3.3). For a large square with three fields per side, it takes sixteen commands (see the program on the bottom of Figure 3.3). Think about how much effort it would take to extend this logic to squares of size ten or twenty – the resulting programs would be ever longer and harder to read while their description would be much more error-prone.

The three programs become increasingly longer although they all produce similar visual outputs. The reason for this phenomenon is that more and more movement commands must be gathered in close succession in order to increase the side length of each square. Once the problem has been acknowledged, repetition is introduced as a simple remedy. Just as Perlman suggested in her approach [149, 150], we start by exclusively repeating single commands. Instead of placing five
Figure 3.2 Three related tasks which all require a square to be drawn. Depending on the dimension of the respective square, however, programs differ in their descriptive length: A small square takes eight commands, a slightly larger one requires twelve commands, or even sixteen for a larger square.

Figure 3.3 Three programs which solve the tasks shown in Figure 3.2. The three programs clearly differ in length although all three draw the same fundamental shape, namely a square.

identical consecutive movement commands, students learn to use a single movement command that is wrapped in a repeat statement. Figure 3.4 shows a program which draws a square of size five with only 12 commands. This solution can be applied to squares of any size without a need to increase the number of commands.

In this stage, students learn to use the concept of repetition as a means to parameterize a command. In the next stage, we extend this functionality by parameterizing all basic movement and rotation commands. In this way, it is possible to draw squares of any size without the use of repetition.

Figure 3.4 Thanks to the concept of repetition, it is possible to draw squares of any size with a maximum of 12 commands. This is a significant reduction compared to otherwise continuously increasing program lengths. This program with 12 commands produces a square of size five – creating the same result without repetition would require twice as many commands.
3.2.4 Stage 4: Parameters in Basic Commands

The previous three stages took place in a context where no basic movement and rotation commands were parametrized yet. All movements were measured in unit distances (for which we used a grid of fields) and both rotation commands were limited to right angles only. In this stage, we extend the semantics of the four primitive movement and rotation commands by parameters. That is, programmers now need to indicate distances in units of pixels and angles as numeric degrees. Doing so allows them to describe lengthy programs even shorter and they learn to draw a variety of new patterns that were not possible before. The learning goal in this stage is concerned with the topic of parametrization and can be summarized by the following two objectives:

The students can read and manually execute program sequences with parameters. (§4c)

The students write and test programs with parameters. (§6c)

The students already started composing programs that describe long and cumbersome tasks in fewer words. They learned that repetitive manual labor can be automated, e.g., by using the command repeat. By introducing a parameter, it is now possible to reduce the program length even further and combine identical consecutive commands in just one command by adding a parameter to it. This change has an important implication: While navigation used to take place on a coarse grid of fields, it now changed to a bitmap using a single pixel as the unit distance. With each movement the turtle makes, it colors a number of pixels that is determined by the given parameter value.

The same structural properties that could be observed in squares earlier are also evident now. That is, programs that draw the same shape at different dimensions will keep the same overall structure and they differ only in the value of their respective parameters. This effect is illustrated in Figure 3.5 which shows three programs that draw squares in different sizes. All three programs exhibit the same form (i.e., they use the same commands in the same order). The only difference consists in the choice of the parameter values.

By the end of this stage, students know how each of the four basic commands can be used in a parameterized form. They use this adapted command set to draw geometric shapes with lines of differing lengths and angles. Moreover, students analyzed the structural characteristics of different programs, compared them and tried to find short and elegant solutions to different problems. In the next stage, these thoughts will be deepened and the students discover how the already known concept
3.2 A Continuous Plan from Kindergarten to Grade 6

3.2.5 Stage 5: Looping Over Multiple Instructions

The concept of a loop is a central and important element in a programmer’s toolbox and in previous stages, students already covered the basic idea of repetition. In this stage, students learn to draw more complex constructs which involve multiple different commands that are used repeatedly. The resulting programs are short but logically complex and their creation indicates a cognitive challenge for beginners. That is, in order to write programs with loops, young programmers must be able to recognize repetitive structures, describe them in programs, test the result, and, in case of an error, retrace and correct their program manually. These points form the basis of the three learning goals that are tackled in this phase:

The students can recognize, describe and structure processes with loops. (§3a)

The students can read and manually execute sequences with loops. (§4)

The students can write and test programs with loops. (§6a)

The first step in this learning process consists in identifying repeating structures in existing programs. All squares have the same basic structure, namely, the same sequence of two commands, e.g., `fd 100 rt 90`, which is repeated four times in a row. Once young programmers are able to recognize such patterns, they only have to convert the full program into a shortened notation using the `repeat` command whose functionality was already partially covered in stage 3.
How to Integrate Logo Into the Swiss School System

Listing 3.1 This program draws squares of size 100.

repeat 4 [fd 100 rt 90]

Listing 3.2 Same effect with repeat.

Figure 3.6 Students who know how a long program can be shortened using loops, start identifying repeating structures in visual artifacts such as this staircase.

The conversion from a full program to its shortened version with repeat is a simple manual process which can be seen in Listings 3.1 and 3.2 – both code snippets produce the same visual result, namely a square of size 100. The program on the left counts eight commands where the same two instructions are executed repeatedly on four consecutive lines. In contrast, the program on the right uses the looping construct repeat to merge the four identical lines into a single, short line of code.

Once students understand the mechanism of how a given program can be shortened using repeat, they learn to write programs for pictures which contain the same graphical elements several times. One such example is shown in Figure 3.6, showing a staircase with eight identical steps.

Students need to master the following three steps in order to draw a complex shape such as a staircase using repeat:

1. Writing a base program: First, students study the image and try to identify repetitive structures (e.g., individual steps in a staircase). These structures can then be formalized as a short base program. In our example, the base program is a command sequence that draws a single step, such as: fd 100 rt 90 fd 100
2. **Establishing loop invariants**: Simply wrapping a `repeat` statement around a base program is often not enough. The program `repeat 8 [fd 100 rt 90 fd 100]`, for instance, does not result in the desired staircase but a square. The reason for this peculiarity lies in an implicit assumption that the turtle will always start a new loop iteration in the same orientation (i.e., facing up). Before wrapping the base program in a loop, it is important to establish this state. In the staircase example, this means that the base program must be extended by an additional left turn: `fd 100 rt 90 fd 100 lt 90`.

3. **Assembling the solution**: The last step simply consists of wrapping the adapted base program into a loop. A wide range of different tasks can be solved according to this pattern, but the procedure is likely to put beginners to the test if steps 1 or 2 have not been carefully thought through. In this case, programmers need to debug their solution.

Learners must not only be able to read programs and execute them manually, but also acquire a strategy to problem solving. In our case, we strive for a bottom-up problem solving approach, in which larger and more complex structures are continuously assembled up from smaller and previously tested building blocks. As abstraction increases, it is advisable to adopt a modular approach and to pack frequently-used programs into new procedures [97].

### 3.2.6 Stage 6: Modularity and Parametrization

Logo’s basic vocabulary is deliberately kept small and consists of only a handful of commands. However, after a while, programs reach such a high level of complexity that programs become difficult to understand. To avoid this, most programming languages (including Logo) allow their users to encapsulate command sequences into new modules (i.e., procedures) and provide them with a meaningful name. Later on, students learn to parameterize their own procedures and thus create commands that suite a more universal purpose. Students deepen their understanding of parameters and learn not only to use parameters in primitive commands, but also to define parameterized commands on their own. The central learning goals in this stage are two-fold:

- **The students can read and manually execute program sequences with parameters.** (§5)
- **The students can write and test programs with parameters.** (§6)

A programming language consists of a set of words that can be used to create a variety of different behaviors. While so far repetition has made it possible to shorten long programs with recurring
to stairs
repeat 8 [ fd 100 rt 90 fd 100 lt 90]
end

Listing 3.3 Using the keywords to and end, programmers learn to define their own custom commands. In this case, a new command is defined with the name stairs and upon execution, the program will draw a staircase with eight steps each of size 100.

Figure 3.7 Four different staircases which differ in the choice of the given parameter values. All four staircases have eight steps but with steps of differing height and width. The first staircase (from the left) uses steps of height 100 and width 100. The second one uses steps of height 50 and width 50. The third one uses steps of height 100 and width 50. The last staircase uses steps of height 50 and width 100.

With only a few adjustments in the program shown in Listing 3.3, it is possible to draw numerous different staircases (see Figure 3.7). For example, adjusting the number of repetitions from 8 to 12 results in a staircase with 12 steps instead of 8. In contrast, changing the parameters of both fd commands from 100 to 50, the resulting staircase is only half as high and half as wide as before. Even changing the slope of the staircase is possible by simply passing in different parameters for the two commands fd.

Children who write and compare programs for different stairs are likely to arrive at the conclusion that all of the programs share the same underlying structure and ultimately differ only in the choice of the parameter values. This insight is a key step towards writing parameterized procedures. By
3.2 A Continuous Plan from Kindergarten to Grade 6

Listing 3.4 Parameters such as :steps, :height, and :width indicate placeholders for values that will be provided at runtime.

extending their programs with parameters, as shown in Listing 3.4, students are able to create any of the prior staircases with one single custom-built command.

In case of an error, it is crucial that young programmers are able to follow the program flow from the input line at the topmost level of abstraction through procedures and sub-procedures. They need to understand how arguments are passed from the input line to custom procedures which appears to be hard to grasp for some novices as we will show in later parts of this thesis.

3.2.7 Stage 7: Conditional Statements

While the idea of parametrization allows for a large number of different effects to be consolidated in one single program, it also holds the danger that a program might be used with unintended input. For example, what result should be drawn if a user asks for a staircase with -4 steps? In this stage, students learn to use conditional statements as a means of intercepting unintended input to their own custom-built procedures. The following objectives are at the core of this last stage:

The students can read and manually execute sequences with conditional statements. (§5)

The students can write and test programs with conditional statements. (§6)

At this point, the students are familiar with the concept of parametrization. With the introduction of parameterized commands, however, they most likely also experienced a new source of errors: invalid arguments. For each parameter, there is usually a limited range of values that are considered valid while values outside of the spectrum lead to errors. For some commands, the spectrum of valid arguments is large, such as the command fd which accepts any floating point number or integer. Other commands have a much more narrow spectrum of valid arguments, such as the command repeat which only accepts positive integers. Command setpc rejects all values smaller than 0 or greater than 16. Procedure invocations that do not respect these rules result in runtime errors, as shown in Listing 3.5. At this point, students most likely got used to these kinds of errors from a user’s
How to Integrate Logo Into the Swiss School System

Listing 3.5 Most Logo commands which take one or several arguments may cause runtime errors if the respective argument does not lie within a spectrum of accepted values. The command \texttt{setpc} takes one numerical input that must lie within the range between 0 and 16. Passing in any value that lies outside the pre-defined spectrum results in a corresponding runtime error.

\begin{verbatim}
setpc 20 The specified color does not exist.
\end{verbatim}

Listing 3.6 Using the conditional statement \texttt{if} it is possible to react to different input values and reject those values which are invalid. In case of our \texttt{stairs} program, the number of steps cannot be negative and if a user was to provide a negative number, we rather return an error message than trying to execute the program.

\begin{verbatim}
to stairs :steps :height :width
  if (:steps < 0)
    [print [ I cannot draw a staircase with less than 0 steps. ] ]
    [ repeat :steps [ fd :height rt 90 fd :width lt 90] ]
end
\end{verbatim}

Listing 3.6 Using the conditional statement \texttt{if} it is possible to react to different input values and reject those values which are invalid. In case of our \texttt{stairs} program, the number of steps cannot be negative and if a user was to provide a negative number, we rather return an error message than trying to execute the program.

Parameters used in custom-built commands oftentimes come with implicit assumptions. The program in Listing 3.4, for instance, makes the implicit assumption that the number of steps must always be positive. Hence, if the first argument for \texttt{stairs} is negative, we want to issue an error message. This effect can be achieved by the programmers themselves, using the conditional statement \texttt{if}. Listing 3.6 shows how the \texttt{stairs} program can be extended to check whether the input for \texttt{:steps} is valid or not. If the argument is negative, an error message is printed, as shown in Listing 3.7.

With this technique, programmers finally learn to create custom commands that are purposefully built and resilient against user errors. To this end, they need to reflect what spectrum a parameter value may lie in and extend their programs to strategically reject values that do not conform their specification. With this, students are one step closer to building, maintaining, and extending their own programming language with new modules that can be used both by the students themselves and by other programmers. Over time, students learn to systematically test their code and include edge cases in their testing strategies.

At this point in the curriculum, the children have developed a solid understanding of concepts such as repetition, modularity, parametrization, and conditionals. When taught in the first cycle, these
3.3 Teaching Programming Across Different Subjects

Listing 3.7 Extending a program with suitable error detection mechanisms allows programmers to define new procedures that are save to use under all circumstances. Even if the program is provided with invalid input, the code provides useful feedback to the user.

concepts cannot be taught on their own but they must be embedded into other subjects. The next section presents an approach of how such an interdisciplinary start might look like.

3.3 Teaching Programming Across Different Subjects

As explained in Section 3.1, an interdisciplinary integration of computer science into the overall curriculum is explicitely desired by Lehrplan 21 [12]. In this section, we show that both mathematics and languages offer the possibility for cross-curricular activities using turtle graphics and Logo. Our examples are dedicated to the first cycle but there are more examples of how Logo enriches mathematics and languages in higher grades, too [97, 75, 99].

3.3.1 Logo in Mathematics Classes

Logo and mathematics went hand in hand from the very beginning since one of Papert’s main goals in developing Logo was to provide a natural and authentic environment to learn mathematics. Papert claimed that mathematics does not necessarily need to be the abstract and difficult subject as which it is often perceived. Instead, learning mathematics can be a natural experience similar to the immersive approach of learning a foreign language by visiting a country and conversing with people in their native tongue. Papert summarizes this idea in a metaphor of “Mathland” [144]:

“The idea of ‘talking mathematics’ to a computer can be generalized to a view of learning mathematics in ‘Mathland’; that is to say, in a context which is to learning mathematics what living in France is to learning French.”

The essence of this idea is that computers, and especially the Logo turtle, are tangible objects which naturally create a mathematical context for programmers to work in. Teaching students to program a computer gives them an incentive to express their ideas in a mathematically precise way and, by doing so, they can gain valuable first-hand experiences on abstract mathematical ideas that
become more concrete using a computer. The mathematics curriculum for the first cycle in Lehrplan 21 requires that young children learn to operate with numbers and count up to 20 elements. They are supposed to learn how to compare numbers, recognize numbers in images, and associate numbers to groups of up to five elements without counting consciously.

In this section, we present four examples of how turtle graphics can support young learners in developing rich mathematical models related to counting and addition. The four tasks are sorted by the increasing difficulty and they rely on different problem solving strategies that can be explored by students.

**Task 1: How many strawberries will the turtle collect?**

One of the first activities a young programmer starts with is to experiment with the new vocabulary provided by a programming language. For this purpose, the first two tasks we present in this sequence (Figures 3.8 and 3.9) are concerned with students acquiring basic programming literacy. That is, programmers need to learn to (i) read and interpret a given program and (ii) write programs to achieve a given goal. In doing so, children learn to communicate in a precise language and use it for their own purposes. Figure 3.8 is aimed at teaching students to read Logo programs while also making them count the number of elements stored on a specific field. Student activity is geared towards two learning goals, one related to programming, the other to mathematics:

- **Programming:** Students are provided with a Logo program that is ready for execution. The program consists of four commands which will be executed in the given order, from left to right. Novices need to learn how such a written program is executed and perform these steps mentally. They perform a step-by-step execution and make a prediction as to what path the turtle will take. Then, in a second step, they test whether their prediction was accurate by running the program and comparing the result to their expectation.

- **Mathematics:** This task does not only fulfill learning goals related to programming but it also allows students to practice basic mathematics skills. Specifically in this task, we involve the concept of counting, which students can experience in a playful surrounding. The task at hand provides programmers with a grid of fields that contain up to four strawberries. Once the turtle arrives on a certain field, all strawberries are collected. In this example, students first establish
3.3 Teaching Programming Across Different Subjects

3.3.1 Teaching Programming Across Different Subjects

3.3.1.1 In this exercise, programmers are given a concrete situation through a grid of fields in which a turtle can be navigated. In addition, they are given a concrete program that is supposed to be executed along with the question how many strawberries the turtle will find at the end of its path.

A mental model of how the execution will progress from the given starting point and only then they are able to answer the question at hand, namely how many strawberries the turtle will find.

The difficulty in this task consists in interpreting the program correctly and in predicting the path of the turtle. Once children predicted where the turtle will go, they count the number of strawberries on the respective field and finally check their assumption by running the program.

**Task 2: How to find X strawberries?**

In the task shown in Figure 3.9, learners continue to acquire the basic skills related to programming literacy – this time, however, by writing programs that serve a specific purpose. Similar to the previous task, this exercise is about collecting a certain number of strawberries in a grid of fields and once again there are fields containing up to four strawberries ready for collection. The task here is to write a program that will navigate the turtle to a specific cell which contains exactly the required number of strawberries. Writing such a program can be done either on a command-by-command basis or...
by writing the complete program at the start and testing its correctness afterwards by running the program.

In this task, young programmers have the chance to come up with their own solution and implement it in a program. Different students may come up with different solutions, because there are many different ways to reach a certain cell (e.g., both \texttt{fd 1 \texttt{lt fd}} and \texttt{fd \texttt{rt bk}} are legitimate solutions). While this task offers partial freedom in the choice of commands, the destination field is set. This very attribute will be relaxed in the next task and thereby we open up new opportunities for mathematical realizations.

**Task 3: How to collect X strawberries (part 1)?**

In the third task (shown in Figure 3.10), students tackle an exercise closely related to the previous one; again they face the challenge of collecting a given number of strawberries within a grid of cells and, again, there are cells with up to four strawberries. One difference, however, consists in the number of objects that the students are required to collect. Task 2 was solved by visiting one single cell which contained the exact number of strawberries required by the task. This time, the required number of strawberries exceeds 4 which means that programmers must accumulate strawberries from multiple fields. This aspect poses interesting new challenges both from a programming and mathematics perspective:

- **Mathematics:** A student’s first activity consists of figuring out how the desired total of six strawberries can be decomposed into several summands. The task offers four summands (i.e., 1, 2, 3, and 4), which can be combined with one another in any desired way. From two, three, or even four summands, a multitude of different values can be generated, while certain values can be reached in multiple ways (e.g., the value 6 can be obtained as $2 + 4$ or $1 + 2 + 3$). In both cases, individual summands can be collected in any order, yielding a multitude of different ways to reach the value 6.

- **Programming:** After the mathematical groundwork has been done and the students know how to obtain the desired value by adding several summands, the next step is to implement an adequate solution. Again, there is a multitude of different ways to solve the problem. Unlike the previous task, however, several different fields qualify as a possible end point of the turtle’s journey. An important finding in this sequence is that all the individual summands can be
3.3 Teaching Programming Across Different Subjects

Collect exactly six strawberries.

Collect exactly nine strawberries.

Figure 3.10 In this task, students need to find a way to collect six strawberries. Since there is no single field that contains six strawberries, they need to combine multiple fields.

Figure 3.11 Students need to collect nine strawberries in a grid where all fields contain fruits. To solve the task, students need to plan ahead and reflect the local environment of all fields along the way.

reached independently of each other. The order in which the individual summands are visited is left open to the programmer.

