AALBORG UNIVERSITET
STUDENTERRAPPORT

Programming Language for Adolescents

P4-Project
Group SW415F15
Software
Fourth semester at the
faculty of Computer Sciences
Aalborg Universitet
May 27, 2015

AALBORG UNIVERSITET
STUDENTERRAPPORT

4th semester at the
Faculty of Natural Sciences
Software
Strandvejen 12
9000 Aalborg
Telefon 96 35 97 31
Fax 98 13 63 93
http://tnb.aau.dk

Theme:
Design, definition and implementation of programming languages

Title:
Programming Language for Adolescents

Project period:
P4, Spring semester 2015

Project group:
SW415F15

Participants:
Peter G. H. Nielsen
Sasha Junker
Ida Rue Thuesen
Jacob Pedersen
Jens Emil Hansen

Supervisor:
Thomas Kobber Panum

Number of pages: 107
Number of Appendixes: 8 (21 pages)
Concluded 27-5-2015

Synopsis:

This report will detail the development of a small part of a programming language, designed to be used by adolescents, the purpose of which is to engage them in programming via an easily understandable, simple language that offers them advantages in an environment they know and enjoy. During the project it was discovered that a visual output from programming is an incentive for learning, especially for adolescents. As such this is the target group of the project and upon studying various methods of visual output, Minecraft became the target platform of a small programming language which enables a user to write the code that builds structures.

Preface

This report is developed by the group SW415F15 as a fourth semester project at Aalborg University's Software study. The project concerns the development of a programming language for adolescents and the report is therefore intended for anyone with interest/knowledge of the subject, or fellow students. The report is constructed with the purpose of enhancing the authors' experience with writing reports and documenting design and implementation decisions when constructing a compiler.

During the development of the project we worked and shared interviews conducted by group SW411F15 and SW412F15 and we want to thank them, the people who was interviewed, as well as Bent Thomsen and Hans Hüttel, for their cooperation.

Aalborg Universitet, May 27, 2015

Peter G. H. Nielsen
<pghn12@student.aau.dk>

Sasha Junker
<ajunke12@student.aau.dk>

Ida Rue Thuesen
<ithues12@student.aau.dk>

Jacob Pedersen
<jacped11@student.aau.dk>

Jens Emil Hansen
<jeha12@student.aau.dk>

Reading Guide

This section will describe the report's use of references, visual aspects and the content of the different parts of the report.

References

The references used in this report appear as [#], where the number is a reference to a source listed in the Bibliography located before Appendix, in the report. Note that some references are to Appendix, or other chapters, and not Bibliography, these instances are clearly stated within the report. The Vancouver method was used, and therefore the order in the Bibliography is not alphabetical, but rather in the order in which they were used in the report. If the reference is placed before a full stop/period, it refers to the sentence it is used in. If placed after, it refers to the preceding paragraph.

Visual Aspects

Figures, listings, equations, tables etc. are named according to their chapter. E.g. if there are two figures and one table in chapter 3, the first figure is 3.1, the next 3.2 and the table will be 3.1. It is clearly stated within the report which table, figure etc. is being described.

The name of the language developed in this project is Minecraft Language 8 (*ML8*), which will be used interchangeably with phrases such as "the language of this project" when referring to it. Italic font is used to mark when a word or sentence is somehow directly associated with the language or how it is implemented. Note that italic font also occurs when highlighting a word or sentence.

Some parts of the report contains additional information that may not be essential in order to continue to read and understand the report. Such information is encapsulated between two horizontal lines.

The report is divided into four parts that individually handle a certain part of the development process. The four parts are: *Analysis*, *Language*, *Compiler* and *Evaluation*.

Part 1: Analysis

The first part of the report describes the analytic work that ultimately leads to the problem statement. This part is divided into four chapters: *Chapter 3: Teaching Programming* that analyses the ways of teaching Adolescents programming. Chapter 4: *Existing Educational Tools* that describes and analyses the languages Scratch and Dolittle as tools for teaching adolescents programming. *Chapter 5: Elements of a Transitional Language* analyses what makes a language transitional and how these languages are defined in this report. *Chapter 6: Problem Definition* which, on the basis of its preceding chapters, defines the problem statement for this project.

Part 2: Language

The second part of the report focuses on the ML8 language including its design and implementation. This part is divided into two chapters: *Chapter 7: Language Features* which explains the requirements of the language and the features the language possesses. *Chapter 8: Grammar* which contains the description of the methods used to construct the grammar and lastly the implemented grammar.

Part 3: Compiler

The third part of the report explains the stages of crafting the compiler for ML8. There are three chapters within this part: *Chapter 9: Syntax Analysis* that contains the description of the lexer and parser as well as a description and implementation of the Abstract Syntax Tree used for later compiler phases. *Chapter 10: Semantics* that contains the decisions for and design and implementation of a symbol table and the operational semantics and how the transitional rules should define the meaning of the language. *Chapter 11: Code Generation* revolves around the implementation of the code generation with code examples from the implementation.

Part 4: Evaluation

The last part of the report includes the Evaluation which evaluates different aspects of the project. This part contains four chapters: *Chapter 12: Language Limitations* evaluates the compiler and its functionality through tests followed by the section "Errors and Shortcomings" that details limitations of the compiler and language. *Chapter 13: Discussion* and *Chapter 14: Conclusion* evaluate and discuss how, and to what extent, the problem statement was solved as well as specific issues that arose during the development of the language. *Chapter 15: Future Work* describes areas for future work that could be explored.