One of the most relevant factors in this task is that all strawberry fields can be reached individually without having to pass over another strawberry field. In the next task, strawberry patches cannot be visited independently which creates new challenges.

Task 4: How to collect X strawberries (part 2)?

The fourth and last task in this series deals with an adapted version of task 3: Again, a certain number of strawberries must be collected in a grid of fields, each of which containing one to four strawberries. One of the main differences between this and the former task is that, unlike before, all grid cells contain a “summand” and thus programmers have to consider the local environment in the immediate vicinity of the turtle to solve the problem.

While it was previously possible to visit and collect individual summands independently, the same is no longer possible now. In consequence, although there are several different ways to obtain the value 9 from a subset of the summands 1, 1, 1, 2, 3, 3, 4, 4 (e.g., $4 + 4 + 1$ or $1 + 1 + 1 + 2 + 4$), not all of these possibilities can be implemented due to the local arrangement within the grid. This additional restriction limits the number of possible solutions and it encourages the learners to apply
an exhaustive search based on the immediate vicinity of the turtle. They repeatedly try one of the
neighboring fields as a candidate and continue the search or backtrack until they succeed. At the
end of this unit, children have seen different ways of counting objects and making subtotals from
summands. All four tasks promote certain learning objectives that conform with Lehrplan 21, both in
regards to mathematics and computer science.

3.3.2 Logo in Language Classes

Mathematics and languages may not seem to share many commonalities at first glance but, in fact, the
two fields do indeed have at least one common interface that is provided through programming. That
is, programming shares both the precision required in a mathematical context and the perspective of
communication that is the purpose of using languages. Already in the 1980s, Papert mentioned these
two aspects [144]:

“[L]earning to communicate with a computer may change the way other learning takes
place. The computer can be a mathematics-speaking and an alphabetic-speaking entity.
We are learning how to make computers with which children love to communicate.”

Papert points out the role of a computer as being an entity that allows for natural and immersive
learning. He claims that a computer can change the way children learn in subjects other than
programming, such as mathematics and languages. An example of how language classes can profit
from programming was given by Papert himself: one of the earliest Logo school projects at Muzzey
Junior High School [170] involved programming activities such as writing a pig latin converter or
a sentence generator (see Section 2.3.1, Listing 2.18). While these examples provide insights into
how Papert imagined language classes to profit from programming, they are generally targeted at
lower secondary school students whose skills are more advanced than those expected in lower primary
school where children only just learned to read and write.

Lehrplan 21 lists a number of learning objectives in the domain of languages, some of which are
related to reading or writing. The learning goals demanded in these two areas are not limited to prose
texts only but they can also be applied to formal languages:
3.3 Teaching Programming Across Different Subjects

Reading Competencies          Writing Competencies

1. Acquire basic reading skills  1. Planning and searching for ideas
2. Reading technical texts (structure)  2. Formulating thoughts as words and sentences
3. Reading literary texts (content)  3. Revision of content and form
4. Reflection on the reading behavior  4. Reflection on the writing process

Both reading and writing competencies are not exclusive to natural languages but they constitute the usual processes of writing and reading computer programs as well (i.e., once basic programming literacy has been reached, programmers read and write texts with structure- and content-related ambitions, and they revise their texts and reflect their learning process). In this regard, reading and writing skills of students can be fostered not only by exploring natural languages but also by programming.

In the following four examples, we show how programming can be taught in the context of language classes and how these activities may support the reading or writing skills of young students.

Task 1: Which word will the turtle read?

The first exercise in this series involves learners to mentally execute a given program to find a hidden word in a grid of letters. The turtle is positioned in a grid of cells where each cell contains exactly one letter. With each step, the turtle navigates through the grid and collects all letters along the way. That is, every new letter adds an additional piece of information and all letters together finally make up a full word. It is crucial to consider the order in which individual letters are visited since they may not necessarily yield the same word otherwise. That is, although “LISTEN” and “SILENT” can both be constructed from the same set of letters, they are obviously not the same words.

Generally speaking, letters cannot be collected in an arbitrary order but they must be assembled in the exact order they are found within a word. A given program simply leads the turtle from one cell to another allowing it to collect letters along the way. Figure 3.12 shows an example where the word “SILENT” can be read but not the word “LISTEN”.

In this task, students get acquainted with reading a program and again, they learn how to translate a static piece of information into a dynamic movement which navigates the turtle through several cells.
How to Integrate Logo Into the Swiss School System

What word does the turtle read?
S I N
L E
T
...

Figure 3.12 This first exercise provides the programmer with a grid of letters and a program that is supposed to be executed. A novice programmer thus needs to interpret the code and figure out what path the turtle will take. Testing can be easily achieved by running the given program and simply comparing the result to the expected outcome.

Task 2: Find a given word

The second task in this series comes without a given program and instead asks the students to write a program themselves – with the goal of reading a specific word. Figure 3.13 shows an example where a grid hides the word “TURTLE” inside. In order to make the turtle read the given word, students first need to discover it themselves. In this case, the task poses an additional challenge since there are two plausible starting points. That is, in order to read the word “TURTLE”, the first step consists in reading the letter “T”. There are however multiple options, both to the right and to the left of the turtle. In order to figure out which of these two options is the correct one, the students focus on the second letter, and then the third. Only once they reach the fifth letter “L”, it becomes evident that the option to the left will not yield a successful result. Conversely, the path on the right can be continued and eventually does reveal the word “TURTLE”.

In this task, the students learn to write programs with specific purposes. In order to solve the given task, the first step consists in finding a path that actually reveals the correct word. Then, they express their thought in a program which finally can be run and tested using a computer.
3.3 Teaching Programming Across Different Subjects

What is the turtle’s favorite dessert?

M I C
A E
E R C

How many ways are there to write the word ‘KAJAK’?

A J A
K A J
A
K A

Figure 3.14 Students need to identify a word that matches the broad category of desserts without knowing which exact term to look out for. Find the word “ICECREAM” requires careful planning and testing.

Figure 3.15 This fourth and last task in our series is the most involved one since it requires the students to systematically test and enumerate different paths in the grid, always searching for the same term “KAJAK”.

Task 3: Find an unknown word with the turtle.

The third task in this series is closely related to the previous one with both of them featuring a hidden word that the students are required to find in a grid of letters. The main difference between the two tasks consists in the availability or respective lack of information about which word to search for. That is, while task 2 provided the students with the information to search for the term “TURTLE”, the only clue they are given in this task is a rough category to which the search term belongs (e.g., in Figure 3.14, the grid contains a word that matches the category of desserts).

Knowing which term to look out for makes the search process significantly easier. If the word is provided, learners can simply try to match up each letter in the given word with a cell in the grid. Without this information, on the other hand, learners have to explore each possible path and check whether they find a meaningful word that matches the search category. Only once they succeeded, students can start to describe the path in the Logo language.

Task 4: How many paths are there?

While task 3 already made the students search the grid exhaustively and test various combinations unconsciously, this task makes them do the same step in full consciousness. Again, they are provided with a grid of letters and a specific search term. This time, however, the given term can be reached through various different paths and the task requires the students to find out how many paths there are. The grid in Figure 3.15, for example, allows the word “KAJAK” to be read in more than ten
different ways. In order to solve this task, the students thus need to find a systematic strategy to test and enumerate different paths.

In this and the previous section, we have shown eight examples of how mathematics and language classes can be combined with Logo programming in accordance with the learning objectives of Lehrplan 21. Next, we will demonstrate how XLogoOnline allows students to come up with their own tasks in order to support student-centered learning.

3.4 Student-Centered Learning and Teaching in Logo

The constructivist movement changed the role of teachers dramatically over the past century [119, 174]. During the pre-constructivist era teachers used to be seen primarily in the role of an instructor, that is, as a leading figure who stands in front of the class and shares his or her knowledge with the students and provide practice and drill exercises to learners [20]. Through the changes induced by constructivists such as Dewey, Vygotsky, or Piaget, the role of teachers shifted towards being a facilitator rather than an instructor [124].

A direct implication of this shift was that the constructivist community of educators gradually moved away from teacher-centered instruction [164] and towards student-centered instruction [39]. That is, in virtually all fields constructivist teachers started to study their students’ learning, tried to understand how students get along with new content, and what common misunderstandings are typical in a certain domain. A new viewpoint was established that demanded instruction to take into account the individual knowledge a student brings along and to allow students to spend time actively engaged in constructive learning activities. With the learner at the center of the focus, the gravity of individuality between learners and joint learning became evident [58], which has proven to be a challenges teachers face on a daily basis and with students of all socio-economic backgrounds [186, 48]. Heterogeneity in learning is acknowledged across all ages and around the word [54].

While this new form of teaching means more individualized learning for students, it also presents many teachers with challenges, since school classes are typically heterogeneous groups of learners which come with vastly different backgrounds and therefore exhibit great differences in their learning. Some students are fast and master the prepared material effortlessly in a minimum of time. Others are severely challenged, take significantly longer to understand and often require more personal support. To deal with such heterogeneity is at the core of a pedagogical tool called “learning by teaching” [74].
This pedagogy is trying to make students step into the role of a teacher, come up with their own exercises, and challenge each other. Students who learn to explain materials to others have been proven to outperform their peers [69, 68] which signifies the advantages of this approach.

Over the last fifty years, the idea behind learning by teaching has been tested in numerous subjects, levels, and contexts [132, 122, 187]. Just like any other course, computer science and programming lessons are characterized by heterogeneous levels of prior knowledge and there is evidence that this prior knowledge does have an impact on student performance [83]. Most programming problems can be solved in different ways and thus teachers often find themselves confronted with different student solutions, some flawed, others correct. Logo programming allows students to interact directly with a computer and actively construct new knowledge [141].

“Learning by teaching” offers interesting opportunities in the context of Logo programming with turtle graphics. Logo is well-suited for auto-verification since all results are visual and students can easily compare their result with the intended outcome [96]. However, it can still happen that some students run out of tasks while others are still working hard on achieving the base line. For this purpose, we developed a tool that allows both students and teachers to create and share tasks. The toolkit allows the development of tasks that contain elements from four categories: (1) counting strawberries, (2) writing letters and words, (3) recognizing colors and shapes, and (4) building labyrinths. The user interface allows students to create a continuous grid of cells of any shape and size. In Figure 3.19, for example, the grid counts seven columns and two rows where some neighboring cells are separated by a wall. In the task creation tool, there are a number of elements available to be chosen and dropped into the grid (see yellow box on the right of Figure 3.16). Once shared, the task is ready to be solved by anyone. With this tool, both students and teachers can create a large variety of different tasks, such as the interdisciplinary use cases shown in Sections 3.3.1 and 3.3.2.

In the following section, we will present how the user interface of XLogoOnline is adapted to fit the different constraints and needs exhibited by children aged 5 to 12 years.

### 3.5 Age-Appropriate Input Methods

More than fifty years, the community of computer science educators and the research community have studied different ways to motivate young people to join the field of computer science. Along the way, various different tools, curricula, and metaphors were proposed to ease an age-appropriate start...
How to Integrate Logo Into the Swiss School System

Figure 3.16 XLogoOnline allows users to create and share their own custom-built exercises which contain among others elements from the following list: fruits, colors, shapes, and letters. On the left, the task creation view is visible where users can change and adapt their exercises. Once shared, the exercise is shown just like in the picture on the right where an exercise can no longer be changed.

for novices. Both Papert’s Turtle [144] and the puzzle-style blocks used nowadays [189] have proven to be useful learning metaphors for novices. Around the globe, computer science has been introduced as a new school subject and the core of this new subject is widely agreed to lie in the concept of computational [193] or algorithmic thinking [111, 79] which are often used as synonyms [175, 98]. Next to the various definitions for this term, several taxonomies [190, 165, 162] have been defined, based on which researchers have proposed learning activities [117].

In the context of programming, a large body of research has been conducted in order to find out what syntactic, semantic, and conceptional problems novices run into. Plenty of research exists for popular languages such as Java [42] or Python [114]. For several languages, there have been dedicated programming environments that have been developed specifically for the purpose of providing support to novices. Such environments exist for both imperative languages such as Python [116, 179, 24] and functional languages such as Scheme [66, 67]. The approaches chosen by these environments differs – some (like TigerJython or Thonny) focus mainly on providing dedicated support in case of errors whereas other environments (such as DrScheme and Hedy) gradually refine the scope of their environments to fit the learners skills [91, 67]. This thesis presents an environment that combines both of these aspects in the context of the Logo programming language.

The curriculum presented in Section 3.2 follows a spiral approach, where, at the beginning, programmers are given only a small handful of commands that can be executed one-by-one. Students learn to combine several commands into longer programs and to shorten such program sequences by means of repetition. Later, they realize that a programming language is not a static construct and that it is both possible and advisable to extend the language with custom commands. Finally, the circle
closes when students learn to extend the versatility of their self-defined commands by introducing parameters.

In order to implement this spiral curriculum in an age-appropriate manner, it is crucial to keep in mind that young children are likely to experience difficulties both in terms of physical and cognitive hurdles. For instance, the use of a computer keyboard is far less intuitive for kindergartners than it is for those at the end of primary school. Moreover, at each step along the way, students may also face misconceptions and make errors. With these considerations, we have developed a programming environment that allows students of all ages between kindergarten and sixth grade to immerse themselves in the world of programming and learn to cope with errors in an age-appropriate introduction to programming.

We consider an age-appropriate introduction to programming as an approach that adapts to the abilities of children throughout their education. In the first phase, we can not assume students to know how to read and write which must be taken in account by a programming environment (e.g., by providing a block-based interface). At the other end of the spectrum, students handle written text much more easily and they also need to be prepared for programming classes in secondary school, where they might encounter another programming language such as Python. For this, they need to know how to cope with errors and acquire a minimum level of syntactic precision. We provide students with three environments (see Figure 3.17) that are adapted to novice programmers ages 5 to 12:

- **Level 1**: The first step in programming lies in becoming familiar with basic commands which can be assembled to longer programs after a while. Our block-based user interface contains only a small number of unparametrized commands which provide a fully symbolic description to programmers which would otherwise struggle with reading or typing.

- **Level 2**: In the second step, the possibility of parameterization is offered directly in the available basic commands. Compared to the previous stage, children now learn to describe programs with arbitrary distances and angles, still using a block-based interface.

- **Level 3**: In the third step, students make their first experiences with a text-based environment which opens up the door for new syntactic errors and increases the students’ awareness of the precision required in programming. Meanwhile, there are also several new concepts reaching from custom procedures to parametrization and conditional execution.
How to Integrate Logo Into the Swiss School System

Figure 3.17 XLogoOnline introduces novices to Logo programming in three phases: First, they use a block-based environment that does not allow for any parameters to be used. Second, all basic commands are parameterized but the overall environment is still block-based. Finally, students experience text-based programming and start to define their own commands which can be parameterized as well.

These three environments are designed to pick up young programmers at the cognitive and motor level of their current development. From a basic first environment that does not require reading, writing, or even counting skills yet, the programmers deepen their understanding of parameters and modularity over several stages and finally develop a picture of what it means to communicate with a machine using a formal language. They encounter logical inaccuracies in their communication first, later they also become aware of structural inaccuracies and learn to analyze their programs, locate bugs and correct them autonomously.

All three stages provide a different interface to the same programming language. Each of the three “windows” into the Logo language allow programs with certain characteristics to be written. The three levels differ in terms of expressiveness and there exists a strict subset relation between them. Programs in level 1, for example, contain only six commands (fd, bk, rt, lt, setpc, and repeat), where both movement and rotation commands have fixed parameter values (i.e., 100 pixels steps and 90 degrees angles). A typical program at this stage may look like \texttt{fd 100 rt 90 fd 100 rt 90} or \texttt{repeat 4 [fd 100 rt 90]}; both of which only use the given commands and fixed parameter values. These same programs can be written in level 2, too, where the restriction to fixed parameter values is lifted. In addition to programs such as those mentioned above, level 2 also allows programs to be written with any parameter values as in \texttt{fd 20 rt 75 fd 75 rt 85 or repeat 8 [fd 125 rt 45]}. The same programs can be described in level 3, which introduces additional commands like \texttt{pu} and \texttt{pd}, as well as custom commands. The resulting programs may look like \texttt{fd 120 pu fd 120 pd or rt 30 triangle} which could not be described in any of the prior levels.

Since there is a strict subset relation between all three levels, we were able to craft a single execution pipeline for all three environments. The overall process is illustrated in Figure 3.19 and will be explained in more depth in the following chapters.
3.5 Age-Appropriate Input Methods

Figure 3.18 The set of programs that can be written in stages 1, 2, and 3 are in a subset relation to one another. Stage 1 is the most restrictive and allows only a small subset of all programs that are possible in stage 2. The programs written in stage 2 are again a subset of all programs possible in stage 3.

Figure 3.19 XLogoOnline provides three interfaces to students of different ages. The first stage provides only a small number of commands that are given as blocks. None of the four basic movement and rotation commands take any arguments. In the second stage, all basic commands can be parameterized. All commands in stages 1 and 2 are provided through a block-based interface. In the third stage finally, students can define their own commands and even learn to parameterize these commands. Programs are written as text, which means that students need to pay additional attention to spelling and syntax.
Chapter 4

Program Translation and Handling

Structural Errors

XLogoOnline is a web-based programming environment that translates Logo programs to JavaScript which in turn is compiled to executable bytecode using a just-in-time compiler (JIT). In this chapter, we focus on the first part of this translation process that begins with the user input (that is raw text) and ends in a structured intermediate representation as a parse tree. In this chapter, we provide insights into the first half of XLogoOnline’s execution pipeline which focuses on grammar and parsing. A special focus is put on error detection. We discuss how structural programming errors can be detected, located, and reported at compile time.

4.1 A Brief Overview – From Raw Text to Parse Tree

A programming environment represents a point of intersection between the programmer and the machine. From the one side it receives input from a human, in a yet unprocessed form as linearized text. On the other side, the environment is supposed to pass a fully interpreted series of instructions down to a machine, ready for execution. This process involves a number of sophisticated steps that XLogoOnline unfolds at the simple stroke of the <enter> key.

For a computer, interpreting a piece of program code requires knowledge about the language in which the text was written in as well as said language’s grammar, syntax, and semantics. Without knowing the grammatical attributes of a given program text, computers cannot retrieve the text’s underlying structural properties which is an essential precondition for automated program execution.
For instance, on a syntactical level the program `fd pow mod 5 4 3` simply consists of a sequence of characters that a lexer would identify as an arrangement of three words followed by three numbers. Without structural information about these syntactical elements (e.g., “How many arguments do each of these procedures take?” and “Do they return a result or not?”), it remains unknown how they relate. With a small piece of additional information, however, a computer is able to automatically distill the underlying structure of the given program text and represent it in a parse tree. Next, we discuss the details of lexing and parsing in Logo.

### 4.1.1 Lexer: Token Formation and Clustering

The first step in the parsing pipeline consists of the clustering of multiple consecutive characters to meaningful words or so-called *tokens*. This step is called *lexing* and the corresponding program unit is called a *lexer*. Like in most other languages, Logo’s tokens are formed mostly from letters and digits. One or several such characters can be combined to form new tokens such as keywords, constants, operators, strings, numbers, and identifiers. Some of these token classes (e.g., keywords and constants) form a *finite set* of elements. That is, there is only a limited number of elements that are used within certain categories such as `make` and `if` in the class of keywords and `green`, `pi`, or `black` for constants. Other token classes (e.g., identifiers, numbers, and strings) form *infinite sets*.

In order to establish automated token detection, each token class must be described in a formal way, usually in a *grammar*. For finite token sets such as keywords this task can be achieved simply by listing all elements individually. This is the case for instance in Listing 4.1 in the description of letters and digits (lines 2 and 3). For infinite token classes, lexer rules describe the general patterns that are shared by all members of a given token class. Line 17 in Listing 4.1 shows that the infinite class of identifiers can be described as a letter followed by any sequence of alphanumeric characters and underscores. This simple rule suffices to identify `my_house`, `HOLLYWOOD`, and `SchwuppDiWupp22` as identifiers.

Based on these rules, it is possible to write a lexer that turns any sequence of characters written in Logo into a sequence of tokens that carry meaning according to the Logo language specification. This process is illustrated in Listing 4.2 where the program `forward 100 rt 90.0 fd pi house` is run through a lexer which identifies individual tokens as defined in the grammar shown in Listing 4.1.

There are multiple options of how the language specification for Logo can be written and how a sequence of characters can be turned into tokens. Depending on which classes of tokens we define, a
/* tokens for individual letters and digits */
Letter : 'A'..'Z' | 'a'..'z' | 'å' | 'Å' | 'ö' | 'Ö' | 'ü' | 'Ü';

Digit : '0'..'9';

/* keywords for primitive statements */
FdStmt : 'fd' | 'forward';
RtStmt : 'rt' | 'right';

/* functions */
FuncNoArg : 'pi' | 'e';
FuncOneArg : 'abs' | 'sqrt' | 'random';
FuncTwoArg : 'mod' | 'pow';

/* tokens classes containing elements of arbitrary length */
Decimal : '0' | '0'..'9' Digit*;
Number : Decimal | Decimal '.' Decimal;
Identifier : Letter ( Letter | Digit | '_')*;

Listing 4.1 A simple grammar with lexer rules that describe a finite set of tokens and on lines 15 to 19 three lexer rules for infinite token classes of the Logo language.

# before lexing: sequence of characters
forward 100 rt 90.0 fd pi house

# after lexing: sequence of tokens
FdStmt Number RtStmt Number FdStmt FuncNoArg Identifier

Listing 4.2 A lexer turns a sequence of characters into a sequence of tokens. In order to do so, it tries to identify character clusters that comply with the rules defined in a grammar.

lexer may interpret the exact same sequence of characters differently. For instance, let us assume that the grammar is missing the two lexer rules in lines 6 and 7 which declare the Logo built-ins forward and right as linguistic keywords. Without these two rules, a lexer can no longer distinguish the two built-ins from general program identifiers. As shown in Listing 4.3, despite the change, a lexer is still able to interpret the input forward 100 rt 90.0 fd pi house and turn it into a sequence of tokens. This time, however, the resulting sequence is different and all tokens that were previously detected as built-in statements fd or rt, now simply map to a generic identifier.

Depending on choice, language designers can move linguistic elements between keywords and identifiers without affecting the overall quality of language detection.
Listing 4.3 Depending on which lexer rules are available, the result of the lexing process may differ. In this example, the same program is run through the lexer, this time however with a slightly adapted grammar that does not provide individual lexer rules for built-in commands such as \texttt{fd} and \texttt{rt}.

4.1.2 Parser: Retrieving Structural Properties

Once the lexer has finished its work, a sequence of tokens is passed on to the next stage in the parsing pipeline, namely the parser. The purpose of the parser is to construct a representation that reflects the input’s underlying structural properties as a parse tree. That is, after lexing, the program \texttt{fd pow mod 5 4 3} is interpreted as a sequence of six tokens. There is, however, no understanding yet of how these six tokens connect to one another. In order to break up the linear program structure, a formal specification of the structural properties of each token is required.

Besides lexer rules, a grammar typically contains a number of parser rules which provide the basis for parse tree construction. Instead of defining how characters can be clustered to meaningful tokens, parser rules define how tokens can be arranged to form syntactically-valid programs. For instance, the mentioned Logo program \texttt{fd pow mod 5 4 3} involves a built-in statement (\texttt{fd}) that requires one argument, as well as two function calls (\texttt{pow} and \texttt{mod}) which require two arguments each and which return a result that can be used as an argument by another command or function call (see Listing 4.4). Note that our notation distinguishes parser and lexer rules by their first letter: lexer rules start with an uppercase character whereas parser rules start with a lowercase character.

One of the parser rules has a special role which represents the root node of the resulting parse tree. We call this rule the entry point of parsing or root symbol. There are two common strategies to turn a sequence of tokens into a parse tree: (i) bottom-up parsing, which works from a token stream “upwards” by continuously using right-most derivations until the root symbol is reached; (ii) in top-down parsing on the other hand, we start at the root symbol and work “downwards” using left-most derivations. Our approach is based on a top-down parser.

A parser allows us to map deeply nested programs such as \texttt{fd pow mod 5 4 3} into a parse tree. Such a parse tree reveals the program’s underlying structure and makes it available to automated
4.2 Two Options to Define a Logo Grammar

Listing 4.4 While lexing, a sequence of characters is turned into a sequence of tokens that can be passed on to the parser later.

```plaintext
/* Root symbol: Starting point in top-down parsing */
input : (stmt | EOL)* EOF;

/* Remaining parser rules */
stmt : FdStmt expr | RtStmt expr;
expr : literal | FuncTwoArg expr expr | FuncOneArg expr | FuncNoArg;
literal : Number;
```

Figure 4.1 The final result of the parsing process is a parse tree. This data structure represents relevant structural properties of a given piece of code and is the basis for automated program execution which will be explained in more detail in Chapter 5.

execution. Using the grammar shown in Listing 4.4, it is possible to translate the above Logo program to the parse tree shown in Figure 4.1.

There are several ways to specify the Logo grammar - two variants differ in whether built-in statements like `fd` or `rt` are counted as keywords or as identifiers. This choice bears consequences that we will discuss in the next section.

4.2 Two Options to Define a Logo Grammar

Logo’s grammar can be described in various different ways – some approaches may follow a syntactically-stringent path that tries to include many structural properties within the grammar, resulting in a syntax-heavy grammar. Such a syntax-heavy grammar for Logo may include an individual rule for each built-in command. A second, less stringent approach treats built-in commands as
generic identifiers and results in a *syntax-light grammar* which consequentially also has an impact on error detection. In the following three subsections we first discuss why the distinction between keywords and identifiers matters and then we present two cases where once, built-ins are treated as keywords and once they are treated as identifiers.

### 4.2.1 Keywords and Identifiers - Why Bother the Distinction?

Some of Logo’s token classes show rather large differences in terms of their syntax, while others do not. A number, for instance, is defined as a sequence of digits whereas a string is defined simply as a sequence of (mostly alphanumeric) characters surrounded by brackets – between these two token classes there are no common elements. Keywords and identifiers, on the other hand, share large similarities and the borders are more transient. For this thesis, we define keywords and identifiers as follows:

- **Keywords** represent special words that are reserved for dedicated purposes in the programming language. Some common examples include control flow elements such as `while` and `if` but also the terms used to indicate the beginning and end of procedure declarations, e.g., `to` and `end`. All of these keywords are identified as unique syntax elements by the parser and most of them have a dedicated purpose while parsing. Another characterizing feature of this token class is the fact that it is a finite set and thus contains only a limited number of elements.

- **Identifiers** provide a generic vessel for any named entities (i.e., procedure and variable names). In contrast to keywords, which form a finite set of terms, the set of identifiers is infinite due to its generic form. All it takes for a term to fit as an identifier is that it must begin with a letter which is followed by any number of alphanumeric characters and underscores. This condition is met by a large number of terms including all keywords.

Most programming languages do not allow keywords to be used as identifiers [8, 9]. In Listings 4.5 and 4.6 we show four Logo snippets that illustrate the reason why this is the case in Logo, too. Four Logo programs are depicted, each using a keyword as an identifier. None of these programs can be parsed correctly which is due to the keywords `to`, `end`, `if`, and `repeat` being used in an unintended context. If any of these keywords is allowed to be used as an identifier, programs of this form result in parsing ambiguities.
4.2 Two Options to Define a Logo Grammar

Most Logo dialects do not allow keywords like `to` and `if` to be used as identifiers for program names due to syntactic ambiguities.

```
Listing 4.5

Listing 4.6
```

Listing 4.6 The same arguments holds for variable names: keywords like `repeat` and `end` cannot be used as variable name identifiers.

Parser rules such as the ones shown in Listing 4.7 provide a specification of the four keywords `if`, `repeat`, `to`, and `end`. The first keyword `if` must be followed by a comparison and a statement block in order for the program to be considered syntactically valid. None of the four examples in Listings 4.5 and 4.6 matches this specification. By refusing keywords to be used as program or variable identifiers, we solve problems of this form.

Since keywords cannot be used as identifiers, we must choose whether built-in statements such as `fd` or `rt` should be treated as keywords or identifiers. The next two subsections present both alternatives.

4.2.2 Approach 1: Built-ins as Keywords

For each built-in command that is used as a keyword there is a respective entry in the grammar which defines the command’s syntactic attributes including (i) the program name and possible alternatives,

```
Listing 4.7
```

Listing 4.7 A grammar provides a specification of how keywords are supposed to be used.
Program Translation and Handling Structural Errors

Listing 4.8 Logo’s basic commands can be specified directly through the grammar. In this example, we see several instances of rules that represent built-in commands as keywords in the grammar. Some of them have two options for program names. Others take more than one argument or fewer, all of different types.

and (ii) the number of arguments required. All of these attributes are represented in a formal specification as parser rules; see Listing 4.8.

It is possible to automatically check the structural correctness of all built-in commands in a given program by simply introducing individual parser rules for each built-in command. In particular, the parser can detect whether a built-in command was provided with the expected number of arguments. That is, due to the syntactic specification of these elements, program calls with too many arguments (e.g., `fd 100 100`) or too few arguments (e.g., `fd`) fail during parsing.

Despite the chances and opportunities of detecting errors through the parser, this approach also has its limitations. In the next section, we dive deeper into the topic of error detection under the assumption that built-in statements are treated as keywords.

4.2.3 Approach 2: Built-ins as Identifiers

Built-in commands do not necessarily need to be treated as special linguistic keywords but they can just be treated as unspecified identifiers (equivalent to user-defined commands). While parsing,
4.2 Two Options to Define a Logo Grammar

Listing 4.9 An identifier is defined to be a letter followed by any sequence of alphanumerical characters and underscores. Identifiers can be used to define a new program name for a custom command or, if not specified otherwise, it can also be used to detect typical commands related to turtle graphics, such as movement and rotation commands.

words that qualified as keywords in the previous approach now fall into the general category of program identifiers which, in contrast to keywords, have wide-range coverage that includes numerous commands, both built-ins and custom ones.

Due to the missing grammar rules, neither the program name nor the number of arguments is clearly specified on a syntactical level. Instead, a procedure invocation is simply characterized by an identifier followed by any number of arguments, as shown in Listing 4.9. Independent of whether the command exists, all of the following instances are mapped into valid parse trees: \texttt{fd 100}, \texttt{ft 100}, \texttt{df 100} and even \texttt{fd100}.

Contrary to the previous approach where program calls could fail if they were given insufficient arguments, here the same program calls are parsed correctly regardless of their structural correctness. This means, that while before program calls like \texttt{fd 100} were parsed correctly only if they got the correct number of arguments (as requested by the grammar), now even program calls like \texttt{fd 100 100} or \texttt{fd} are reliably converted into a parse tree despite containing structural flaws. This is because there are no syntactical requirements available to the parser and hence it treats both built-ins and user-defined commands alike, as simple procedure invocations without any further details. Under these circumstances, structural flaws in built-in commands are not detected by the parser and cannot be considered “syntax”. Instead, most structural errors of this form are typically detected at runtime as part of the language semantics.

In the next section, we present an overview of what structural programming errors typically occur in Logo programming classes with novices.
4.3 Common Structural Errors in Novices’ Logo Programs

We want to build diagnostics that help novices to locate and resolve most structural defects in their Logo code on their own and, to this end, we were interested in first investigating what errors beginning programmers commonly make in the early stage of them learning a new programming language. In this section, we present four classes of structural errors that make up the majority of all structural errors in Logo.

4.3.1 The Majority of Structural Errors

To detect what errors beginning programmers make, we ran a preliminary user study with 180 primary school children, aged 11 and 12. They spent their very first two hours programming in our labs and, after a 5-minute theoretical introduction to Logo’s movement commands, they started solving exercises. We collected a trace of 5040 submissions and investigated what structural errors the programmers faced during their learning. Out of 6500 errors, we distilled four major categories of errors. Towards the beginning, students got acquainted with the platform and made deliberate errors in what we call exploration phase. Once this phase was over, students settled into programming but still made occasional errors, mostly around invalid procedure invocations, brackets, and program declaration.

1. Exploration Phase.

Children do not know what happens within a computer. They treat it as a magic black box and first need to discover that computers lack human intelligence by exploring its boundaries. A few examples in our data show how students explore a new programming language. These examples include:

1. Wordy and descriptive instructions (as if they were talking to a human).
2. Commands in their native language (forward 100 becomes geradeaus 100).
3. Misread characters that are substituted by optically similar ones (e.g., using the letter ‘O’ in fd 100 rather than real digits: fd 100).
4. Entirely random strings by mashing on the keyboard (hvgfghfzffzugo8zt7).

Programming in a language with formal syntax and rigorous structure requires precision. The experience students gain during this initial phase is therefore valuable for building an intuition and demystifying how the language constructs work.
2. Procedure Invocation

By this stage, students have been exposed to several examples from a workbook but not yet been formally introduced to the concept of arguments and program design. We see this reflected in three types of errors that evolve around the concept of procedure invocation:

1. **Missing spaces:** Students struggle to grasp the difference between `fd 100` and `fd100`.
2. **Incorrect number of arguments:** Students provide too many or too few arguments as in: ‘`fd`’ or ‘`fd 100 100`’.
3. **Typos:** Most of Logo’s keywords are based on English terms like *forward* or *repeat*. Our target group is at an early stage of learning English and so we noticed many incorrect spelling variations. For example: `repead`, `repat`, `reapeat`, `repaet`, and `repet` (sorted in descending frequency) were all common misspellings of the keyword *repeat*. Students tried several alternatives, while a simple inline hint would have sufficed to put them back on track.

These three types of errors tie back to the broader topic of parameterized commands and program decomposition, which students will be exposed to later in the curriculum. With more expertise, the rate of these errors should decrease; however, locating these subtle defects remains notoriously hard and makes them worthy candidates for our diagnosis tool.


Logo uses matching pairs of square brackets to unambiguously identify the beginning and end of code blocks, for example in repetition and conditionals. In reality, children produce code where one or both brackets are missing, as in `repeat 4 [fd 100`, or they choose the wrong kind of bracket: `repeat 4 (rt 90)`. This is a common error which becomes increasingly difficult to resolve as students start nesting multiple code blocks.

4. Program Declaration.

Once students are acquainted with both basic commands and repetition, they learn to design new programs modularly. That is, they assign names to sequences of commands, and thus introduce new purpose-built commands. Errors *within the bodies* of such commands or related to their invocation fall back into the class of faulty procedure invocations. Not all errors, however, occur in the body or the invocation of such commands. Flawed specifications of program signatures (e.g., `to abc ... end`) lead to five new categories of structural errors that novice Logo programmers experience in their programming classes:
1. Missing or extraneous keywords to or end (to to abc end).
2. Misspelled keywords to or end (to abc emd).
3. Missing program name (to end).
4. Missing colon before parameters (to abc a :b end).
5. Missing parameter name (to abc : :b end).

If a program call cannot be matched to the signature of any user-defined command, the blame can be put either on the caller or the definition-side. Following common design conventions, we trust program declarations over invocation sites, which assumes that all program signatures are correctly specified.

Section 7.2 extends this study and provides quantitative measures for each of the most common error classes. The following two sections cover two methodologies that allow numerous error cases belonging to the above error classes to be detected at compile time.

4.4 Detecting Structural Errors – Syntax Checking

In this section, we delve deeper into the topic of error detection and answer the question of how a wide range of structural errors can be detected automatically, at compile time. This section is constrained to the assumption that built-ins statements are treated as keywords. We present our methodology for detecting structural errors and discuss the implications and limitations of the chosen approach.

4.4.1 A Methodology to Detect as Many Structural Errors as Possible

We explain how XLogoOnline was extended with an additional error checking mechanism that detects errors in Logo’s syntax. While pinpointing and reporting an arbitrary number of flaws in any given program, we also provide useful hints to novice programmers on how errors could be resolved. In the following, we first sketch the state of error detection we started from and then describe how adding parser rules allows to map even syntactically incorrect programs into a parse tree. Last, we explain how we detect structural errors in program calls by trying to match them against compatible program signatures.
4.4 Detecting Structural Errors – Syntax Checking

Figure 4.2 Example of a reduced grammar for a subset of Logo. The parser successfully turns a sequence of characters into a parse tree. Prior to our extension, the grammar only had ‘positive’ rules that match against valid syntactical elements.

**State of Error Detection Prior to our Modification**

Whenever a parser successfully builds a parse tree (see Figure 4.2), all underlying built-in language constructs are guaranteed to be free of syntactical defects.

In error cases like built-ins with missing or extraneous arguments, the parser cannot retrieve a program’s syntactical structure. Instead, it aborts and fails to generate a parse tree. Error detection is affected by this in two ways:

1. **Identifying multiple errors requires repeated parsing**: Every error causes the parser to stop. Thus, it is not possible to identify more than one error in a single pass through the code.

2. **Poor classification quality**: In ‘**repeat [rt 90] 4’** the expression 4 is provided at the wrong position. The parser aborts trying to match [rt 90] against a number. At that point, it is ignorant of the subsequent expression. Therefore, it cannot distinguish the example from ‘**repeat [rt 90]**’, where the expression is missing entirely. The two errors are fundamentally different and deserve to be classified individually.

**Extending the Grammar to Detect Failure States**

To overcome the above-mentioned issues, we extended the grammar with extra parser rules which cover typical error cases involving built-in commands, program declarations, and bracket errors. Additional rules allow the parser to successfully retrieve the syntactic structure, although a given program may be structurally flawed. Special markers are retained in the parse tree to denote the

<table>
<thead>
<tr>
<th>Lexer Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM: [0-9]+</td>
</tr>
<tr>
<td>STRING: [a-z][a-zA-Z0-9]*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parser Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog: cmd*</td>
</tr>
<tr>
<td>cmd: repeat</td>
</tr>
<tr>
<td>fd: 'fd' NUM</td>
</tr>
<tr>
<td>rt: 'rt' NUM</td>
</tr>
<tr>
<td>repeat: 'repeat' NUM block</td>
</tr>
<tr>
<td>block: [' cmd* ']</td>
</tr>
</tbody>
</table>

---

**Diagram**

```
Lexer
\[repeat 4 [fd 100 rt 90]\]

Parser
```

---

1. Character sequence
2. Token sequence
3. Parse tree
Program Translation and Handling Structural Errors

Figure 4.3 Parser rules are enhanced with additional error cases that flag syntactic errors while building the parse tree.

location of errors and are presented to the user at a later stage. Figure 4.3 shows a case where a defective program is parsed and the resulting parse tree contains a special marker at the respective error location. Note how the parser contains three rules related to the \texttt{fd} command, each flagged with an associated error case.