Contents

Chapter 1

Introduction

The past few decades have seen an exponential growth in the number of software applications and their demand. This increase has given rise to an exceeding need for more software programmers[1]. In part because of this, several organisations, and even nations, have made initiatives for teaching computer programming, or at least aspects thereof, to children and adolescents (in this project, "adolescents" refer to 7th to 9th graders). Programming has, for example, been proposed and is underway to be a part of the Danish elementary school curriculum.[2][3]

A rationale for more programming is that adolescents, who learn the basic aspects of programming, tend to become more creative and better at solving problems, by first understanding the individual parts of the problem and then making a model, mental or otherwise, to help them solve it.[4]

The objective is to attract the attention and focus of these young people in order to make programming an educational, interesting and fun experience for all involved. There are many ways to achieve this, but coding which results in an immediate visual output is an exciting prospect.(appendix 15.3)

An example could be programming a small robot to move in a specific pattern [5], or making modifications to hugely popular video game titles, such as Minecraft [6]. Visual programming proves to be a great point-of-entry for teaching computer programming; visual programming will be further discussed in chapter 3.

1.1 Initial Problem

Pertaining to the developments and arguments discussed above, constructing a platform that can be used for engaging adolescents in computer programming, is a problem of interest. The adolescents who can use an educational tool can be split into new programmers with no initial knowledge of the subject and those with minor prior knowledge either through self-study or previous visual programming education. As described, the interest in getting adolescents to learn and understand programming is increasing, however, it is reasonable to assume that before they are willing to learn, they must first gain a natural interest and excitement about programming and its possibilities.

How can a programming language be developed, to increase the interest in programming for adolescents?

Part I

Analysis

Chapter 2

Teaching Programming

Technology is steadily becoming a greater part of everyday life[7] and it is therefore necessary to teach adolescents what it is and how it works, in order to prepare the next generation[8]. There are reasons why adolescents should gain an understanding of programming. Communication is no longer only text-based, but occurs through several different means, be it sound, video or animation, and even social medias. As these trends get more prominent, a new vernacular is being developed. It is skills and knowledge of areas like these that can be developed and enhanced through understanding programming. [8]

The International Computer and Information Literacy Study (ICILS) is an international test, which reveals that approximately 80% of the students in the Danish 8th grades, have little to no knowledge of programming. This percentage is under the average of other countries that participated in the test. [7]

In order to increase the number of adolescents who have a basic understanding of programming, motivation is needed. This raises the question of how to accomplish this, without being too technical, which often makes the adolescents lose interest. (appendix 15.3)

To support the statements and discover today's challenges in teaching adolescents programming, interviews have been made; see Appendix chapter 15. To get an idea of the two different target groups of teaching programming, two people have been interviewed:

Hanne Kåg (Kåg) is an elementary school teacher working at Kongerslev Skole teaching 7-9th grade programming as an elective course using Scratch and Lego Mindstorms as the teaching platform. See appendix 15.2

Magnus Toftdal Lund (Lund) is a volunteer at Coding Pirates which is an organisation that guides and helps, rather than teaching, programming to 7 to 17 year old children; for a small fee. Their vision is to create product developers rather than programmers. The teaching platform is chosen by the child, depending on what motivates them. See appendix 15.3

According to Kåg, forcing adolescents to learn programming is not the solution. In her experiences it is helpful if the output is instant and visual, because it motivates the students to do more, but the time they get to learn is limited (appendix 15.2). In the interview with Lund, he states unsolicitedly: "*You learn to program because you can see some use of it. The children might think: 'Aha, Minecraft, I can tinker with that' - and the children are off to program*" (appendix 15.3) which suggests that video games, hereafter referred to as "games", could be an approach to teaching adolescents programming.

When comparing the two interviews, a difference between the motivation of the adolescents is noticeable. Where the students in an elementary school (appendix 15.2) might be less motivated, the attendees of Coding Pirates (appendix 15.3) depend more on self-motivation. The formal school curriculum has to be approved by the proper authorities which limits the approaches of teaching them programming, no such requirement exists for Coding Pirates as it is based on volunteering.

Gathered from the interviews, fun and learning are closely related and it is important that the adolescents are not forced to learn. The target group of the project should be self motivated adolescents, but to be motivated, there has to be some concrete use for the programming, in a context they enjoy.

It has been shown[9] that learning through games can motivate adolescents far more, than through ordinary means of teaching. The research of Constance Steinkuehler, associate professor of digital media at the University of Wisconsin-Madison, suggests that boys tend to have reading skills that are a couple of grades below the average of their school level, whereas girls are more skilled readers in general[10]. However, the research also shows that the boys in question were significantly better readers when the text was part of a game. The study showed that when the boys could choose what to read, as they could with games, they pushed themselves harder. [9] By introducing programming in the context of games, the boys could pursue programming, and learning, in an environment in which they are already motivated.

On average, more than 50% of the students in an 8th grade class in a Danish elementary school play games regularly and only 18% never play games[7]. This could indicate that introducing programming in the context of games, could be a valid method for teaching adolescents programming.

In connection to games, the possibility of modification, or in other ways increasing or changing the experience of video games, could be one aspect of

programming that could entice adolescents. Another aspect could be through visual- or physical output. Along comes the need for immediate output and response to the actions or changes that are made.(appendix 15.3) Future educational tools for programming (hereafter referred to as educational tools) have to support these criteria, if they are to be effective.