By adding extra rules for typical errors to our grammar, parse trees can be constructed despite them containing structural errors which ordinarily cause parse failures. An important point to note is how this approach allows us to successfully parse programs containing an arbitrary number of errors, without aborting and restarting the parsing process as long as the corresponding error has been flagged in the grammar. With this technique it is possible to detect instances of built-in commands that are provided with the wrong number of arguments, bracket errors, or faulty program declarations.

Like many other languages, Logo allows programmers to extend the language’s basic vocabulary by defining new commands. Detecting errors in invocations to such \textit{user-defined commands} is not possible simply by extending parser rules. There have been several proposals on how this problem could be solved from dynamic grammars\cite{157} to traditional program pre-processing. In the next section, we explain how collecting program signatures aids this challenge in the context of Logo programming.

\textbf{Collecting Program Signatures to Match Against}

A user-defined program is made up of a \textit{declaration} (i.e., signature) and a \textit{body}. By using the program name and parameter definitions we have a signature which fully specifies how the command expects
Listing 4.10 Invocations can be interpreted differently depending on surrounding context. In order to validate program calls, we also need to know all user-defined modules and their program signatures.

Figure 4.4 Identifying errors in program calls to user-defined modules. After building the parse tree, the program is indexed and procedure invocations are matched against available program signatures.
mechanism can be run while the student types, which allows on-the-fly program validation even before the program has been executed.

### 4.4.2 Linguistic Ambiguities and Their Implications on Grammar Design

Despite the great advantage of detecting any number of common errors with high reliability as early as parse time, the presented approach also suffers from some limitations – first and foremost a practical performance problem that boils down to the intentional introduction of grammatical ambiguities. Typically, syntactic ambiguities are caused by two or more grammar rules which all seem like legitimate options to the parser. Logo by itself already involves some linguistic ambiguities but the approach presented in this section adds even more grammatical ambiguities on top. In the following, we first motivate that ambiguities are a relevant topic in Logo and then highlight the undesired effect such ambiguities can have on the performance of a parser.

One example of Logo’s inherent linguistic ambiguities can be seen by studying the grammatical properties of the \texttt{print} statement. This command takes strings, Booleans, colors, numbers, or generic expressions as its argument. As shown in Listing 4.11, some of these types have striking similarities on a syntactic level, e.g., strings and RGB colors. Consequently, some situations involving strings and RGB colors result in uncertainty about whether the argument is supposed to be interpreted as the one type or the other. Depending on preference rules in the grammar, the statement \texttt{print [:a :b :c]} may print the string “:a :b :c” or a color (i.e., where :a, :b, and :c are interpreted as variable names for each of the three color components. The reason for this issue is that colors syntactically qualify as strings: an RGB color is defined as three numerical expressions surrounded by square brackets, while strings are defined more generally as any character sequence surrounded by square brackets. That is, while \texttt{[Hello!]}, \texttt{[fd 100]}, and \texttt{[0 0 0]} clearly qualify as strings only, the three text snippets \texttt{[0 0 0]}, \texttt{[1+1 2+2 3+3]}, or \texttt{[:a :b :c]} are inherently ambiguous and may be interpreted as members of either of the two types (i.e., Figure 4.5 shows how the statement \texttt{print [:a :b :c]} may be parsed differently depending on how this ambiguity is resolved in the grammar).

In order to provide deterministic behavior, we need to consider such issues while defining the grammar. Ambiguities must be resolved by choosing one of two ambiguous alternatives. In our case, this choice is expressed by always picking the first of all suitable alternatives in the grammar. That is, if the argument can be parsed as a string or as a color and the grammar lists the string alternative first, as in Listing 4.12, then the ambiguity is resolved by choosing the string option. If the color alternative
Listing 4.11 Due to syntactical similarities, it is not possible to distinguish between colors and strings in certain cases. Whether a linguistic element is interpreted as a string or as a color thus mainly depends on its context of use.

Figure 4.5 Parse tree for `print [:a :b :c]` where the argument is interpreted as a string or as a color.

Listing 4.12 In this grammar rule the ambiguity is resolved by prioritizing strings over colors in the `print` command.

Listing 4.13 In this grammar rule, when in doubt, arguments are interpreted as colors instead of strings.
decisions while still fully unaware of what tokens are to come next. This means that even unambiguous code may seem ambiguous to the parser until enough tokens are provided to dismiss the ambiguity. For instance, while \texttt{print [:a :b :c]} is syntactically ambiguous, as explained before, the closely related program \texttt{print [:a :b :c :d]} is not. While parsing the input on an element-by-element basis, the two streams are, however, indistinguishable from one another until the fifth token is read which finally resolves the ambiguity. The uncertainty involved in this grammar brings additional cost which is expressed by a concept called lookahead. Due to Logo’s syntax in RGB colors, its grammar can be described as an \textit{LL}(k) grammar with lookahead $k \geq 4$.

Besides linguistic ambiguities inherent to Logo’s syntax, additional ambiguities arise due to the presented approach which causes a larger lookahead. For instance, in order to detect whether \texttt{repeat} is missing a closing bracket, the presented approach proposed to extend the grammar with an additional rule that detects all repeat statements with missing brackets (see Listing 4.14). Rather than failing to parse the input in case of a missing bracket, the adapted grammar is capable of detecting repeat statements with a missing closing bracket. However, due to an overwhelming coherence between the two alternatives, the parser only knows which of the two rules to choose once it reaches the last token. Before this point, it is trapped in multiple alternatives. Note that (i) a block can contain any number of tokens which results in an arbitrary large lookahead and (ii) by nesting several repeat statements, the problem of ambiguity resolution increases.

We evaluated the performance impact of the presented approach by running a Logo program (see Appendix A) 100 times in a row through the lexing and parsing process, measuring the overall duration in each run. The same experiment was once run with error detection (e.g., with additional grammar rules) and once without. Due to known variations in JIT compilers over time, we ran the same experiment both in a cold state (i.e., just after startup) and in a warm state after having parsed the same program 100 times in a row prior to the first measurements. The experiment was run on a Lenovo T440p machine using Node.js version 10.4 and OpenSUSE Leap 15.0 as operating system.
Table 4.1 The same Logo program (involving all common Logo primitives and control structures have been parsed repeatedly to measure the impact of introducing error detection cases into the grammar. These rules show to have a significant impact, which causes a slowdown of a factor of roughly ten, both after startup and after many parsing steps.

<table>
<thead>
<tr>
<th>JIT</th>
<th>Grammar with error cases</th>
<th>Grammar without extra error cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>cold</td>
<td>0.827 seconds</td>
<td>0.085 seconds</td>
</tr>
<tr>
<td>warm</td>
<td>0.372 seconds</td>
<td>0.014 seconds</td>
</tr>
</tbody>
</table>

with 12 GB of RAM and an Intel(R) Core(TM) i7-4710MQ CPU. The results on Table 4.1 show the average duration over all measurements in each of the four cases. These results show that, in essence, the proposed approach can cause a tenfold slowdown of the parsing and lexing process.

In the following section, we present an alternative approach that treats built-ins as general program identifiers. This approach shifts the problem of error detection away from the parser and instead makes use of a separate semantic verification step that outperforms the previous approach.

### 4.5 Detecting Structural Errors – Semantic Verification

Treating built-ins as identifiers allows to reduce the number of syntactical elements that need to be specified in the grammar. Structural program properties (e.g., program name, number, and type of arguments) can be analyzed after parsing is complete in a so-called *static program analysis*. For this purpose, we infer structural attributes from all procedure calls which can then be matched against existing program signatures. This procedure works for built-ins and user-defined commands alike.

Both built-in statements such as `fd` and `rt` as well as user-defined commands such as `house` or `staircase` can be treated as the same syntactical element; a simple procedure invocation. In this approach, we do not distinguish between built-ins and custom commands syntactically which means that one single parser rule can be used to map procedure invocations of any form into a parse tree. The corresponding parser rule characterizes all identifiers which are followed by any number of arguments of any type (see Listing 4.15) as legitimate procedure calls.

This rule maps both built-ins and user-defined commands into the same kind of node and, more importantly, it does so irrespective of whether the given command is structurally correct or not. Besides structurally correct built-ins (e.g., `fd 100`) and custom commands (e.g., `house`), programs with misspelled program names (e.g., `forward 100`, `huose`, and `set pc blue`), missing or extraneous arguments (e.g., `fd 10 10`, `pu 100`, or `house 50`), and incorrect types (e.g., `fd blue` and `setpc`
Listing 4.15 On a syntactical level, all program calls can be combined in one simple grammar rule: an identifier followed by any number of arguments. Without additional keywords for built-ins, those “special” commands automatically fall into the same category as user-defined procedures and thus are mapped into only one kind of node in a parse tree, namely a program call statement.

false) are mapped into a parse tree, too. It is not guaranteed that such a parse tree could be traversed and interpreted successfully and thus, the structural correctness of a parse tree and must be verified before traversing it in order to prevent runtime errors.

Semantic program verification aims at testing a given parse tree for structural programming errors. This process is tailored to three main tasks: (1) detecting non-existing program names, (2) identifying program calls with too few or too many arguments, (3) making sure that all arguments have the correct type. In the following, we discuss each of these three points separately.

1. **Detecting typos in program names**

On a syntactic level, all program calls are treated as one single grammatical rule which corresponds to a program call in a parse tree. This mapping happens independently of whether the used program name actually exists or not. Semantic program verification is responsible for determining whether a given program name can be matched against an existing procedure or not. This can be achieved in the same way as before in Section 4.4.1 yet this time, the same technique must be applied to both built-ins and user-defined commands. In order to know whether a given procedure, say house or house, exists, the procedure name must be compared against all existing program declarations and built-ins. If a program call cannot be matched with any user-defined command and built-in, it is interpreted as a structural error and reported to the user (see Figure 4.6).

2. **Identifying program calls with too many or too few arguments**

Orthogonal to the approach presented in Section 4.4.1, program calls with too many or too few arguments can be detected using a pre-processing step in which the signature of all user-defined
4.5 Detecting Structural Errors – Semantic Verification

and built-ins procedures are collected. Then, each given procedure is matched against this collection to verify whether the provided number of arguments matches the specification.

3. Making sure all arguments have the correct type

Logo is a programming language with dynamic types. That is, all expressions are typed and all commands expect to receive their arguments from a certain type class. Without static type annotations, however, program execution can end up in a type error whenever a procedure argument does not fit the procedure’s expected argument type. In order to detect such problematic procedure calls at compile time, we developed a simple Logo type checker for built-ins. Moreover, we implemented a type inference mechanism that determines the type of variables (including parameters) just from their context of use. In the following, we discuss type checking and type inference separately.

• Type checking with built-ins: There are four basic classes of types in Logo: numbers, strings, colors, and Booleans. All built-in commands expect arguments of a certain type. If commands are not provided with the expected types, a program fails during execution although it has passed both of the previous tests: neither the program name nor the number of arguments is responsible for the crash. In order to detect type errors in calls to built-ins such as `fd green` or `setpc true`, we implemented a simple Logo type checker whose role it is to determine whether a given argument, say `true`, matches the type expectations of a given built-in command, say `setpc`.

Figure 4.6 Typos in the program name of a procedure invocation do not prevent the construction of a parse tree, however, the parse tree does not reflect the intended meaning (i.e., a link between the invocation of a program name and its declaration).
Figure 4.7 Procedure calls to built-in commands can be extended with type checks. In addition to the number of arguments, we verify whether the given arguments are members of any of the declared type classes expected by the procedure. This way, we are able to detect `setpc true` as typed incorrectly although both the program name and the number of arguments match the procedure specification.

In order to detect type errors in procedure calls to built-in commands, we extended the program specification of built-ins with internal type annotations. Note that some commands allow more than one type, such as `setpc`, which takes both colors and numbers or `print` which accepts arguments of all four types. This change in specification allows the previous verification process (step 2) to be extended with a simple type check. In addition to the number of arguments, we also check whether the given arguments can be interpreted as elements of the expected type classes (see Figure 4.7). With this mechanism, we are able to detect flawed program invocations with built-ins before the program is executed.

**Type inference with parameters and variables**

Arguments can be static values like 12, `orange`, or `true` but they can also be parameters or variables. In contrast to static values that do not change over time and whose correspondence to specific data types can be verified easily, variables can contain arbitrary values and can therefore be assigned to any data type. As a result, it is hard to tell whether a statement like `fd :x` is typed correctly or not. We typically do not have enough information to infer the type of a variable just from the local context of a single statement. Larger contexts (i.e., entire program declarations), on the other hand, often provide enough information to infer the type of a variable. For this purpose, we developed a type inference algorithm that follows parameters and local variables within the context.
4.5 Detecting Structural Errors – Semantic Verification

Listing 4.16 Three parameterized programs with the same specification with one parameter :x whose type is unknown in the beginning. Our type inference algorithm narrows down the type of :x in all three cases, assessing whether the requirements can still be met or not. The third example fails since :x cannot be a number and a Boolean simultaneously. We mark the last if statement as erroneous with an error message declaring that :x is expected to be a number.

The example in Listing 4.16 shows how the type of an initially unknown variable can be iteratively narrowed down and used to detect type errors. All three procedure declarations take one argument :x whose type is unknown in the beginning. By following the variable through the parse tree, we narrow down the possible types :x can have. The first example quickly determines that :x must be a number which is not violated throughout the procedure declaration. The second example first determines that :x must be either a number or a color and then, in a second step, narrows the possibilities down to numbers only. The third example ends up in a type error since :x cannot be a number and a Boolean simultaneously. The algorithm detects the first statement in which type information conflicts (that is why in the failing third example, the error is not raised in the fd command but in the if statement).

Logo allows a variable to take on different types in the course of its lifetime. With every assignment, a variable undergoes a restart with respect to its type inference. That is, with every make command the type inference algorithm resets the range of possible types for a given variable to unknown (i.e., number or color or Boolean or string). Listing 4.17 illustrates this point: The program on the left is an exact copy of the program that
Listing 4.17 Variable assignments cause the type inference algorithm to widen the range of possible types again, which is similar to a new start that is unrelated to a variable’s past.

Previously caused a type error (Listing 4.16c). The program on the right is identical except for one line of code (i.e., `make "x true`). This assignment enables a smooth transition from one type to another and prevents the type inference algorithm from triggering an error.

The proposed algorithm is designed for a specific use case with built-in commands and encapsulated variable checks. It has several limitations in the following areas: (i) branching, (ii) cross-procedural type inference, (iii) type checks for user-defined procedure calls, (iv) out-of-bounds checks for color numbers and RGB values, (v) fine-grained type distinction between integers and floats. Future work could comprise further research in any of these fields.

Thanks to semantic verification, a number of type errors can be detected at compile time and reported before execution starts. This circumstance allows the written code to be continuously tested for errors and the checks are fast enough to run in the background and on-the-fly.

Due to Logo’s linguistic characteristics and syntax, some errors simply cannot be detected automatically and pose inherent ambiguities. These instances will be illustrated in the next section.
4.6 The Limits of Automated Error Detection in Logo

Logo was purposefully designed as a programming language for novices. This was achieved, among other design principles, by choosing a simplified syntax. For instance, while Python uses commas to separate consecutive arguments and Lisp uses parentheses as statement separators, Logo does not need either of these two mechanisms. Instead, Logo uses whitespace separators between both arguments and statements. This choice, however, has its implications in terms of error handling. Specifically, due to the lack of another delimiter, reporting an accurate and precise reason of an error may not always be possible.

One instance of this problem can be seen in nested program calls with extraneous or missing arguments. In this case, we can end up in a situation where it is not possible to know which argument was intended for which command. For example, using the command \texttt{fd} (that takes one argument) and the command \texttt{mod} (that takes two arguments), we build the following ambiguous program, which has two interpretations: \texttt{fd mod 5 3 7}. Without explicit parentheses, the following two options are possible:

\begin{itemize}
  \item \texttt{a) fd (mod 5 3) 7}
  \item \texttt{b) fd (mod 5 3 7)}
\end{itemize}

Neither of the two interpretations is correct but depending on which of the two is chosen, the error will be associated with either the program invocation of \texttt{fd} or \texttt{mod}. That is, if the first option is chosen, \texttt{fd} is illegally given two arguments. If we pick the second option, it is the \texttt{mod} function that illegally receives three arguments rather than two. Without brackets, we cannot know which command the erroneous argument was meant for and therefore we react by reporting overly generic error messages. Under these circumstances, no error diagnosis mechanism can help at exactly pinpointing the error since the underlying problem is inherent to the language.

In the next chapter, we explain how a given Logo parse tree is executed and how XLogoOnline users can observe program execution both in physical and virtual runtime systems.
Chapter 5

Program Execution and Runtime Systems for Logo

Starting from a structured intermediate representation as a parse tree, it is possible to execute any Logo program automatically, no matter how complex and nested it may be. In this chapter, we discuss how a Logo program can be executed by traversing a given parse tree. We define a visitor that visits one node after the other and performs a set of operations on specific nodes. For turtle graphics applications, both physical and virtual runtime systems are possible; some visualize the result of a program using a virtual screen turtle while others make use of a physical floor robot. The presented approach is universal in that it can be applied to both virtual and physical runtime systems. Three examples of such environments have been implemented and will be presented at the end of this chapter.

5.1 Parse Tree Traversal

In the previous chapter, we have explained the first part of an execution pipeline that transforms any text written in the Logo programming language into an intermediate representation that is designed for automated execution, namely in a data structure called parse tree. In this section, we cover the second part of this execution pipeline and delve deeper into XLogoOnline’s interpretation and execution mechanism. We use a visitor that traverses parse trees and executes a set of actions along the way. The chosen procedure allows any parse tree to be traversed, including parts representing nested program calls, control flow structures, and local variables. We first illustrate how broad the spectrum of possible
A programming language like Logo allows its users to solve a given turtle graphics task in various different ways. Logo programmers are not only free to choose the starting point of their solution, but they also have the freedom to choose among different concepts and strategies to solve the problem. For instance, in order to make the turtle draw four adjacent squares (as shown in Figure 5.1), there are numerous different ways to go ahead. Listings 5.1 to 5.4 show six approaches that cannot be distinguished by their runtime behavior. All six programs draw the same lines in the same order from the same starting point. From the perspective of an external observer, the effect of each of these programs cannot be distinguished from that of the others. Despite their similarities at runtime, however, the programs obviously differ when it comes to the written program text. Each example uses a different set of programming concepts to solve the task and, not surprising, all of these differences are directly reflected in the corresponding parse trees. 

Figures 5.2 and 5.3 visualize two parse trees that correspond to two of the illustrated programs. When comparing the two parse trees, we see discrepancies in terms of their respective shape and size. The solution consisting only of built-in commands (i.e., Listing 5.1) produces a tree of small height...
5.1 Parse Tree Traversal

Listing 5.2 These three solutions reduce the overall length of the program using repetition, nested repetition, and repetition in combination with a sub-procedure.

Listing 5.3 This solution uses a while-loop with a counter.

Listing 5.4 This solution uses recursion, parameters, and conditional statements.

that contains a large number of nodes (i.e., Figure 5.2 shows a tree that contains 44 program calls to built-ins, which account for more than two hundred nodes in total). In contrast, the tree shown in Figure 5.3 is considerably smaller and counts only a fraction of the nodes used in the first example. It does so using two nodes that do not exist in the prior example, namely the nodes repeat and square, which represent control flow elements and user-defined procedure calls.

How is it possible that different parse trees result in the same outcome although their underlying structure differs so drastically? The answer to this question is rooted in the particular way a parse tree is traversed. While ordinary parse trees (i.e., without procedure calls and control flow elements) are typically traversed in a depth-first approach from left to right, specific language elements cause a more complicated traversal order. In the following subsections, we examine three cases: parse tree traversal with (i) basic commands only, (ii) procedure declarations and invocations, and (iii) control flow elements including repetition.
5.1.2 Basic Parse Tree Traversal

We start with Logo programs that consist of only basic commands and where each action of the turtle corresponds to one single basic command in the program text. Listing 5.1 contains 44 commands which, during execution, account for 44 actions that are performed by the turtle. This bidirectional relationship between turtle actions and the corresponding commands becomes evident once execution order is taken into account: all instructions are given in the exact same order as they are executed in. During execution, the corresponding parse tree (see Figure 5.2) is traversed in a fixed order: starting at the root node, the visitor visits vertices in a depth-first order from left to right (“left” and “right” are determined by the order of the corresponding tokens within the input text). That is, the visitor traverses each sub-tree one after the other, starting with the sub-tree that represents \texttt{fd 100}, then the one for \texttt{rt 90}, and so on. All edges in a sub-tree are visited twice: once on the way down (from the root towards the leaves), and once up (from the leaves back to the root). Along the way, relevant information is collected for later use, e.g., the argument value 100 for the basic command \texttt{fd}. Nodes labeled “\texttt{progCall}” are responsible for performing procedure calls to built-in or user-defined procedure calls. These calls are performed whenever the corresponding sub-trees have been visited. That is, once the visitor is on its way back to the root and all necessary arguments have been visited.

With this type of tree, hardly any state information is needed to navigate successfully from the beginning to the end of the program. The visitor always moves along the existing edges from parent to child or back. If a node has multiple children, they are all visited in the order as defined by the
5.1 Parse Tree Traversal

Figure 5.3 This parse tree corresponds to the code snippet shown in Listing 5.2. In comparison to the parse tree shown before, this tree is higher but less wide. The overall parse tree involves fewer nodes too. Note that in this visualization we still omit some nodes for readability reasons.

program code. In this example, it is sufficient to know which node is currently being visited as well as which of its children still need to be visited. Parse trees of this form can be traversed from the beginning to the end without additional state information.

5.1.3 Visiting Procedure Calls

Assume, a program `foo` is to be executed where `foo` is a user-defined procedure (see Listing 5.5). The corresponding parse tree for this procedure call is quite simple (Figure 5.4 on the left); it does not contain anything but a simple program call. Without the corresponding program declaration, however, it is not clear what executing `foo` exactly entails. So, in order to traverse parse trees that contain user-defined procedure calls like `foo`, the corresponding entry must be fetched from the available program declarations (Figure 5.4 on the right). Within the program declaration for the example `foo`, the program body `fd 100 bk 100` is found and can be executed subsequently as part of the program call. At runtime, the program declaration must be linked to the program call and traversed as part of the parse tree which is connected to the original “progCall” node.

When switching from one function scope to another via a procedure call (e.g., when transitioning from the global scope to the function scope of a specific procedure such as `foo`), a new stack frame is pushed onto the call stack for storing local variables. The same stack frame is removed again once the
Listing 5.5 During the execution of program call `foo` the visitor has to fetch the corresponding program declaration and execute its body.

Figure 5.4 In order to execute the program call `foo` from Listing 5.5 and correctly perform the two corresponding commands `fd 100 bk 100`, we must connect the parse tree representing the program declaration of `foo` to its call site.

the control flow is returned to the caller at the end of a procedure call. This is a common practice in most programming languages and enables the variable mappings to be kept separate for each procedure. As mentioned in Section 2.1.2, our implementation of Logo uses a flat variable binding which means that variables are not shared among different function scopes except as parameters. Listing 5.6 shows an example that indicates the impact of using a flat variable binding. The example uses a variable `:size` in two contexts, once within the caller (on the left), and once within the callee (on the right). The function scope on the left is independent from the function scope on the right and, after the procedure call to `bar` finishes, the caller’s stack frame is restored including all local variables. That is why the final `fd` command in Listing 5.6 uses the value 30 as an argument to `fd`, although `:size` is assigned the value 70 in `bar`.

If `:size` is passed on to `bar` as a parameter, as in Listing 5.7, a simple call-by-value scheme is applied. This means that a copy of `:size`’s current value is passed into the stack frame of `bar`. Once the procedure call ends, the caller’s function scope is restored with all of its local variables in their original form (including `:size`). This way, the proposed approach reduces the risk of unwanted side effects.
Listing 5.6 Variable mappings stay unaffected even if the callee changes the value of a variable in its private context. All variables are considered local and they do not exist outside of their context.

Listing 5.7 Even with parameters, the variable mappings stay unaffected once the control flow is returned back to the caller.

In the next section, we consider how parse trees can be traversed while containing control flow elements such as repeat. In contrast to the previous two cases, control flow elements can cause program parts to be traversed several times or not at all.

5.1.4 Visiting Control Flow Structures

One of the first concepts young programmers learn in the curriculum presented in Section 3.2 is the concept of repetition. The topic can be motivated easily, since even the simple shape of a circle requires a considerable amount of effort and diligence to write by hand when using basic commands only. It is tedious work even when using copy and paste. The example in Listing 5.8 illustrates this point: to draw a circle using only the basic commands fd and rt, programmers need to execute the two commands \texttt{fd 1 rt 1} a total of 360 times in succession. The program \texttt{repeat 360 [fd 1 rt 1]} in comparison is obviously much shorter to write.

As a result, the two corresponding parse trees show differences in terms of the number of nodes they contain. The example in Listing 5.8 on the left leads to a tree similar to Figure 5.2, with 720 sub-trees that are all directly attached to the root and contain nothing but program calls to \texttt{fd} and \texttt{rt}.
Listing 5.8 Two ways to create a circle: the left uses basic commands exclusively and is thus much more laborious than the approach on the right which uses the repeat construct.

Figure 5.5 This parse tree corresponds to the program `repeat 360 [fd 1 rt 1]`. The tree is considerably smaller than its counterpart which does not use a repeat statement.

commands. This tree contains more than 3600 nodes that each need to be traversed by a visitor. In contrast, the program on the right in Listing 5.8 leads to a much smaller parse tree, which is illustrated in Figure 5.5. This tree contains less than 30 nodes – a fraction of what was needed before – to achieve the same result.

Obviously, the two sub-trees representing `fd 1` and `rt 1` still must be visited 360 times in a row in order to achieve the same output as before. This time, however, the effect is created by visiting the same sub-tree repeatedly (namely the sub-tree related to the repeat statement’s block node). While previously no state information was required since all edges had to be visited exactly twice, this time we need a stateful approach that keeps track of the loop counter. This information is stored in all repeatStmt nodes and must be updated for each loop individually. That is, in case of nested loops, there are multiple loop counters which keep track of each loop’s iteration number individually. Figure 5.6 shows an example where two loop counters must be managed interally: The program `repeat 36 [repeat 360 [fd 1 rt 1] rt 10]` uses two nested repetitions to draw 36 circles.
5.2 Physical and Virtual Environments

XLogoOnline relies on a browser for lexing, parsing, program analysis, and execution. In order to make program execution more tangible, however, novices receive the option to run their code not only in a virtual setup (i.e., on a computer screen), but also on a physical device (e.g., external, educational floor robots). In the following subsections, we present three examples of physical and virtual program execution mechanisms that are available in XLogoOnline. We describe similarities and differences between the three systems and explain how these different devices are integrated into the XLogoOnline learning environment.

5.2.1 Virtual Execution

The Logo turtle became a popular symbol for one of the earliest computation models that was specifically designed for novice programmers. While most professional programming languages usually provide a plethora of features and full access to the computer’s underlying hardware resources, Logo and turtle graphics intentionally provide only a small and well-manageable number of basic
commands that form a beginner-friendly computation model. Depending on the available commands and their respective semantics, different turtle implementations provide different functionality domains that can be accessed by programmers. XLogoOnline’s virtual runtime system offers five basic functionality domains: turtle, pen, screen, eraser, and history. Each of these domains involve one or several state variables that can be directly manipulated using different commands (see Table 5.1).

<table>
<thead>
<tr>
<th>Domain</th>
<th>State Variables</th>
<th>Related Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turtle</td>
<td>Position, Orientation, Visibility</td>
<td>fd 100, rt 90, ht</td>
</tr>
<tr>
<td>Pen</td>
<td>Width, Color, Visibility</td>
<td>setpw 10, setpc red, pu</td>
</tr>
<tr>
<td>Screen</td>
<td>Color</td>
<td>setsc orange</td>
</tr>
<tr>
<td>Eraser</td>
<td>Mode (on/off)</td>
<td>pe</td>
</tr>
<tr>
<td>History</td>
<td>Text</td>
<td>print</td>
</tr>
</tbody>
</table>

Table 5.1 Logo uses state variables from four different domains: turtle, pen, screen, erasor, and history.

Each state variable is part of the overall program state which reflects the sum of all state attributes and their respective values, together with all local variables and parameters that are defined by the programmer.

Several of these functionality domains are related to drawing which signifies a core functionality in XLogoOnline’s virtual turtle graphics implementation. Young programmers use a canvas to create their own digital drawings. Since the early 2000s, there are two common techniques to represent graphics. The traditional approach uses simple bitmaps, whereas newer implementations often use vector-based graphics instead, even in Logo [27]. For reduced memory consumption and loss-less resizing, vector-based graphics are usually the preferred option which is also the case in XLogoOnline.

We chose this newer approach primarily due to memory restrictions in our debugger, where individual snapshots need to be stored.

5.2.2 Physical Execution

In addition to program execution in a virtual runtime system on screen, XLogoOnline also offers the option to run code on physical devices. Two floor robots are available which are both designed for educational purposes and whose functionality largely matches the Logo vocabulary. The two robots BlueBot and Root are used across different countries to teach programming to novices [176, 125]. Literature considers robots as a “multi-modal embodied learning opportunity” [105], which can boost the motivation of students [26]. Moreover, evidence suggests that robots are often used in
5.2 Physical and Virtual Environments

a constructivist/constructionist learning framework [107], which aligns with the goals of the Logo programming language. We show two implementations with different robots and demonstrate how Logo programs can be run on physical runtime systems that are steered from a web-based environment.

**BlueBot: A Robot for Pre-School and Lower-Grade Students**

Among the robots designed for educational purposes, there are only few that aim specifically at pre-school students while also focussing on the task of programming [176]. One of the common choices is a robot called BlueBot which is a small educational floor robot that can be navigated by pressing the buttons on its back. This robot has already been deployed in Swiss schools for several years[19, 90, 18, 188]. Tasks typically consist of simple navigation tasks. Four basic commands are available (two movement and two rotation commands), similar to the Logo turtle. The computational model used in these robots is closely related but not identical to the model used in our virtual system. In contrast to the screen turtle, each change of direction is bound to a fixed angle of 90 degrees and both movement commands are fixed to unit distances of 12cm per step. BlueBot can be modeled as a simplified version of the Logo turtle with a restricted command set consisting of four unparameterized commands. We integrated BlueBot into the web-based learning environment XLogoOnline (to learn about the communication protocol, see Appendix B) and allow it to be used as a physical turtle in activities such as those explained in Sections 3.2.1 to 3.2.3.

BlueBot uses unparameterized basic commands which signifies an inherent limitation of this device. In order to execute parameterized commands on a physical robot, a different device must be chosen. One such example will be presented next.

**Root Robot: Executing Programs with Parameters, Repetition, and Sub-Procedures**

BlueBot’s limited functionality poses a problem when it comes to drawing more sophisticated geometric patterns with different angles and lines of different distances. For patterns with more advanced features, an educational robot must match the semantics of the Logo vocabulary more closely. iRobot’s *Root robot*, formerly known as AERobot [163] is one such device. Just like BlueBot, it is Bluetooth- and BLE-capable, but its functionality overlaps the common turtle graphics commands more closely (e.g., Root allows movement and rotation commands to be parametrized). There are two main clusters of services that are supported by this robot:
1. The first cluster involves two services that are related to **motor control** and the broad domain of navigation. One service is used for forwards and backwards *movement* while another service is used for left and right *rotation*. In both cases, the robot allows the commands to be specified using arguments.

2. A second cluster provides a service to control the device’s **pen** and **eraser**. Unlike the virtual turtle, whose pen and erasor can be used independently, Root’s pen and eraser are linked. That is, there is one service to steer both and there are only three out of four positions possible: (i) pen down and eraser up, (ii) pen up and eraser down, (iii) pen up and eraser up. In contrast to the virtual turtle on the screen, the setting “pen down and eraser down” is not supported due to physical limitations.

Appendix C presents how XLogoOnline communicates with the robot and allows programmers to execute syntactically correct Logo programs on a physical device.
Chapter 6

Debugging

Logo novices write simple programs that draw geometric shapes onto a screen. Logical flaws, however, cause unintended results and pose a major challenge for young programmers who need to learn how to search for errors in their code. In this chapter, we discuss which logical problems novices face when learning to program in Logo. Moreover, we present a reverse (i.e., bidirectional) debugger that supports programmers who are searching for logical errors. Step by step, they navigate through their code until they identify the core of a problem. The presented approach balances performance and memory consumption and can be used to debug even long and complex programs.

6.1 Introduction

Learning how to cope with errors makes up a vital part of a programmer’s competence. In what follows, we discuss the struggles novices face when learning how to program, and we present a debugging tool that supports novices during the process of understanding logical errors and recovering from them.

6.1.1 Making Mistakes: A Matter of Attitude

Humans cannot help but make mistakes. In the context of programming, computer science pioneer Ada Lovelace held an exemplary attitude on this matter [92]: “I used once to regret these sort of errors, & to speak of time lost over them. But I have materially altered my mind on this subject. I often gain more from the discovery of a mistake of this sort, than from 10 acquisitions made at once & without any kind of difficulty.” Today, 180 years after this statement was made, much younger and
less experienced students also learn programming and are still confronted with their own weaknesses. We can only hope they adopt a similar attitude to errors as Ms. Lovelace when they face issues in their programs.

6.1.2 The Rift Between Expectation and Outcome

By programming, students learn to express their ideas in a formal language. The resulting programs are written with clear expectations of what should happen when executed. Due to errors, however, a program may behave differently than expected. Seeing how a program has a different outcome than expected often has a shocking and somewhat alienating effect on novice programmers. In order to fix their program, they need to learn how to debug. Seymour Papert said [144]: “When you learn to program a computer you almost never get it right the first time. Learning to be a master programmer is learning to become highly skilled at isolating and correcting ‘bugs’. [...] The question to ask about the program is not whether it is right or wrong but if it is fixable.” By learning how to fix incorrect programs, students become self-sufficient programmers: without external guidance, they write and refine programs with a specific purpose. Unfortunately, students’ reality often looks different.

6.1.3 Students Do Not Know How to Cope with Logical Errors

Without proper guidance, students are often lost when facing logical errors: They either start haphazardly tinkering with the code or they come to a full stop and wait for external support [123, 148]. In order to cope with such errors autonomously, students need two fundamental skills: first, they need to understand their program and be able to locate the issue(s) in their code. To find the needle in the haystack, carefully tracing the program logic one command at a time is a suitable strategy for Logo programmers [43]. There are several strategies that can be applied [136, 36] and novices have been shown to stick with debugging procedures they are used to [108]. We apply a modular approach to debugging that is inherently connected with modular program design as describe by Hromkovič et al [97]. Once an error has been located, a second skill comes into play: finding and implementing a solution. While the latter may depend on the programmer’s creativity, the first can easily be supported by automatic tools. Explicit teaching [136] and special debugging interventions [126] allow novices to better understand their erroneous code.
6.1.4 Structural Errors vs. Logical Errors

Programming errors belong in one of two classes: structural or logical errors. Structural errors cause the execution to fail (e.g. due to typos or missing arguments). This class of errors can be clustered into three categories, depending on the time when an error is detected: (i) syntactical errors occur while parsing, (ii) structural semantic errors are not detected while parsing but can detected at compile time using static program analysis, and (iii) runtime errors occur during execution. Orthogonal to these three categories of errors, programmers may experience logical errors, too. These are caused by flawed logic and, in contrast to structural errors, the computer cannot detect logical issues without knowing the programmer’s intention. Programmers can write code that parses correctly and runs, but which might result in incorrect and unintended behavior. In this chapter, we focus on logical errors.

6.2 A Glimpse Back in History

Learning how to recover from logical errors is an essential skill that is relevant to all programmers, independent of age and level of expertise. Even professionals are known to spend 20 to 40 percent of their time debugging [151]. For this reason, the professional community has developed various debugging tools and techniques of how to approach errors in general [120, 172, 167], and how to devise a curriculum that aligns with the specific goal of teaching debugging to novices [159]. One of various techniques that has gained widespread use is called reverse debugging [57]. Some programming environments for novices come with good examples of reverse debuggers. In the context of Logo, however, the potential of reverse debugging as not been explored so far.

The concept of debugging has been around since the early days of programming. Before the first compiler was invented by Grace Hopper, all instructions had to be given in a low-level machine language, which was quite susceptible to errors. Therefore, code was usually first handwritten in pencil, line by line, on a standardized coding sheet [158]. Only then was it translated to punch cards and finally fed to the machine for execution. Along this way, various errors could have snuck in, causing unexpected output. Robert Campbell remembers [182]: “We had to go through the operation step by step, until we found something which wasn’t right.” Using special ‘rollback’ procedures, they tried to retrace the point at which the execution started to differ from what they expected [47]. Since then, more sophisticated and elaborate tools have been developed.
In case of a logical error, programmers can navigate through their code command by command, monitor variables, and decide whether to skip over or step inside function calls. Using breakpoints, they can suspend their program at an arbitrary point in execution. Debugging tools are arguably the most powerful and efficient means to help programmers locate and fix logical errors. Learning how to handle such tools, however, often seems overwhelmingly difficult for novices.

Logical errors can occur in Logo just like in any other programming language. Thanks to turtle graphics, however, programmers can understand and trace their programs more easily: incorrect implementations are characterized through visual defects in the resulting picture. In order to locate the root of such errors, programmers need to understand how programs are executed. The original Logo documentation [63] offered a command for the purpose of debugging: `trace`, which prints all procedure invocations (input and output) that were called during execution. The command allows programmers to see what happened during execution and to perform error-analysis in a post-mortem fashion.

Even though the scientific community largely agrees on the importance of debugging for novices [135], some of the most widespread programming environments for children do not provide any notable debugging support. Scratch, one of the most widely used programming environments for primary schools, discontinued its single stepping feature when switching from version 1.4 to 2.0 [192] – much to the regret of their users. In contrast, Smalltalk’s Pharo [33] debugger allows programmers to change their code while debugging and the Python IDE Thonny [24] even offers reverse debugging. In the context of Logo, some environments (e.g., Turtle Blocks [32] and Robo Blocks [168]) provide debugging support through single-stepping, however, so far no Logo debugger allows programmers to also step backward in time. We address this problem and present a reverse debugger for XLogoOnline. First, we will elaborate what natural coping strategies Logo novices typically use when tackling logical errors.

### 6.3 What Natural Coping Strategies do Novices Use?

What approaches do Logo novices find to tackle algorithmic tasks? What are the typical errors that occur in turtle graphics? And how do children recover from their programming errors? To answer these questions, we observed how 73 students tackled algorithmic tasks in XLogoOnline.
6.3 What Natural Coping Strategies do Novices Use?