Chapter 3

Existing Educational Tools

An educational tool is in this report defined as a tool that can help people get started in programming by using a tool that provides fast feedback and functionality without much work.

There are two different approaches when it comes to programming: Visual or text-based programming languages.

The meaning of a visual programming language, is to let users manipulate program elements graphically rather than type them textually. This means that the blocks for these program elements is built in advance and the user can plot in the desired elements. For example a block could represent an if-statement, which always requires the user to set a boolean expression, a body and perhaps else statements.

In contrast, text-based programming is programming as thought of in its usual sense: Writing the keywords, statements, expressions and other commands in a text editor, according to syntactic and semantic rules to construct some program. Instead of plotting in a block of the if-statement, it will here have to be explicitly written.

Several languages, the purpose of which is to teach adolescents the concept of programming languages, already exist. Some of these are: Scratch, DoLittle and PyGame which are based on different types of programming. The concepts of Scratch and DoLittle will be explained in the following sections.

3.1 Scratch

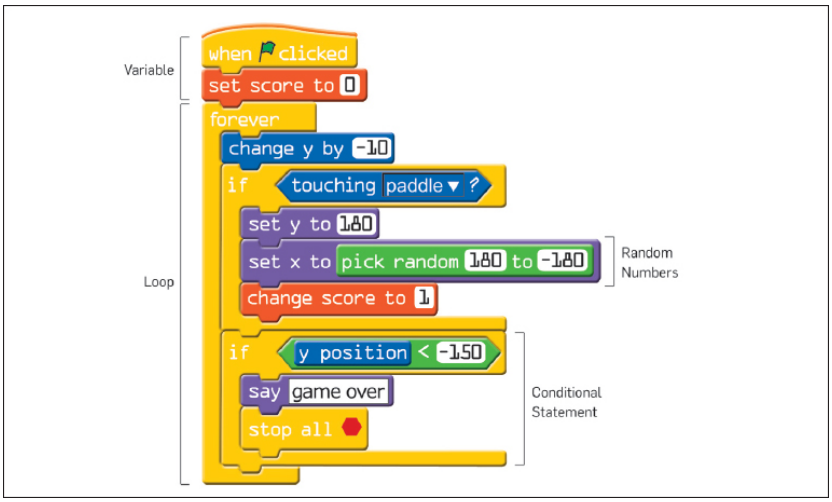

Figure 3.1: Example code in Scratch. [11]

Scratch, an example of visual programming, is a programming language, developed by a team at MIT, in order to present a way of programming to people who haven't previously imagined themselves as programmers. [11] The language, being very accessible, is often used to catch the interest of children. [12]

Programming in Scratch consists of assembling pieces that each represent actions or more basic programming principles, like loops. The actions can affect elements, such as sprites, to make them move, speak through speech bubbles, play sounds and more. In figure 3.1 an example of how programming in Scratch can be done, is shown. The different blocks are shaped in such a way that they only fit with other blocks where a connection between them makes syntactic sense. This makes sure that the program is executable as long as the blocks fit. This way of programming provides a more intuitive and less technical way of programming which can appeal to beginners.(appendix 15.3)

Many users experience a lack of functionality as they get better at programming. They feel that Scratch is missing some blocks, limiting their creativity.

Although the language being developed in this project might be textual rather than visual, the keywords in the language may need to cover more aspects, than they do in languages like Java and C#. This inspiration of fewer constructs with a broader functionality, can come from scratch, which is already encapsulating code in blocks.

3.2 Dolittle

In 2002 a group at the University of Tsukuba developed a text-based, object-oriented language: Dolittle. Consequently, the Dolittle programming language was tested in practice in the Japanese education system. The focus of the language was to be able to control robots making it possible for the students to understand and see the result of their work.

According to the developers [13] some of the design policies when developing the language was:

- It was deemed important that the language had a simple syntax in their native language, so that the students could use their time efficiently learning programming instead of using a lot of time learning the language itself.
- They moved away from the abstraction of classes, functions and variables to make it easier for the students to understand. It is then possible to write a program on only one line and it would still be functional.

Using that design policy the finished language ended up looking as shown in figure 3.2. The statements are ended with a period and functions called by using an exclamation mark.

```
カメ太＝タートル！作る。
「カメ太！100 歩く 120 右回り」！3 繰り返す。
三角形＝カメ太！図形にする（赤）塗る。
時計＝タイマー！作る。
実行ボタン＝ボタン！"実行" 作る。
実行ボタン：動作＝「時計！「三角形！36 右回り」実行」。
```

```
kameta=turtle!create.
[kameta!100 forward 120 rightturn]!3 repeat.
tri=kameta!makefigure (red) paint.
clock=timer!create 1 period 10 duration.
rBtn=button!"Run" create.
rBtn:click=[clock![tri!36 rightturn]execute].
```

Figure 3.2: Top part showing the original Japanese code and the bottom part shows the english translation.

The language has a built-in GUI where the user can see the objects they make and control them.

Surveys with Dolittle

A small-scale test was initially made with a group of high school students. They had 3 one hour lessons where they learned the basics of Dolittle and tried writing small pieces of code. From the lessons, they learned that the computer used sequential execution. They also got an understanding of how visual representation on a screen is done via pixels and an understanding of the existence of an operating system and its peripherals.

A large scale survey was also conducted on 132 junior high students. They had regular classes in computer science where they used the Dolittle programming language and finished with an exam testing their acquired knowledge in Dolittle. From the examination they could conclude that more than 85% of the students had understood all the basic programming concepts in the Dolittle language. It was also noted that many students liked the idea of programming GUI buttons that could execute an action on being pressed.

At each lesson the juniors were asked to fill out a questionnaire about their experiences relating to the lessons. The results show that at the beginning 20% found the course very enjoyable, which grew to 40% by the end of the course, and another 40% found the course enjoyable. About 70% completed all or almost all of the challenges in the course and only 15% found the course very difficult by the end.

3.3 Language Evaluation

Scratch is a language that is used to introduce beginners to programming by using the visual aspect both in the programming phase and as the program's output. With focus on the visual part, Scratch makes for a suitable beginners language to programming, especially with programming being done in blocks that only fits when the result makes syntactical sense.

Dolittle is based on textual code and the output of the program can be visual. The approach of having a close-to-native language makes for a suitable beginners language to programming. It is interesting to note that the amount of students who found programming enjoyable by the end of the process were greater than at the beginning, this could be due to the language design of DoLittle and indicates that the language is suitable as a motivation for programming. The idea of visual feedback in the form of a GUI button, was appreciated by more students. It also indicates that the approach formulated in their design policies; simpel syntax in a well understood language and hiding the concepts of classes, functions and types, had a positive impact on the junior high students learning ability and enjoyability while learning.

From the Language Evaluation Criteria[14], readability is related to a languages simplicity, orthogonality, data types and syntax design. For a language to be simplistic it must not have so many constructs that programmers only learn a subset of them and it has to limit the constructs to not accomplish the same operations. It also has to limit the ability to use operator overloading, which can change the functionality of simple operators in the language. Orthogonality is related to the number of ways the control and data structures in a language can be used. A high orthogonality helps with the readability of the language. With a high orthogonality constructs in the language behaves the same in different contexts of the program, meaning there are few or none exceptions to the functionality of a construct no matter where it is being used. For there to be adequate Data Types in a program, it must have the needed data types related to the context the language is being used in. As an example, if the language represents an environment where logic calculations are done, the language should have a boolean data type to maintain a high readability. Syntax Design is also important for readability. The concept of reserved words helps with readability, preventing the programmer from using the same words as those used to define the constructs in the language e.g. not allowing the programmer to use "for" as a variable name if it is used to construct a "for"-loop. Another example is reusing reserved words in different constructs, which also degrades the readability of the language.

Looking at Dolittle it is very simplistic in its constructs, there is only one way to use any given construct giving it a good orthogonality, it has adequate data types to specify movement, time and degrees and they use unique reserved words in different constructs. They managed to construct a language with a high readability using their initial design policy. Scratch also maintains a simplistic structure of having relatively few blocks, with a broad functionality, that fits together visually. As the language has its own visual output environment, it was developed from the beginning to represent and manipulate all aspects of that environment and has a good representation of data types to accomplish that. Even though Scratch represents constructs as blocks and therefore doesn't have any concept of reserved words, the blocks themselves clearly differentiate the structures and the user written code. The result being that an "if"-block will never be misread as a variable called "if", because that variable is another block. This results in Scratch having a high readability.

Chapter 4

Elements of a Transitional Language

In order to create a transitional language between basic and complex languages, it is necessary to understand what makes these languages basic and complex and what causes problems for adolescents when learning the complex languages.

Dolittle and Scratch can be defined as *basic languages* because they only support visual and instant output. They both have data types, but the user does cannot explicitly use, declare or define them. Both languages use only text, or symbols, from the Latin alphabet. Dolittle uses a period to indicate a statement or declaration instead of the semi colon, which is often used in complex languages. Scratch uses known words, or sentences, e.g. *forever* instead of *while(true)*. The languages also support implemented key sentences, or words, that in themselves have no meaning in other languages, and cover multiple functionalities which, in other languages, would be implemented via libraries. In example of this in Scratch is the block: "say ___" that makes a sprite say something using a speech bubble. In Dolittle, the word *Turtle* means: "Make a sprite and place it in the middle". The other languages mentioned, covers high level languages like such as C++, Java and C#, and will be defined as *complex languages*. These require the user to implement almost everything by hand thus provides the user with freedom in composition, but makes the languages impractical to learn for beginners as the code becomes much more complex to write, making the user more prone to make errors.

Creating a transitional language between the basic and the complex languages is interesting, because no such language exists (appendix 15.3). Adolescents wanting to learn a complex language usually have to do so through trial and error, accompanied by reading material and/or teaching. Ideally, there should be a language that uses the knowledge from the basic languages and then adds more concepts on top of it such as types, classes or functions. It can be discussed whether including a complex language in the school schedule will work or not, since there are a variety of elements that students do not know about, which may make the experience overwhelming for beginners.