6.3.1 Setup

We conducted another experiment counting four Logo programming courses with primary school children aged 10 to 13 who had no prior knowledge in Logo programming. Each course took place at Swiss primary schools and lasted for twenty lessons. At different stages during the course (beginning, middle, and end), we handed out three algorithmic tasks (see Figure 6.1) and collected the children’s solutions and all intermediate results. We manually inspected the children’s work and found different solution paths and problems. Unlike the setup commonly used [126, 22, 82, 139] our approach does not rely on students identifying and fixing errors in predefined programs but they implement their own solutions and we manually analyzed their solutions in order to detect logical flaws they committed when solving our tasks.

6.3.2 Problem Decomposition: There is More than One Correct Solution

Problem decomposition is the first step of the cognitive process a child goes through when solving a task. Different programmers find different solutions to the same problems. Even though all tasks in our pool are rather small (it takes just a handful of lines to draw them), the participants found surprisingly many approaches that correctly solve the tasks. A square, for instance, is only composed of four simple lines and yet, depending on the starting point and the order of traversal, far more than just one or two solutions are possible. Complex shapes yield an even more diverse solution space. In Figure 6.2, we illustrate some of the recurring patterns children found in tasks 2 and 3.

6.3.3 Not Only Beginners Face Logical Problems

Once students start programming, they inevitably face logical errors. Two problems are typical among Logo programmers: (i) rotations are confused due to an inconvenient change of perspective, and (ii) novices neglect loop invariants (i.e., conditions that need to be upheld in every loop iteration). The
first problem dominates in the beginning, but, over time, it reduces and gives way to the second class of logical errors.

- **Clash of perspective:** Logo commands are interpreted from the turtle’s perspective rather than the programmer's. Hence, novices need to put themselves into the turtle’s position, which is cognitively challenging. Clashing perspectives have bewildering effects ($\texttt{rt}$ and $\texttt{lt}$ suddenly change their semantic effect). Children learn to help themselves by aligning their perspective with the turtle – first physically, then mentally.

- **Managing complexity:** Logo is a stateful language and all its commands have side effects on the turtle’s state or the pen’s state. Understanding Logo semantics means to understand what effect each command has on the overall program state. Managing local changes, however, is not enough: due to repeating programs, previously executed states may be revisited and students need to trace how the state evolves over time. Loop invariants are important when reasoning about correctness and working towards modularity. However, novices often struggle with loop invariants.

Due to several reasons, students write incorrect programs whose visual output does not fit their expectations. To recover from these errors, they need to establish a connection between each command and its corresponding effect while retracing their code. The boy in Figure 6.3, for instance, is trying to locate a flaw in his code by simulating execution using pen and paper. He closely follows each instruction on the screen and draws the corresponding effect on paper. Several attempts on the paper indicate that he is facing difficulties while retracing the program flow.
6.3 What Natural Coping Strategies do Novices Use?

1. **Sequence of commands**: Programs formulated as sequences of commands are read and executed sequentially (Figure 6.4a). The more complex a drawing, the more commands are required to describe the behavior in a corresponding program. Long and unstructured programs, however, are difficult to trace since inexperienced programmers, like the boy in Figure 6.3, are likely to lose track.

2. **Repetition**: Repetition reduces the descriptive length of a program exploiting recurring patterns (Figure 6.4b). On the one hand, it takes fewer commands to describe repeating patterns, on
the other hand the cognitive work involved in reading and tracing rises due to non-sequential information flow.

3. **Modularity**: By extending the language with new commands, students learn to hide complexity behind meaningful names (Figure 6.4c). More and more complex programs can be written without increasing the cognitive complexity involved in tracing: custom commands can be used as if they were built-ins as long as their functionality was tested thoroughly before they are used as a building block for new and more complex modules.

### 6.4 Implementing a Reverse Logo Debugger

In this section, we explain how we extended XLogoOnline with a *reverse debugger* that allows novices to manually inspect their program during execution and step through their code forward and backward in time. We present our approach and explain how it balances performance against memory consumption.

#### 6.4.1 Logo’s Language Constructs

XLogoOnline offers two main programming constructs: commands and control structures.

1. **Commands** serve as functional building blocks with observable effects: Built-in commands (e.g., `fd`, `rt`, or `setpc`) form the basic vocabulary which can be extended with user-defined commands (e.g., `square`, `triangle`, or `house`).

2. **Control structures** steer execution: They manipulate the order in which commands are executed (e.g., `repeat`, `while`, or `if`).
6.4.2 Single Stepping: Understanding Program Execution

Execution is often hidden from programmers: code magically produces some output while all intermediate steps remain a mystery. Understanding how a solution emerged is a crucial part of debugging. Single stepping is a mechanism that visualizes all steps during execution and hence gives programmers the opportunity to understand execution flow. We distinguish automatic from manual stepping depending on whether the programmer has control over execution or not.

- **Automatic stepping.** Execution can be visualized incrementally by immediately flushing all effects (i.e., all of the turtle’s movements and rotations) as the computer traverses the syntax tree. Any program’s execution can be visualized, yet different programs pose different demands on execution speed. Programs with only a few commands (e.g., those that draw a square) execute within a fraction of a second. For execution to become observable, a massive slowdown in execution speed is needed. Other programs execute thousands of instructions (e.g., \texttt{repeat 360 [repeat 360 [fd 1 rt 1] rt 1]}) and execution takes considerably longer. In those cases, a higher execution speed is more appropriate to not cause unwanted delay. In summary, finding a universal execution speed (i.e., flush rate) that works for any program is not easy. Hence, we allow programmers to pick a rate according to their needs and change the flush rate dynamically on demand. DrScheme \cite{drscheme} is a programming environment with a similar design philosophy like XLogoOnline. In contrast to Logo’s imperative nature, however, Scheme is a functional programming language and stepping is connected to a sequence of reduction rules rather than iterating over a sequence of commands.

- **Manual stepping.** Once execution speed reaches zero, the turtle’s world freezes in its current state. Conceptually, this is the moment when automatic stepping turns into manual stepping, a mode that provides additional support in tracing. Programmers manually navigate from command to command and inspect how the program state evolves along the way. When debugging, programmers need to develop an understanding of how their program state evolved into an undesired situation. For this, manual stepping is useful as it assists programmers in tracing.
6.4.3 Reverse Debugging: Going Back in Time

Even experienced programmers frequently step too far when debugging and run past locations of interest. In reverse debugging, this problem is addressed by navigating through execution backwards in time. We discuss how different approaches handle the trade-off between performance and memory consumption before presenting our approach that balances among both metrics.

Approach 1: Rerunning

One possibility to simulate the behavior of reverse debugging is to simply rerun the program from the start. With every step back, execution is re-initiated. All that needs to be stored is a single pointer to a node in the syntax tree that is currently being investigated. Instead of running through the entire program, we stop execution at whatever node the pointer currently points at.

Problem: One drawback of this solution is its scalability. Some programs take too long to execute for this solution to be feasible. For instance, \texttt{repeat 360 [repeat 360 [fd 1 rt 1] rt 1]} consists of several hundred thousand strokes. If taking one step back requires all of these instructions to be re-executed, the programmer would have to wait too long.

Approach 2: Inversion

An alternative to re-running relies on reverting the previously-executed instructions: we visit each node in the syntax tree in reverse order and undo each command along the way. This idea cannot be implemented for Logo in a purely mathematical way (i.e., without additional memory) since some Logo instructions are inherently irreversible. Also, state information can get lost due to being overwritten.

- **Insight 1: Not all commands are reversible.** Commands like \texttt{rt} and \texttt{fd} are reversible since they cause a relative change (i.e., they perform additive changes to location and orientation). Other commands like \texttt{setxy}, \texttt{setpc}, and \texttt{cs} cause absolute change (i.e., they overwrite the previous state) which makes them irreversible in a purely mathematical sense.

- **Insight 2: Reverse traversal is lossy.** Logo’s irreversible commands overwrite information that cannot be retrieved later. For instance \texttt{setpc blue fd 100 setpc red bk 100} first draws a blue line which then is painted over in red. To correctly undo \texttt{bk 100}, we need to know what pen color was used before red, which is no longer known at that point. A stack
Figure 6.5 All states that are traversed need to be stored on the stack.

Listing 6.1 One of eight snapshots taken when drawing a square.

allows us to solve this problem: state information can be stored to allow for a lossless reversal of seemingly irreversible commands.

**Approach 3: Snapshots**

Using a stack, we can easily overcome the problem of lost state information. By taking snapshots of the canvas, we capture how the program evolves over time. On demand, we can revert previously executed commands without even causing much computational overhead: the corresponding state can simply be retrieved from the stack. Each snapshot stores only a copy of (1) the canvas and (2) the turtle’s location and orientation.

Storing the entire bitmap of the canvas, however, consumes a significant amount of memory. Depending on the screen resolution, a considerable amount of information needs to be stored. For instance, during execution of the program \texttt{repeat 4 [fd 100 rt 90]} eight commands are traversed and all eight states are stored on the stack (see Figure 6.5). When executed on a screen with a resolution of 1920x1080 pixels, we end up with a gigantic amount of over 16 million pixels (i.e., almost 50 MB) that need to be stored in memory, just for a simple square.

Memory consumption can be improved by changing the internal representation of the canvas. Instead of using a bitmap, we use vector graphics to store all the information. This way, we no longer need to save a large, fixed amount of pixels after each step, but instead only save the information contained in the vector graphic (see Listing 6.1).
While we still store one snapshot per step, in this alternative representation the size of each snapshot is reduced heavily. Before, each snapshot stored a fixed number of pixels depending on the screen resolution. Using vector graphics, the size of each snapshot depends only on the amount of information (number of lines) visible on the screen. This improved approach has a reduced memory consumption that is orders of magnitude smaller than using bitmaps (e.g., for the example of a square, eight snapshots of a vector graphic account for merely 115 KB of RAM). With such a reduced memory consumption, space considerations diminish and even long and complex Logo programs can be debugged using this approach.

### 6.5 Debugging is Relevant for Everyone

Our contribution has an impact on both students and teachers: Young programmers need to learn debugging – it is an essential skill for any programmer since logical errors can neither be prevented nor can they be detected by a computer (unless the given task is known). The presented debugging tool supports children in tracing erroneous programs until they detect an error on their own. Rather than waiting for external support, they learn to approach bugs and tackle them self-sufficiently. This makes them face errors with a positive attitude; getting things to work is the main objective, rather than being afraid of making mistakes. Teachers profit from this debugging tool as well. It supports them in giving programming courses in class and focusing on individual students while the whole class is able to make progress.
Chapter 7

Evaluation

In the past four years, XLogoOnline has been used in numerous school projects and teacher training programs across the country; all of which have contributed to an ever-growing user base. More than 30 thousand visitors interacted with the web environment in the last nine months alone, counting both national and international visitors. By evaluating access logs, it is possible to determine what technical equipment these visitors use and in which cities, cantons (states), and countries the environment is used in most. In order to get a better understanding of what errors novices struggle with and to evaluate the error diagnostics presented in Chapter 4, we collected and analyzed a trace of 2 Million faulty Logo programs that were committed in more than 80 000 user sessions. We found that XLogoOnline’s error diagnostics allow up to 97% of all structural errors to be detected and reported at compile time, leaving only a small fraction of all such errors undetected which then need to be debugged at runtime.

7.1 Demographics

XLogoOnline’s server-client architecture allows anonymous log files to be collected and evaluated. By analyzing the access logs collected over nine months, it was possible to get an insight into the technology used by the platform’s visitors and their rough location. We analyzed the data of more than 33 000 users in order to answer to the following three questions: (i) “what technology are visitors equipped with and how do they use our environment?”, (ii) “which regions of Switzerland use the environment most?”, and (iii) “where are the international visitors from?”. 
All information presented in this section is derived from aggregated data that was collected anonymously. The following data analysis depends on some specific assumptions and definitions. For this section, we define the terms user, location, and page view like this:

- **Users**: A user (or visitor) is a person who accesses the webpage XLogoOnline using a specific device with a specific browser. The user ID identifies unique distinct visitors using a cookie that is set up during the first user interaction with the webpage and which is kept in the browser until manual reset. Clients who disabled first-party cookies are identified by their IP address in combination with their browser’s user agent. A consequence of this choice is that the number of users might not correspond to the actual number of people who used the webpage since (i) the same person can be counted several times if they use different devices or browsers, and (ii) several people can be counted as one user if they share the same device or browser.

- **Location**: Using IP geolocation services, it is possible to locate users and map their position to a specific city and country using only their IP address. We created a map of all users who requested the XLogoOnline environment within a period of 9 months. The accuracy of IP geolocation differs from case to case – some IP addresses can be located with high accuracy whereas others reach only moderate or even low accuracy. Geolocation services report a reliability measure with each request, ranging from 0 (lowest reliability) to 1 (highest reliability). We discard all requests that do not reach at least an accuracy level of 0.5 in order to reliably determine from which canton (state) or region a user request originates.

- **Page view**: We consider the number of page views as the number of times XLogoOnline was visited by a unique user. These page views are used in our statistics as a metric of user activity which is a legitimate technique in general but can be distorted by users who often refresh their page or close and re-open their browser. These cases result in double-counting.

### 7.1.1 Infrastructure and Programming Setup at Schools

From the available access logs, both the access day and access time of a page request can be determined. With most users presumably accessing the webpage from schools and educational institutions, it is to be expected that access rates fluctuate during a day and throughout different days of a week. We consolidated more than 320 000 page views and grouped them according to (i) the day of the week, and (ii) the hour of day when the request was made. The results are shown in Figure 7.1 and
Figure 7.1 XLogoOnline is used more heavily during weekdays than on weekends and even the common tradition of school-free Wednesday afternoons is reflected in the data.

Figure 7.2 Visitors access the learning environment mostly during the time frame from 7am to 5pm which roughly corresponds to a typical school day in Swiss public schools.

Figure 7.2, which confirm the expectation: XLogoOnline is accessed roughly three to four times more often during week days than on weekends. On Wednesdays, the access rate is lower than on other week days, which can be explained by the traditionally school-free afternoon on Wednesdays in most Swiss public schools. Figure 7.2 shows how user requests distribute throughout the day. 12% of all daily activity happens in the evening (after 5pm) and early morning (before 7am), outside the typical block times used at Swiss schools [183]. These requests likely reflect teacher preparation, homework, and leisure time activities in our target group.

By analyzing page requests, it is possible to determine what operating systems and browsers XLogoOnline’s visitors use. This is interesting since it provides a picture of what infrastructure schools typically work with and hence what platforms XLogoOnline should cater for in the future. For this evaluation, we worked with the user data of 25 000 visitors whose IP address could be located within Switzerland (i.e., visitors which span across all language regions of the country). In 2002, the Swiss Federal Office of Statistics published a report that illustrates the status quo of school infrastructure from twenty years ago [62]. By comparing our results with the status quo of 2002, we were able to show how Swiss school ICT infrastructure changed over the last 20 years.

The 2002 report conveys a picture of school computers which were strongly dominated by two operating systems: 97% of all devices ran either Windows or macOS. Today, among the Swiss community of 25 000 XLogoOnline users, this rate has dropped to merely 84%. Schools increasingly seem to use mobile devices running iOS or Android rather than traditional laptop computers, as shown in Figure 7.3. Nowadays, the range of operating systems and platforms used in Swiss schools is much more diverse than 20 years ago.
In terms of browsers, there is a rather large diversity, too, as shown in Figure 7.4. With these values, the browser share is more evenly distributed among our users than among the Swiss average which, at the time of writing, consists of 40% Chrome, 35% Safari, 8% Firefox, and 7% Edge [7]. For most functionality provided by XLogoOnline, any modern browser can be used. Only physical program execution (as presented in Section 5.2.2) is restricted to browsers running the Chromium engine, i.e., Chrome, Edge, and Opera.

The conclusion we can draw from these results is that Swiss schools use a broad spectrum of different operating systems and browsers for classical desktop and laptop computers but increasingly mobile devices, too. It is to be assumed that this trend will continue and with this wide variety of platforms and operating systems, developing native solutions is not always easy. Browser-based software has the advantages of being platform-independent and, moreover, that no installation setup and maintenance work is required from teachers.

7.1.2 Coverage in Switzerland

Due to the joint work of various Swiss educational institutions (including PH Bern, PH Luzern, PH Graubünden, University of Basel, and ETH Zurich), the curriculum presented in Chapter 3 has been incorporated into the time tables of hundreds of schools across the country over the past fifteen years. The programming environment presented in this thesis is the primary software accompanying this curriculum and it is widely used in schools. In this section, we tackle the question in which cities and cantons XLogoOnline is used most and how the usage spreads across Switzerland.
7.1 Demographics

Table 7.1 This map shows the location where 25,000 distinct users accessed XLogoOnline from. Each red dot represents the user base in a given town and the larger the dot the more XLogoOnline users we counted in a given location. On the right, the 12 largest XLogoOnline communities in Switzerland are listed.

<table>
<thead>
<tr>
<th>City</th>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zürich</td>
<td>4852</td>
</tr>
<tr>
<td>Bern</td>
<td>2044</td>
</tr>
<tr>
<td>Chur</td>
<td>1133</td>
</tr>
<tr>
<td>Basel</td>
<td>816</td>
</tr>
<tr>
<td>Luzern</td>
<td>556</td>
</tr>
<tr>
<td>St. Gallen</td>
<td>438</td>
</tr>
<tr>
<td>Lausanne</td>
<td>323</td>
</tr>
<tr>
<td>Geneva</td>
<td>321</td>
</tr>
<tr>
<td>Wetzikon</td>
<td>310</td>
</tr>
<tr>
<td>Zug</td>
<td>235</td>
</tr>
<tr>
<td>Biel</td>
<td>230</td>
</tr>
<tr>
<td>Langenthal</td>
<td>229</td>
</tr>
<tr>
<td>Total</td>
<td>24895</td>
</tr>
</tbody>
</table>

In order to answer the question where XLogoOnline is accessed from, 24,895 IP addresses were analyzed and associated with a specific city. The result is visualized in Table 7.1 where colorful dots illustrate the spread of the XLogoOnline user base in Switzerland. Each dot represents a city or town where XLogoOnline was accessed from at least once during the nine-month time frame of the investigation. The radius of each circle visualizes the number of unique users in a given location.

We observe that a significant number of users accesses the environment from Zürich, Bern, Chur, Basel, or Luzern. These cities harbor relatively large populations and they each also locate one of our partner institutions, i.e., a university of teacher education. Logo finally has been adopted by several Swiss universities of teacher education as part of their teacher training. Each of these institutions actively engages in the promotion of Logo courses at school which is why we observe an active participation in the metropolitan areas around larger cities. Several school projects could be conducted in smaller cities and villages across the country. There are 79 locations for which we count more than 50 unique users in the past nine months.

To gain an understanding of how XLogoOnline is used in each of the 26 sovereign regions of the country, we further processed the data and affiliated each user with one canton. Figure 7.5 shows the result: cantons Zürich (ZH), Bern (BE), and Graubünden (GR) are among the three most active cantons in using our programming environment in absolute terms. Seeing that Zürich and Bern are
the two cantons with the largest population size in the country [73], it is not surprising to see a large user base in these two cantons. Figure 7.6 shows the population sizes of all 26 cantons. In order to take them into account, we normalized the number of users per canton by its population size. The result is shown in Figure 7.7.