Basic languages cover concepts in simplified form, like variables, expressions, loops, functions and image manipulation. The basic languages cover the above mentioned concepts, but most complex languages entail classes, data types, data abstraction, libraries etc., as well. Creating a transitional language from basic to complex languages should prepare the user for the new concepts that are not taught through the basic languages. A transitional language should introduce elements from the complex languages in a simpler way, attempting to make it easier to understand for adolescents. Classes are not part of all complex languages, but in the languages where they *are* found they are usually an essential part of the language. Data types make it possible for the user to see what type a certain variable is, which may be important in some cases. It can, however, also be complicated to learn the difference between them and understanding why they should be typed. Data abstraction is only completely possible when classes are present, but can be beneficial if a certain type does not exist in the language and is needed in the program. Libraries can be beneficial for the user if extended code, or functionalities, are needed, but this can be complicated to learn and understand when the concepts of types have not yet been learned.

Some important aspects regarding the more technical aspects of a language and its constructs were extracted from the interviews with Lund (appendix 15.3) and Kåg (appendix 15.2). They will primarily be used in the design phase for the project.

When asked about motivation for programming and why visual output is important Lund says that although it shouldn't be a competition there are prestige in showing what you have made and this motivates them.

He also states that English isn't a problem for the children, rather they feel strongly for the language when asked about how to represent keywords in the language. Contrary to Lund, Kåg notes that on top of having to learn how to use mathematical symbols and technical terminology, having keywords in English does add to the complexity of learning the language.

Lund was further asked about if mathematical symbols e.g. "=" are easiest to understand or if words should be used to describe operations should be used instead. According to Lund they already know the symbols from math and using keywords would only lead to confusion. Difficult concepts in a language are the concepts of variables, lists and loops according to Lund because they are very abstract concepts. Additionally to Lund, the survey of 11 students from Karensminde also found that the majority thought that loop constructs were difficult to understand15.1. Further the concept of types and how to use them are hard to learn for the students.

Through the previously mentioned interviews it could be gathered that functions, classes and data types were especially difficult to grasp for adolescents. The problem with functions and classes indicates that abstraction is a difficult concept when learning.

Chapter 5

Problem Definition

During the analysis there were several different issues discussed which will be recapitulated in this chapter.

In chapter 2 the target group of the project is chosen to be intermediate, or self-motivated adolescents rather than beginners. The reasons for this are the difficulties in teaching non-motivated beginners programming. This also means that the language will be functioning as a transitional language. It has additionally been discussed whether or not the language should exist within the world of a game, "in-game", or as a standalone program, but it was found that the best motivation for adolescents is visual and instant output, preferably through games as it increases the motivation and interest. Therefore the language should exist in-game.

In chapter 3, the two languages Scratch and Dolittle were discussed with focus on the pros and cons. It was found that the developed transitional language should be text-based to facilitate a better transition from basic to complex languages and that the language should be focused on visual and instant output. Inspiration can be drawn from the code that is encapsulated in the blocks of Scratch to create keywords in the language developed in this project. It was also found that the high readability of Dolittle proved to be a success when testing the language on junior high students. Scratch also proved to have a high readability through its visual approach to programming.

In chapter 4 the notions of basic- and complex languages were defined to underline the definition of a transitional language. The main problems for adolescents to understand were classes, functions and data types, but since classes are only relevant for some complex languages it should not be necessary in a transitional language. The language, however, should support simple data types and functions to prepare the user for complex languages.

5.1 Problem Statement

The analysis reveals important factors for creating a good programming language. Some of these factors include that the language:

- Is a textual programming language (chapter 2).
- Provides immediate visual output (chapter 2).
- Is a transitional language (chapter 4).
- Functions in connection to a game (chapter 2).
- Has focus on readability (chapter 3).

Which leads to the following research question:

How can a transitional, text-based programming language be designed to function in conjunction with a game, such that it inspires adolescents to learn and engage in programming?

Part II

Language

Chapter 6

Language Features

In this chapter, the framework of the language called ML8 will be discussed. The chapter is divided into two parts: Requirement and Syntax. The first part describes the requirements of the language and then looks at how those requirements can be fulfilled in relation to a choice of platform and finally how what impacts that platform has on the language. The second part looks at the different constructs the language has to be build upon to fit the requirements of the problem analysis and the chosen platform.

6.1 Requirements

For attempting to motivate adolescents, chapter 2 discovered that games are a possible platform for learning programming. Furthermore, it is mentioned that a transitional language is useful when working with adolescents, especially because of difficulties when dealing with abstract features. It is also mentioned that the learning environment must feel natural to the user in both the programming- and execution phase. This is due to the fact that the focus should be on learning the possibilities programming can provide for solving many different problems, and not specifically to become familiar with a single new platform or an advanced integrated development environment (IDE). This leads to the conclusion that it would be beneficial if the target platform is already known to the adolescent.

The target platform should also be open for modification, usually referred to as modding in this context, if that were not the case, it could be difficult to make an actual implementation thus rendering the designed language useless. The accessibility of a game related platform is dictated by the developer of the game. Some developers intend for their games to be modded, and distribute free tools to do so, while others express that any modifications to their game is a breach of the End User Licence Agreement (EULA). Modifying, is changing any visual or mechanical aspects of the game in any way not originally intended by the developer. Therefore, an accessible platform is considered to be an important factor in the design of the language, as well. From chapter 2 it was gathered that it was important for the learning process to be in a context they enjoy.

To summarise the requirements:

- The target platform should be a game.
- The language should function as a transitional language.
- The language should be targeting a known platform.
- The platform should be available for adolescents.
- The platform should be open to modding.
- The platform should support the statement of being fun while learning.

Based on these requirements the following paragraph discusses the choice of platform.

Choice of Platform

As stated in the previous section the target platform has to be a game and it would have to be a game adolescents would be familiar with. There are a broad variety of games available such as those mentioned in table 6.1, but with the requirements in the previous section the field is narrowed down.

In the table, play-% is the amount of time the game has been played from a total of all the compared games (that is; not all games are included and therefore not adding up to 100%). Genres are described in this section. ESRB ratings are translated into equivalent age, 13+ means recommended minimum age of 13.

Name	Play-%	Genre	Mod-able	Age rating [17] [18] PEGI	ESRB
League of Legends	19.99%	MOBA	no [16]	12+	13+
World of Warcraft	15.61%	MMORPG	no [19]	12+	13+
DOTA 2	5.25%	MOBA	no [20]	-	-
Counter-Strike: *Global Offensive*	3.66%	FPS	no [21]	18+	17+
Smite	3.29%	MOBA	no [22]	12+	13+
ArcheAge	2.05%	MMORPG	no [23]	16+	17+
Minecraft	1.85%	Sandbox	yes [24]	7+	0+
Diablo III	1.73%	ARPG	no [19]	16+	17+
Hearthstone: *Heroes of Warcraft*	1.43%	CCG	no [19]	7+	13+
Dragon Age: *Inquisition*	1.42%	ARPG	no [15]	18+	17+

Table 6.1: The table shows which PC-games as of December 2014 were the 10 most played[25].

For most of the games it is important to note that modding is not allowed. For some of the games it is not strictly prohibited but any person or company has to apply for a dispensation in order to create content based on the game. One common expression in the EULA is similar to: *"You may not modify EA Software or use it in any way not expressly authorised in writing by EA."* [15], but a problematic situation is that due to the competitive element in some games it is often formulated as it is in League of Legends *"No software should interfere with the League of Legends player experience between when you press Play and the end of game screen."* [16] as they are concerned that some players might achieve better performance due to such modification thus resulting in an unfair game. This results in the competitive games to be a less attractive choice as target platform because both legal and practical elements have to be considered. When this is considered at the same time as the age group of adolescents and the guidelines presented by PEGI [17] and ESRB [18] the choice for target platform would be Minecraft, as this game is not restricted to any part of the target group (adolescents), is allowed to be modded and is very popular (no. 7 of most played PC-games December 2014).

Minecraft[26] has an open and malleable game world and the content of the game has the lowest minimum age requirement of all the games on the list 6.1 featuring the top ten most played PC-games as of December 2014. In the table 6.1 the top ten most played PC-games are shown with information about some of the requirements provided in the previous section. As Minecraft supports all the requirements in the previous section the choice for target platform is Minecraft, although there might be other candidates, a specific choice had to be made in order to progress to specifying the language further, but it could have been any other game fulfilling these requirements.

Purpose of ML8

A lot of time in Minecraft is spent doing repetitive tasks, such as mining for materials and placing one block at a time when building, see fig. 6.1 as an example.

Figure 6.1: Example from a tutorial on how to create a building in Minecraft. Much of the process is repetitive, as each block is put in place one by one to construct the virtual house. [27]

The world is a three dimensional grid and the blocks in-game are to simulate a cubic meter in the real world (1 meter by 1 meter by 1 meter), see figure 6.2. They span over a wide variety of types (e.g. dirt, stone, birch wood etc.).

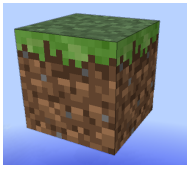

Figure 6.2: A single block of dirt with grass as it looks like in Minecraft. This is an example of a block that can be used to build with.

The webpage for Minecraft[28] states: *"Minecraft is a game about breaking and placing blocks. At first, people built structures to protect against nocturnal monsters, but as the game grew, players worked together to create wonderful, imaginative things."* and as such the constructive part of the game is a central element. But the way it is carried out in the game is not very effective. Most of these tasks are repetitive, time-consuming and could be achieved through programming. This fulfils the criterion of giving the user a more efficient game experience while learning to program.

The purpose of ML8 is to design a meaningful and working programming language that interacts with Minecraft. The language has to enable modification of elements in the game and allow for the user to program whatever they might find interesting, regardless of it being placing a few blocks or building a structure, such as a house. The language facilitates the building process and at the same time gives adolescents the possibility to learn more about programming. The programmed structures can be reused and placed elsewhere allowing the user to e.g. make a village using the same structure at different locations.

6.2 Syntax

From a technical standpoint the language has to enable modification of elements in the Minecraft world and it has to have constructs for representing and manipulating these elements. But looking at the context of this language related to the adolescents who will use it, other aspects also have to be taken into account when choosing what constructs this language will be made of. Those aspects include that it has to be a transitional, textual language and the focus when constructing the language must be on readability, as described in section 3.3. Different constructs will now be looked at in relation to the project and the Minecraft platform.

Functions

Through the analysis it was found that notion of functions is a difficult concept for adolescents (section 5.1) and when looking at Dolittle, the designers chose not to include the construct entirely. But as this will be a transitional language the most common constructs used in fully fledged languages has to be represented in some form to ease the adolescents from the basic to more complex languages. With that in mind, limited functionality with functions needs to be included. A visual approach to functions could be useful in the programming language. An example of this could be to separate the functions the same way classes are naturally split into independent files in the Java language, thereby keeping each function separated.