Despite its large number of inhabitants, Bern can still be considered active even in relative terms. Additionally, we observe that some smaller cantons show high activity, too, for instance Zug (ZG) and Glarus (GL). One result that seems especially interesting is the activity rate measured in the canton of Graubünden. In this region of the country, roughly 1% of the overall population can be considered active users of XLogoOnline. This effect can be explained by a long-established Logo tradition in local schools as well as a strong initiative of the Pedagogical University of Graubünden,
which declared Logo programming to be an obligatory core component of teacher education and in-service teacher training.

Besides Switzerland, there is also a growing community in Germany’s North-Rhine Westphalia with 5,118 users as well as 482 users from Italy which count towards our international visitors.

### 7.2 Error Analysis

The question of what errors novice programmers often run into is interesting from numerous viewpoints: teachers need to get an objective picture of the level of knowledge and problems of their protégés; especially seeing that educators’ beliefs often do not correspond with reality [40, 102], curriculum designers are challenged to incorporate the topic of debugging more actively into their materials [136, 126], environment developers want to find appropriate error messages [29, 30, 116], others simply see an opportunity in comparing the differences between novices and experts [191] and how the error recovery behavior changes over time [56], still others try to establishing a metric to express a programmer’s success in fixing errors [103]. With BlueJ’s Blockbox project [42, 41], there is a massive data pool for researchers to ask questions like “What are the most common errors Java novices run into?” [23]. The same question is interesting from the perspective of Logo, too. Klahr and Carver have examined the most common logical errors in Logo programming [110]. So far, however, a large-scale error collection project similar to Blackbox has not been conducted for Logo yet. In Section 4.3.1, we already presented a preliminary study that summarized what structural programming errors novice Logo programmers typically make in the course of their learning using our curriculum. In this section, we build upon these previous results and provide a quantitative measures for the previously-presented error classes. Over a time span of 11 months, we collected more than 2 million erroneous Logo programs from more than 80,000 user sessions. 97% of these accumulated errors can be detected and highlighted at compile time using the error diagnostics presented in Chapter 4. Only a mere 3% of all structural programming errors remain to be debugged at runtime. We are convinced that this approach will prove useful to novices seeing that both Kohn [114] and Edwards/Kandru [56] mention that some of the most common errors in their respective studies could be resolved easily by novices when simply pinpointing the error. In our context, this means that a large fraction of errors may be easily dealt with before execution even starts.
7.2.1 Setup

During an 11-month period from January to December 2020, we collected a data set of more than 2 million snapshots of failing Logo programs that were executed on the XLogoOnline programming platform. This data can be used to establish an understanding of individual user behavior by grouping related snapshots to consecutive user sessions. With 80,393 anonymous user sessions, the presented data reveals interesting insights into the problems young Logo programmers face during their learning. The range of errors covered in this section spans from simple syntactic errors to intricate misunderstandings of advanced programming concepts. For the purpose of the following evaluation, we define the terms user, error, snapshot, and user session as follows:

1. **User**: This study is based on anonymized data; users are identified by randomly generated UUIDs, which are assigned at the beginning of the first user interaction. These identification tags are kept until the user session ends (e.g., by refreshing the webpage or closing the browser window). As a result, our concept of a user is tied to a browser session which usually does not span over longer periods of time. Consequently, a recurring user may be identified with different identification tags each time he or she accesses the programming environment.

2. **Error**: This study covers a large range of structural program defects from syntax errors to runtime errors. All of the issues covered in this study cause the execution pipeline to fail at some point along the process. In addition to classical syntax errors (which cause parsing failures) and runtime errors (which cause execution failures), we introduce a third category of errors called structural semantic errors. This class of errors can be detected by means of a static program analysis, which allows a large group of errors to be reported at compile time, despite them not causing parse failures per se.

3. **Snapshot**: For this study, we documented all user activity related to errors by taking snapshots of failing user programs. A snapshot involves the program text, an identification tag, as well as a timestamp. A new snapshot was taken whenever a user initiated the execution of a structurally defective Logo program. By the end of the study, the server had stored a large data set of more than 2 million snapshots that can be clustered into user sessions and ordered chronologically.

4. **User Session**: To examine the data on a per-user basis, the snapshots are grouped into individual user sessions and sorted chronologically. All snapshots that contain the same identification tag
can be bundled into a user session and put into a chronological order based on the respective snapshot’s timestamp. The number of snapshots per user session varies considerably and ranges from just a single snapshot per session to more than 4,000 flawed program calls that are associated with a single user.

7.2.2 Results

At the heart of this analysis are two questions, one of quantitative and one of qualitative nature: (i) “How many structural errors do Logo novices encounter in the course of a learning session?” and (ii) “Which kinds of errors do novices make, and how often does each category of error occur?” We answer these two questions separately and provide empirical results.

Number of Errors Encountered in Each Session

In order to find out how many structural errors young programmers make during a programming session, we determined the duration of an average programming session on the one hand, and the number of errors per user and session on the other.

To estimate the duration of a user session based on our snapshots, we calculated the duration between the first and the last snapshot in each session. This metric provides a suitable lower bound for the duration of a user session. Since not all user sessions contain the required minimum of at least two snapshots, we filtered out 22,568 sessions which contain less than two snapshots. The remaining 57,825 user sessions yield an average duration of 27 minutes and 33 seconds (standard deviation 191 minutes).

Each user session contains an average of 30.5 snapshots (standard deviation 105.5) which typically contain more than one error each. Specifically, each snapshot contains an average of 6.7 distinct errors (standard deviation 13.9). Extrapolated to one hour, this implies that novice programmers produce 445 individual structural errors in just 60 minutes of programming. What seems to be a rather large number of errors contains both simple blunders and more complex misunderstandings. Next, we discuss what kinds of errors novices typically make and how often each class of error occurs.

Kinds of Errors and Relative Frequencies

XLogoOnline distinguishes three types of errors that occur at different points in the execution pipeline: (1) syntactic errors that lead to parse failures, (2) structural semantic errors that are detected during
Figure 7.8 XLogoOnline detects 97% of all structural errors at compile time (i.e., during parsing or in static program analysis).

Our approach aims at detecting and reporting as many structural errors as possible already at compile time and visualizing these errors using simple in-line error markers (so-called “squiggly lines”). We claim that a large fraction of all errors can be detected, reported, and fixed on-the-fly while a programmer is still typing. The following evaluation shows that 97% of all structural errors collected in this study can be detected at compile time. We present a breakdown of the most common error classes (see Figure 7.8) and quantify their frequency.

1. **Syntax Errors (Detected during Parsing)**

   While parsing a Logo program with the grammar presented in Section 4.2.3, four classes of syntactic errors can occur, namely (a) excess tokens, (b) illegal tokens, (c) missing tokens, or (d) otherwise invalid syntax. Some examples of these classes of errors are as follows:

   - **Excess tokens (4.6% of all errors overall)**
     - `house :125`  
       Some errors involve a procedure call whose argument includes a colon. This issue might relate to novices who just learned the concept of parameters and who do not yet know how to pass arguments to procedures.
     - `[fd 100 rt 90]`  
       Some programmers struggle to grasp the meaning of brackets in programming. Once introduced with repetition, they start using brackets outside of their intended scope.
• **Illegal tokens (1.8% of all errors overall)**

    `repeat 3 (rt 90)`  The `repeat` keyword cannot be used with parentheses or curly braces but only with brackets.

    `to olympia ... end`  Although XLogoOnline is implemented to be more permissive with special characters such as German umlauts ä, ö, ü, or French accents é, è, â, there is still a large number of characters that are not permitted in Logo programs (e.g., emoticons).

• **Missing tokens (1.26% of all errors overall)**

    `repeat 4 fd 100]`  In this program, the parser identifies that the token ‘[’ is missing before `fd`.

    `to olympia ...`  This program is missing the `end` keyword and is thus structurally defective. With 0.8%, this error makes the majority of all syntax errors related to missing tokens.

• **Invalid syntax (1.35% of all errors overall)**

    `make gr 3`  The `make` command has a special syntax that requires the variable name to be preceded by opening quotes. Without these quotes, the parser fails at identifying `gr` as an identifier of a local variable or parameter.

2. **Structural Semantic Errors (Detected during Static Program Analysis)**

    Programs that are free of syntax errors (i.e., they can be parsed correctly) are not automatically free of other structural errors. In order to examine language-semantic questions such as “Is there a program declaration for each program call?” or “Do all program calls provide the specified number of arguments?”, we use static program analysis. There are 8 classes of errors covered by this analysis: (a) program calls without an associated program declaration, (b) program calls with too few arguments, (c) program calls with too many arguments, (d) program calls to procedures with internal errors, (e) parameter or variable access without a corresponding variable declaration, (f) duplicate program declarations, (g) keyword as program name, and (h) duplicate parameter declarations. These eight error classes account for 88% of all errors committed by Logo programmers in our study. This percentage is comprised of 72.2% program calls without associated program declarations, which constitute a clear majority of all errors in our study. Program calls with missing arguments (7.9%) or excess arguments (4.4%) also
account for a considerable portion of all errors. In the following overview, we show examples of structural semantic errors found to be common cases in our study:

- **Program calls without associated declaration (72.2% of all errors overall)**
  
  fd100 rt90  
  10% of all errors are related to missing spaces between built-ins and their argument, such as fd100 instead of fd 100.

  fd d160  
  0.6% of all errors are due to incorrect space separation in built-in commands.

  df 100  
  2.2% of all errors are rooted in reversed letter order in built-ins. Surprisingly, df and tr are much more likely to occur than tl and kb: out of 100,000 rt- or fd-commands, respectively, 43 and 29 are affected by swapped letters. With lt and bk, in comparison, we count only 5.8 and 4.7 instances in 100,000 commands. Presumably, the proximity of two keys on the keyboard makes this error more likely.

  reape 4 [fd 100]  
  59.4% of all errors fall into the general category of spelling errors. The various reasons include some fraction that is caused due to our audience’s early stage of learning English.

- **Program calls with too few or too many arguments (12.3% of all errors overall)**

  fd 100 pu 100  
  4% of errors are due to built-ins being passed too many arguments. With 1%, pu is the command most often involved in this problem. Examples suggest that pu is often misunderstood as a movement command.

  fd 10 wait fd 10  
  7% of all errors are related to built-ins being provided with too few arguments. Most commonly, this happens with fd and rt, but it does occur with other built-ins as well. Cases related to setpc often involve typos in color names (i.e., setpc yellow fd 100)

- **Program calls to procedures with internal errors (2.6% of all errors overall)**

  to a ...  
  Procedures whose declaration contains errors (e.g., a missing end) cannot be invoked.
• **Parameters or variables that were not defined** (0.9% of all errors overall)

Some students try to invoke a procedure with an abstract, undefined parameter rather than a concrete value.

Others use both parameter name and value together in an invocation. This error suggests that learners are unsure how their argument is being linked to an abstract parameter name on the definition site.

The remaining three error classes in this category are *duplicate program declaration* (which accounts for 0.2% of all errors) as well as *invalid program name* and *duplicate parameter declaration* (which both account for 0.01% of all errors in our study).

3. **Runtime Errors (Detected during Execution)**

Among all runtime errors that XLogoOnline does not currently detect as part of its static program analysis, there are two main categories to be considered, namely (a) division by zero, and (b) out-of-bounds errors with color indices. We present two selected examples from our dataset and discuss their peculiarities and implications.

• *Division by Zero*

Dividing a number by zero is mathematically invalid. The result of such a division can cause one of the following five behaviors on a computer, depending on which programming language and type of number is used: (a) a hard crash, (b) an exception, (c) not-a-number, (d) positive infinity, or (e) negative infinity. The XLogoOnline interpreter reacts to a division by zero by throwing an exception. All runtime exceptions are subsequently caught by the environment and visualized for the user.

The program shown in Listing 7.1 results in a runtime exception due to a division by zero: the statement \( \frac{360}{\text{RANDOM} \, 6} \) may fail since \( \text{RANDOM} \, X \) produces a random value that can become 0. The statement has a chance of 1 in 7 to end up in a runtime error. Due to repeated invocation, however, increases the risk of an exception with every loop iteration. Based on the program text, we assume that the programmer intended to draw a polygon with a random number of corners. Under this assumption, we note that the program contains one more error: \( \text{RANDOM} \) returns a new value in each loop iteration which means that the resulting shape, the alleged “polygon”, may have different angles in each corner.
Listing 7.1 This code snippet produces a runtime error due to a division by zero. Due to its probabilistic nature, the error may not show in every run.

```
repeat RANDOM 6 [
    fd 100 rt 360/RANDOM 6
]
```

Listing 7.2 This code snippet produces a runtime error due to an out-of-bounds error connected with the `setpc` command. The allowed range of input is 0 to 16, whereas this program provides the command with the argument -12.

```
repeat 12 [
    setpc 4-16 fd 600 rt 90 fd 30 rt 90 fd 600 rt 90 fd 30 lt 90
]
```

• *Out-of-bounds color index:*

The `setpc` command takes a color index between 0 and 16. Each value in this range is assigned a specific color. Values outside of this range have no semantic meaning, e.g., neither 19 nor -3 are permissible arguments for the `setpc` command and both lead to a runtime error. The program in Listing 7.2 provides `setpc` with an arithmetic expression (i.e., 4-16) which evaluates to a negative number. Due to its negative argument, the statement is rejected and causes a runtime error.

Based on the program text, we assume that the programmer intended 4 − 16 to be interpreted as a range specification rather than an arithmetic expression. The minus symbol is often used to specify a range (e.g., “read pages 3-9”) in addition to its typical mathematical meaning. In this sense, we classify this issue as a *problem of overlapping character semantics* in different contexts.

### 7.2.3 Most Logo Programming Errors Can Be Detected at Compile Time

The presented evaluation provides a deeper insight into the structural errors Logo novices encounter during their learning. These errors cover the spectrum from typos and simple blunders, to more complex and fundamental misunderstandings. In Python [114], C++ [181], and Java [56], there seems to be a large fraction of errors that can be attributed to minor mistakes or inadvertent slips. With this evaluation, we able able to confirm the same statement for Logo, too. In our case, a striking
fraction of 72% of all errors are unresolved identifiers; a much larger fraction than reported for Python[114, 154] and Java [102], respectively. Arguably, this is an effect of Logo’s special syntax – complete proceduralization is an error-prone attribute of Logo just like indentation in Python and Semicolons in Java. We claim that a large majority of all structural errors in novice’s Logo programs are minor bugs that can be corrected by programmers without too many problems. Whether or not an early pinpointing of typical errors actually helps novices to reduce the time they spend debugging until their code is runable, still remains to be shown in future.
Chapter 8

Conclusion

8.1 Summary

With the introduction of computer science as a new school subject, programming is enjoying a rapid and exciting boost in Swiss primary schools. Programming is a creative activity that promotes constructive problem solving, but it simultaneously also requires a high degree of precision and is prone to errors. Related work has addressed the most common errors novices run into for languages like Java and Python [154, 23, 56]. It was shown that students spend considerable amounts of time with error handling [151]. This reality poses a challenge for teachers, who are responsible for numerous students at once and often face considerable performance gaps between learners [104]. Our own experience shows that Swiss primary school teachers often feel overwhelmed when too many students require their help with fixing programming errors simultaneously. We highlight that, due to this situation, autonomous troubleshooting [72] is an essential requirement for the successful introduction of programming classes into Swiss (primary) school.

Troubleshooting covers many factors and, just like programming itself, is something that takes time to learn and must be built up slowly. Novice programmers make a variety of errors, from simple typos and syntactical inaccuracies [72] to deep logical flaws [113, 115]. Previous research [129] proposes the introduction of block-based interfaces, which allow different classes of errors to be introduced at different points in time. Using block-based interfaces, structural programming errors such as typos or missing arguments can be postponed temporarily, which allows young programmers to focus on logical errors first. Once programmers transition into the world of text-based programming, they also encounter structural errors, which requires an increased level of syntactic precision.
Currently, the scientific community is still divided as to when the transition from block- to text-based programming should optimally take place. While some educational institutions pursue a block-based approach long into tertiary education [142], others require that at latest by secondary-school education, students should have transitioned to text-based programming [143]. We are firmly convinced that even primary school children can become proficient programmers in a text-based programming environment, given the right tools to support autonomous troubleshooting.

This thesis discusses an approach to primary school programming that transitions from block-based to text-based programming within K–6. At all levels, age-appropriate error diagnosis tools are provided to support learners in fixing programming bugs autonomously (i.e., without necessarily requiring the external help of a teacher). The tools presented in this thesis comprise an extensive compile-time check for structural errors with a syntax checker, a semantic verification, and a type checker. With these services, 97% of all structural errors can be detected and visualized in a proactive fashion, that is at compile time. For logical errors, a bidirectional debugger was developed which allows defective programs to be traversed both forwards and backwards in time and thus also contributes to autonomous learning.

The featured programming environment enjoys a continuously growing user base. In 2020 alone, several thousand individuals accessed the learning environment from all across Switzerland, Germany, and Italy. Using a tight feedback loop with these users, we are able to study which programming difficulties Logo novices experience, adapt our materials, and evaluate the effect of our interventions with real users. In the long run, this approach makes it is possible to develop a programming environment that acts as an individual learning companion which facilitates students’ troubleshooting and provides tailor-made support.

8.2 Future Directions

The introduction of computer science as a compulsory subject in Swiss primary school has only just begun and more outreach measures are still needed to fully establish the subject and to build up peoples’ understanding of what role computer science plays for general education. Being a core component of computer science, programming is one of the activities that now needs to be integrated and firmly anchored into the education system. Children of all ages need to be motivated for the creative and precise way of thinking that is endemic for computer science in general and for
8.2 Future Directions

programming in particular. In order to boost the motivation for programming, additional software components should be developed which ought to help mobilizing broad masses. One way of doing so consists in creating a system for large-scale, recurring Logo competitions for primary school students in and outside of Switzerland.

Since the seventies [6], large-scale programming competitions have been offered to computer science enthusiasts to compete with others and to challenge themselves with a fun programming experience. Nowadays, such competitions are increasingly targeting younger participants, too [5], but the majority of all computer science competitions still aims at highschool or university students. For primary schools, the number of computer science competitions is still rather small and most existing competitions either focus on robotics [121, 3] or unplugged tasks which often do not require classical programming per se [1, 17]. To the best of our knowledge, there is no contest that replicates the experience of a traditional programming contest for primary school students.

Traditional programming competitions are based on the idea of making participants come up with an algorithm that solves a given class of problems. The solution must be described in a programming language and submitted to an external entity for automatic assessment. Such an entity (a “judge”) typically runs the provided program on secret test cases whose output must match a predefined result. Using textual output, the comparison of results is usually straight forward. This is not the case in Logo: turtle graphics produces graphical results rather than text and, in order to leave algorithmic freedom, problems are typically phrased somewhat open (e.g., no starting point, order of drawing, or dimension may be specified). Rather than testing for an exact match, a Logo judge must perform a similarity check and accept all results that are graphically similar irrespective of position and size.