Classes

Object oriented languages were made to represent real world objects directly as individual objects in the language. And although Minecraft is a virtual world it still has objects, like actors and blocks. Therefore in regard to Minecraft making the language object oriented would be a valid approach. But with objects comes a lot of abstraction, functionality and new constructs. Some of the complexities come from how the classes and objects interact with each other through e.g. inheritance and polymorphism. Every class is also a type that adds to the complexity of the language. Without any of the extra functionality in the object oriented language objects themselves doesn't add anything to the language. To follow the problem analysis (chapter 5.1) on readability, classes and objects should not be included in the language if it is possible to represent Minecraft in a meaningful way without them.

Keywords

Keywords are related to the readability of the language through the syntax design. To get a high readability the language has to be designed with reserved words that can only be used for a specific construct and not in any other context. Furthermore, using unique words for every construct also helps with readability e.g. instead of using braces as C, C++ and C# does, ending a do-loop body with "enddo" specifies that this is the end of the "do" construct, while "endfunction" indicates the end of the "function" construct and thereby preserving a high readability.

In the basic languages all reserved words and keywords are strictly lower cased for simplicity and ease of remembering, as seen on the figures in chapter 3. This, in turn, means that there are fewer options for keywords or reserved words, and some variable names become unavailable for the user to use.

In complex languages the lower and uppercase letters in a word are defined by the category they are in. As an example, the words "string" and "String" mean two different things in C#. If it starts with an upper case "S" it indicates that it is the String class, a reference type object inheriting from the Object class. Lower case "s" means that it is a data type. The benefits of this is that more words can be reserved or used as keywords, but it doesn't help with readability.

The language should only support building in Minecraft, which means that the required amount of keywords and/or reserved words will be fewer than if it was a language for programming in general. This limits the problems with choosing the lower case letter keywords and since the language should be more about programming than remembering, it is more practical to use lower case letters for keywords and reserved words. The final language ended up having the keywords &, as seen in figure 6.2.

function	endfunction	call	times
place	step	do	enddo
using	return	origin	if
then	elseif	else	endif
+	-	*	/
&	|	up	down
forward	back	right	left
int	block	bool	go
void	true	false	(
)	.	=	;
<	>	<=	>=
==	!=	/*	*/
//			

Table 6.2: List of reserved keywords

Natural Language

Since this is a project targeting adolescents in Denmark, who are taught English early on, the keywords in the language can either be Danish or English. If written in Danish, younger adolescents, or those struggling with English, will not have the added complexity of having to master an already difficult 2nd language in order to code, which for some proves to be a problem (chapter 4). But adolescents usually have a high understanding of English (chapter 4) and Minecraft is in English, therefore that language would fit more seamlessly in the game world, which would also avoid the translation of in-game words for objects like dirt, stone etc. Some adolescents also claim that it is more satisfying to show their friends and family that they not only wrote a program, but that it was in English (chapter 4).

Since English is not a problem for the adolescents and it should function as a transitional language, while most complex languages are written in English the ML8 language should be as well.

Symbols and words

In most programming languages, many operations are denoted by symbols, derived from mathematical signs, such as "==" for "is equal to" and "=" as the assignment character, as well as a host of other arithmetic and logical operators.

Adolescents understand symbols better than words (chapter 4) and the problem with using words, is that they are ambiguous and the use of the word can therefore be misunderstood. Symbols are unambiguous and many symbols serve similar functions in several programming languages. Students will already be familiar with most of the symbols, mainly from mathematics and should quickly learn how to use them when programming (chapter 4). The basic languages use words instead of symbols, but since this new language is to be a transitional language, and most complex languages use symbols, the language should use symbols instead of words.

Loops

Complex languages support different kinds of loop types. In C#, for example, there are *for*, *foreach* and *while* whereas Scratch has *forever* and *repeat* which are equivalent to *while(true)* and *for* respectfully. The benefit of having different types of loops is that they help with writeability[14] through ease of expressivity, in that it helps prevent cumbersome coding with more specialised structures for each computation. This again helps limit the code needed to write a program. Although this is an interesting concept when developing a complex language, with optimal efficiency when programming, from gaining a higher writeability, it worsens the simplicity of the language and is therefore not a wanted functionality in a transitional language, that should maintain a high readability.

From questions posed for the 11 students in the elective course, it was also found that the adolescents found the concept of loops such as *for* and *while* (chapter 4) difficult. This again enforces the choice of minimising the amount of these structures when constructing a transitional language, so the adolescents are presented with the concept of loops while only having to learn one construct. Therefore, the language should only contain one loop type that is usable in any construct that needs a loop.

Typing

Dynamic typing, from the user's perspective, means that the type of a variable does not have to be specified when declaring variables. The benefit is that they do not need to be concerned with types and it is easier to understand (chapter 2) why to write "a = 5" rather than "int a = 5". Another benefit is that one does not have to think about what kind of variable is needed. For example it is not needed to specify that "a" is an int that has the number "5" assigned; writing "a = 5" will suffice. But the downside of this is that the dynamic typing makes it difficult to see what type a variable is. For example, if a function takes two input parameters and the functions signature merely states "a" and "b" as parameters, one needs to examine the code in order to see the types of these parameters.

Also of note is the earlier detection of errors with static typing. With static typing, it is possible to catch typechecking errors at compile-time rather than at runtime. This is an especially important aspect when making a compiler that compiles to an intermediate language that is then run inside a game platform. This delays the time between making an error and detecting it when running the code inside the game and does not correspond to the criteria of having instant feedback as described in chapter 2.