Two preliminary prototypes of Logo judges have already been developed as proof-of-concepts in our group. The described similarity check succeeds for a number of tasks based on a rescaled snapshot of the cropped result involving pixel-by-pixel matching [60]. Although the algorithm can successfully handle some cases, it produces false negatives in others. Problematic cases include (i) drawings with peripheral white/correction lines that unwittingly affect the size of a cropped image and (ii) pixel inaccuracies at points of color transition. Most of these instances lead to pixel discrepancies that are invisible to the human eye and yet, they still cause an automatic judge to reject a given solution. In the context of a competition, false negatives of this form are unacceptable and must be avoided by all means.
Overall, several questions need to be answered before an environment like XLogoOnline could be used as a framework for international Logo programming competitions. Several of these questions are related to automatic assessment in the context of turtle graphics tasks, such as: (i) “which circumstances make a turtle graphics problem instance suitable for automatic assessment?”, (ii) “how can turtle graphics programs be assessed automatically without false negatives especially considering that there are infinitely many correct and incorrect solutions?”, and (iii) “is there a difference between boys and girls in how novice programmers tackle problems in turtle graphics?”. Although Logo has a rather long and rich history as a research topic, there are still many questions left to be answered in future.
Bibliography


Bibliography


Appendix A

to manycenterquadrat :size
make "number :number - 1
if :number > 0 [ rt 90 fd :size lt 90]]
end

to checkerquad :length :size
repeat :length [ 
    checker :length :size pu bk :size lt 90 fd (:length - 1)*:size rt 90
]
end

to checker1 :number :size
while [:number > 0] [ 
    manycenterquadrat :size make "number :number - 1 if :number > 0 [  
        rt 90 fd :size lt 90
    ]
]
end

to checker2 :number :size
while [:number > 0] [ 
    manycenterquadrat :size make "number :number - 1 if :number > 0 [  
        rt 90 fd :size lt 90
    ]
]
to dbsp1
  cs
  repeat 4 [ fd 100 rt 90]
  repeat 4 [ fd 200 rt 90]
  repeat 4 [ fd 300 rt 90]
  repeat 4 [ fd 400 rt 90]
end

to quadrat :seite
  repeat 4 [ fd :seite rt 90]
end

to d1 :seite
  repeat 6 [ fd :seite rt 60]
end

to d2 :seite
  repeat 360 [ fd :seite rt 1]
end

to rechteckizu2 :seitekurz
  repeat 2 [ fd :seitekurz rt 90 fd 2*:seitekurz rt 90]
end

to vieleck :eck
  repeat :eck [ fd 40 rt 360/:eck]
end
to d4 :seite
setpc red
repeat :seite [ fd :seite bk :seite rt 90 fd 1 lt 90]
end

to d5 :schritt
repeat 20 [
    setpc black
    quadrat 100 wait 1
    setpc white
    quadrat 100 fd :schritt
]
end

to d6 :gross
repeat 2 [ repeat :gross [ fd 3 rt 1] rt (360-2*:gross)/2]
end

to dbsp3
setpc red repeat 12 [ fd 100 rt 360/12]
setpc green repeat 20 [ fd 50 rt 360/20]
setpc blue repeat 5 [ fd 200 rt 360/5]
end

to vielecke :eck :seite :farbe
setpc :farbe repeat :eck [ fd :seite rt 360/:eck]
end

to d9 :laenge :f
rt 90 repeat 10 [ setpc :f fd :laenge setpc white fd :laenge]
end
do d10 :f1 :f2 :f3 :gr
  setpc :f1 fd :gr rt 120
  setpc :f2 fd :gr rt 120
  setpc :f3 fd :gr rt 120
end
do d11 :f1 :f2 :gr1 :gr2
  repeat 2 [ setpc :f1 fd :gr1 rt 90 setpc :f2 fd :gr2 rt 90]
end
do d12 :l :b
  repeat :b [ fd :l bk :l rt 90 fd 1 lt 90]
end
do d13 :gr
  repeat 5 [ fd :gr rt 90]
  repeat 3 [ fd :gr lt 120]
end
do d14 :x :y :z
  rt 90 fd 75 bk 150 fd 75 lt 90
  fd 200 rt 90 rt 45 fd :x bk :x rt 90 fd :x bk :x
  rt 135 fd 100 rt 135 fd :y bk :y rt 90 fd :y bk :y
  rt 135 fd 100 rt 135 fd :z bk :z rt 90 fd :z bk :z
end
do d15 :f :gross
  setpc :f
  repeat 2 [ repeat :gross [ fd 3 rt 1] rt (360-2*:gross)/2]
end

to quadfolge :anzahl :laenge
repeat :anzahl [quadrat :laenge rt 90 fd :laenge lt 90]
end

to dreieck :seite
repeat 3 [ fd :seite rt 120]
end

to d16 :anz :seite
repeat :anz [ rt 30 dreieck :seite rt 60 fd :seite lt 90]
end

to stufe :b :h
fd :h rt 90 fd :b lt 90
end

to d17 :anz :b :h
repeat :anz [stufe :b :h]
end

to kreis :g :f
setpc :f
repeat 360 [ fd :g rt 1]
end

to d18 :f1 :f2 :f3 :g
kreis :g :f1
repeat 180 [ fd :g rt 1] rt 180
kreis :g :f2
repeat 180 [ fd :g rt 1] rt 180
kreis :g :f3
repeat 180 [ fd :g rt 1] rt 180
end

to dreiquadrat :f1 :f2 :f3 :g1 :g2 :g3
setpc :f1 quadrat :g1 rt 90 fd :g1 lt 90
setpc :f2 quadrat :g2 rt 90 fd :g2 lt 90
setpc :f3 quadrat :g3 rt 90 fd :g3 lt 90
end

to blattschmal :gross
repeat 2 [ repeat 100 [ fd :gross rt 1] rt 80]
end

to d19 :g :anz :f
setpc :f
repeat :anz [blattschmal :g rt 360/:anz]
end

to d20 :g :zeile :spalte
repeat :zeile [quadrat :g rt 90 fd :g lt 90]
lt 90 fd :zeile*:g rt 90 fd :g
end

to quadf :g :f
setpc :f
repeat :g [ fd :g bk :g rt 90 fd 1 lt 90]
end

to zeile :f1 :f2 :anz :g
repeat :anz [quadf :g :f1 quadf :g :f2 ]
l t 90 fd 2*:g*:anz rt 90 fd :g
end

to d21 :f1 :f2 :zeilen :spalten :g
repeat :zeilen [zeile :f1 :f2 :spalten :g
zeile :f2 :f1 :spalten :g]
end

to d22 :schritte :groesse
repeat :schritte [
setpc black dreieck :groesse wait 1
setpc white
dreieck :groesse
rt 90 fd 2 lt 90
]
end

to dbsp6
vielecke 2 200 red
vielecke 1 200 green
vielecke 0 200 blue
end

to vieleckeab3 :eck :seite :farbe
end

to d23 :seite
if :seite >= 10 [quadrat :seite]
end
to d24 :seite
if 6*:seite >= 180 [d1 :seite]
end

to zweiquadrate :seite1 :seite2
repeat 7 [ fd :seite1 rt 90]
rt 90
repeat 4 [ fd :seite2 rt 90]
end

to d25 :seite1 :seite2
if :seite1 = :seite2 [zweiquadrate :seite1 :seite2 ]
end

to dbsp7 :wahl
if :wahl = 0 [quad100]
if :wahl = 1 [ repeat 4 [ fd 100 bk 100 rt 90] ]
if :wahl = 2 [ repeat 4 [quad100 rt 90] ]
end

to quad100
repeat 4 [ fd 100 rt 90]
end

to d26 :x
if :x < 1 [
    repeat 4 [ fd 100 rt 90]
    fd 100 rt 90
    repeat 3 [ fd 100 lt 120]
]

if :x = 1 [ 
    repeat 3 [ fd 100 rt 120]
] 
if :x > 1 [ 
    repeat 360 [ fd 1 rt 1] 
] 
end 

to d27 :s :anz 
if :s*:anz < 1600 [vieleckSeite :s :anz]
end 

to vieleckSeite :s :anz  
repeat :anz [ fd :s rt 360/:anz]
end

to d28 :gr  
if :gr > 400 [ repeat 4 [ fd 400 rt 90] ]
if :gr <= 400 [ repeat 4 [ fd :gr rt 90] ]
end

to d29 :anz :gr 
if :anz > 2 [  
    if :anz*:gr < 1600 [  
        repeat :anz [ fd :gr rt 360/:anz]  
    ] 
]
if :anz <= 2 [  
    repeat 4 [ fd 300 rt 90]  
    rt 90 fd 20 lt 90 setpc white fd 200 setpc black 
    repeat 360+180 [ fd 0.8 rt 1] lt 90 setpc white
fd 120 rt 90 setpc black repeat 360 [ fd 0.8 rt 1] setpc white
fd 170 rt 180 lt 90 fd 200 rt 90 setpc black
repeat 180 [ fd 2 rt 1]
]
end

to zeiger :lang
repeat 5 [ fd :lang bk :lang rt 90 fd 1 lt 90] lt 90 fd 5 rt 90
end

to z1 :dauer
repeat 60 [ setpc black zeiger 200 wait :dauer
setpc white zeiger 200 rt 360/60]
end

to fenster :gr :f
setpc :f
repeat 4 [quadrat :gr rt 90]
end

to fensterecke :gr :f
fd :gr lt 90 fd :gr rt 90
fenster :gr :f
bk :gr rt 90 fd :gr lt 90
end

to gitter4mal4 :gr :f
repeat 4 [fensterecke :gr :f rt 90]
end

to zweidreiecke :gr
repeat 4 [ fd :gr lt 120]
repeat 3 [ fd :gr rt 120]
end

to einzelnesdreieck :gr
repeat 4 [ fd :gr lt 120]
end

to z4 :anz :groesse :gerade
rt 90
setpc red
if :gerade = 0 [ repeat :anz [zweidreiecke :groesse] ]
if :gerade = 1 [ repeat :anz [zweidreiecke :groesse] einzelnesdreieck :groesse ]
end
Appendix B

BlueBot is a Bluetooth device that supports Bluetooth version 4.0+ and specifically Bluetooth-Low-Energy (BLE). Its communication protocol is based on a server-client model in which BlueBot acts as the server that provides certain functionalities to its client (e.g., the learning environment XLogoOnline). To implement the connection from a webpage to a physical device, we use the new Web Bluetooth technology, which is currently supported by the browsers Chrome, Opera, Edge, Samsung Internet, and Android Browser (with a current usage share of more than 50% in Switzerland [7]). BlueBot offers a generic attribute profile (GATT) which specifies the protocol of how external devices can communicate with the robot. This protocol involves several services and characteristics which describe the specific sub-devices and operations that can be controlled by some part of the robot’s public API. Some of the most important services and characteristics provided by BlueBot reflect the Logo commands forward, back, right, left. Using an additional command go, programmers trigger the execution to start.

A given sequence of commands must first be encoded and then sent from the browser to the robot. In the case of BlueBot, each message consists of a sequence of hexadecimal values that each represent one command. The movement commands forward and back are encoded as the hexadecimal values 0x04 and 0x05, respectively. The rotation commands left and right are encoded as 0x06 and 0x07, respectively. The special command go, finally, uses the encoding 0x01. That is, the program fd rt fd lt go is encoded as the sequence 0x04 0x07 0x04 0x06 0x01. This part of the message is extended with five additional fields that are used for subsequent decoding and message verification as illustrated in Figure B.1.

To automatically generate such a message for a given Logo program in XLogoOnline, we use the parse tree traversal process presented in Section 5.1. During the traversal, all basic commands are identified, visited individually, and appended to a dedicated accumulation string. Figure B.2
shows an example of how this idea works for the program `fd 100 rt 90 fd 100 lt 90`. (Note that the XLogoOnline interpreter requires parameters for all basic commands. In the first stage, these parameters are added automatically with each command tile the programmer drags to the input line.) Each command is visited individually and once the the last command has been visited, the entire program is encoded and stored in a message string. The required command go as well as the five remaining message fields (start code, data length, execution mode, frame, and checksum) are added to the string once the end-of-file node (marked green in Figure B.2) is visited.

Any non-interactive sequential program that terminates and which uses only non-parameterized basic commands can be linearized to a simple sequence of basic commands and executed on the robot.
Appendix C

Root Robot is an educational robot, similar to BlueBot. This robot, too, offers an API for external communication partners to steer its behavior. In order to steer the robot’s action via remote control, messages must be exchanged between the client, e.g. XLogoOnline, and the Root robot in the role of the server. Each message encodes exactly one command and has a fixed length of 20 bytes. Two bytes are used for the checksum and as a continuous message counter, respectively. The remaining 18 bytes specify which command is sent by clarifying (i) the device category, (ii) the command type, and (iii) the desired argument. For a motor unit command, device number 1 is used, with command type 8 for a movement command and command type 12 for a rotation. The argument allows to distinguish between forward and backward movement: a positive payload of value $x$ is interpreted as a forward movement with distance $x$ millimeters whereas a negative payload is interpreted as a backward movement. A similar technique allows to distinguish between left and right turn rotations where a positive payload value is interpreted as a clockwise rotation whereas a negative value is interpreted as a counterclockwise rotation. All details about motor-related commands are summarized in Figure C.1.

Pen- and eraser-related commands have a similar structure. They also use one message per command and each message counts 20 bytes in total. These commands use value 2 as their device number with command type 0 (irrespective of whether the command affects the pen or the eraser). Both pen and eraser are steered with the same command and in this case, the payload determines which of the three possible positions the robot uses (see Figure C.2).

Due to the coupling of pen and eraser, certain programs may be executed differently by Root and by the Logo turtle. One such example is the program \texttt{fd 100 pe pd bk 100}. Both the screen turtle and Root start in the same state but they do not end with the same visual result. The classic interpretation of this program ends with a blank screen – there are no lines drawn in eraser mode,
Figure C.1 Root’s message structure is fixed to 20 bytes per message with five structural elements. Device 1 corresponds to the motor unit for which we can specify whether a message is to be interpreted as a movement command or a rotation command via the second byte. The payload contains the corresponding distance or angle where the sign is used to indicate forward or backward movement and clockwise or counterclockwise rotation, respectively.

Figure C.2 For the marker and eraser functionality a second device is used. In this case, both the marker and the eraser fall into the same command and the payload can be used to determine which of the three positions to use.

no matter whether the pen is raised or lowered. When this program is executed by Root, however, the third command \texttt{pd} cannot be executed without also lifting the eraser, causing the final command \texttt{bk 100} to leave a visual trace. The command \texttt{pd} has a side effect on the erasor that cancels the previously-executed \texttt{pe} command.

The presented approach allows the visualization of numerous programs with Logo’s parametrized basic commands, repetition, modularity, and even parameterization by a physical robot. We only just started using Root in schools and there are still few students who worked with this device using our teaching materials. Gathering more experience is left to be explored as part of future work.
Project management, design, and evaluation of a programming environment that is actively used in hundreds of schools. The adopted spiral approach allows the continuous development of problem-solving skills from kindergarten to high school. Several years of experience as a lecturer with participation in training courses for more than 3000 teachers.

<table>
<thead>
<tr>
<th>Work</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PhD Student ........................................................................... 2.2017 – 3.2021</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; ETH Zürich, Switzerland</td>
<td></td>
</tr>
<tr>
<td>Project lead in the development of a programming environment for primary schools (with more than 10 Bachelor and Master theses involved).</td>
<td></td>
</tr>
<tr>
<td>Research Associate ................................................................... 2.2017 – 12.2020</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; PH Graubünden, Switzerland</td>
<td></td>
</tr>
<tr>
<td>Development, implementation, and evaluation of a teaching concept for interdisciplinary promotion of algorithmic thinking in elementary schools.</td>
<td></td>
</tr>
<tr>
<td>Lecturer ................................................................................. since 2.2017</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; PH Graubünden, Switzerland</td>
<td></td>
</tr>
<tr>
<td>Several years of experience as a lecturer at a university of teacher education with students aiming at pre- and primary school level.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Education</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PhD in computer science ................................................................</td>
<td>2.2017 – 3.2021</td>
</tr>
<tr>
<td>&gt;&gt; ETH Zurich, Information Technology and Education</td>
<td></td>
</tr>
<tr>
<td>Concept development, management, and empirical research using a Logo programming environment focused on autonomous learning in a spiral approach from kindergarten to 6th grade. The learning environment has been actively used in hundreds of schools for several years.</td>
<td></td>
</tr>
<tr>
<td>Teaching diploma for high schools ..........................................</td>
<td>9.2014 – 2.2019</td>
</tr>
<tr>
<td>&gt;&gt; ETH Zurich, Switzerland</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; ETH Zurich, Switzerland</td>
<td></td>
</tr>
<tr>
<td>Thesis &quot;XLogo Online - a Web-based Programming IDE for Logo&quot;, (supervised by Giovanni Serafini and Prof. Dr. Juraj Hromkovič)</td>
<td></td>
</tr>
<tr>
<td>Bachelor of Science in Computer Science ETH .............................</td>
<td>9.2009 – 9.2014</td>
</tr>
<tr>
<td>&gt;&gt; ETH Zurich, Switzerland</td>
<td></td>
</tr>
</tbody>
</table>
Project lead

Supervised Bachelor theses

>> ETH Zürich, Department of Computer Science


Supervised Master theses

>> ETH Zürich, Departement of Computer Science


Academic Teaching

Fachdidaktik Informatik I (Guest Lecturer)

>> ETH Zürich, Teaching diploma for high schools, 4 ECTS

Seit 9.2017

Fachdidaktik Informatik II (Guest Lecturer)

>> ETH Zürich, Teaching diploma for high schools, 4 ECTS

Seit 2.2017

Medien und Informatik

>> PH Graubünden, Teaching diploma for pre- & primary school, 4 ECTS

Seit 9.2017

Fachdidaktik Programmieren

>> PH Graubünden, CAS Informatik Primarstufe, 3 ECTS

since 9.2020

Academic Services

Thesis and semester project supervision

Broad experience in supervising student research and semester projects (Bachelor and Master students at the Department of Computer Science as well as student teachers).

Public Relations

In the period between 2011 and 2020, I was able to hold numerous public workshops, school projects, and teacher trainings on programming as a member of the "Ausbildungs- und Beratungszentrum für Informatikunterricht" at ETH Zürich.

Reviewing

Contribution as a reviewer for the ISSEP conference (International Conference on Informatics in Schools: Situation, Evolution, and Perspectives) over three years.

Network of Women in Computer Science (CSNOW)

Promoting the quota of women at ETH’s department of CS through several initiatives:
- Development of teaching materials and implementation of workshops in recurring taster week for female high school students.
- Organization and acquisition of third-party funding for symposium visit to Oxford university with a group of 6 students

### Other Teaching and Workshops

<table>
<thead>
<tr>
<th>Workshop leader at “Schweizer Tag für Informatikunterricht”</th>
<th>2017-2019</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Modularer Entwurf im Programmierunterricht an der Volksschule (2017)</td>
<td></td>
</tr>
<tr>
<td>- Per Zeiteise zurück zu den Wurzeln der Kryptologie (2018)</td>
<td></td>
</tr>
<tr>
<td>- Programmieren ohne Lesen und Schreiben zu können (2019)</td>
<td></td>
</tr>
</tbody>
</table>

**Programming courses and training**

Almost ten years of experience as a teacher in Logo/Python programming with children and juveniles at school. Active contribution to training courses for more than 3000 in-service teachers across the country.

**Expert in Computer Science**

Jury member for «Niklaus Wirth Young Talent Computer Science Award», which awards outstanding performances in semester projects at high school since 2011 since 2017

### Scientific Publications


### Awards

**Women Techmaker scholarship finalist 2016**

Top 50 (Applicants from all Europe, Middle East, and Africa). The scholarship honors outstanding academic achievement, leadership and impact on women in technology