Since most complex languages use explicit typing, it is practical for the users of the transitional language to learn this concept and explicit typing provides a better overview of the code and the workings of functions within.

Datatypes

The readability of a language is partly maintained by having the right constructs to represent the world that the language is made to express, called data types. In this world, Minecraft, some constructs exist, as shown in table 6.3.

Type	**Definition**
block	Building material or blocks
bool	Logics, e.g. is a block present at some location, true or false
go	Directions, in the 3-dimensional world
int	Integers, e.g. placing a specific number of blocks
void	And a way to specify a no-type

Table 6.3: List of data types

Floating points/decimal was another possibility, but due to Minecrafts 1-block pr. tile setup it is not needed in the language and would not help readability. Looking at the background of the adolescents (7.-9. grade primary school) they have only been introduced to simple trigonometry and not the broad usage of sine, cosine and tangent thus limiting the usefulness of floating points[29]. One consequence of not having decimal numbers are that any uneven calculation will automatically be rounded to the nearest whole number, this has the positive side-effect of preventing the adolescents from trying to place half a block or moving 2/3 of a block in a direction before tying to place a block, which in both cases, is impossible within the Minecraft environment.

Orientation

In-game the player can place and remove blocks in 6 directions (north, south, west, east, up and down). When programming, these directions should be defined. There are two options: One, the orientation is always the same, when calling a function. Second, that the orientation changes during the execution of the function.

The first option makes it easier to draw on a piece of paper and then write the code down following the desired path of construction. E.g. if the function is executed facing north, then the keyword *right* will always place or move towards east, while maintaining the north direction.

Whereas the second option requires the programmer to always keep the current orientation in mind. E.g. if the function is executed while facing north, then the keyword *right* will provide the player with a new orientation; east.

The second option also makes it difficult to read and debug the code, because the keyword *right* means something entirely different if the direction is north than when it is west.

In order to simplify orientation when building with the language in Minecraft, it was chosen to have objects always be built relative to the perspective of the player (first option), this way e.g. a house would always be build in front of the player - unless stated to have another direction (e.g., right) - thus all directions are needed as 'special words'.

6.3 Chapter Conclusion

In chapter 3, visual information was found to be most desirable when teaching programming to adolescents. Thus it was found that a target platform was needed. The platform had to be a known game for adolescents and it had to be open for modification. It should also support being fun while learning. From these requirements the Minecraft platform was chosen. From the Minecraft open malleable world of blocks and the repetitive tasks done when playing the game a possible entry for the language was found. Based on the project problem analysis, section 5.1, and with Minecraft as the platform the constructs the language needs to have, were found:

- Limited functions.
- Reserved words in lowercase with focus on uniqueness.
- Keywords in English.
- Symbols derived from mathematical sign.
- Only one loop construct.
- Static typing.
- One direction throughout the execution of the function.

In order to create a grammar for ML8, a choice of parser must be made, as this have a direct impact on how the grammar is written. The next chapter elaborates further on this.

Chapter 7

Grammar

The composition of the grammar needs to be adjusted according to the chosen parser, as different parser methods will recognise different grammars. The choice will be between two different parser methods; top down- vs. bottom up parsing. These will be compared to determine which method is most appropriate for this particular project. This is followed by examples from the grammar, explaining the syntax. Lastly, the follow sets will be found for the grammar.

7.1 Top-Down Parsing

Top down parsing, also called Recursive descent is a method used for top-down parsers that follows the production rules of the formal grammar using a recursive procedure[30]. In other words, recursive descent is an algorithm for parsing an expression using the formal grammar.

As the algorithm proceeds it builds a so-called parse tree, most commonly an abstract syntax tree, of the input strings tokens and their relations as dictated by the grammar of the language.

7.1.1 Predictive Parser

The predictive parser can predict which production to use. A prediction of the production rule is only possible for the LL(k) grammar class. The LL(k) grammar class is a subset of the context-free grammars.

LL-Parser

LL(k) is an abbreviation of **L**eft-to-right, **L**eftmost-derivation, where k is a variable for the number of lookahead investigations. Left-to-right means that the grammar productions are examined from left to right, for each non-terminal, and leftmost-derivation means that the first non-terminal in the sentence, produced by the production, is the first to be expanded upon. The number of lookaheads is the number of tokens the algorithms uses for determining the correct production, i.e. k is a number that determines the number of tokens that has to be examined before the production rule is chosen.
The prediction depends on the next token that appears in the input buffer. If the buffer's next token exists in the defined grammar, the parser will be able to predict which of the production rules to choose next. The advantages of LL(k) parsing is that it does not require backtracking.
LL(1) is the algorithm that is used practically; LL(1) means that the number of "lookahead" tokens is one.

The grammar rule is a context-free grammar. Therefore any production can be replaced regardless of context. The non-terminals can be replaced recursively in this sequence of grammar.[30]

Example

In listing 7.1 an example of grammar is shown.

Listing 7.1: Example grammar

```
1  decl         : TYPE? ID ('=' (andOrExpr | MATERIAL))? ;
2
3  andOrExpr    : boolExpr (ANDOR boolExpr)* ;
4  boolExpr     : boolOpr (COMPARATOR boolOpr)* ;
5  boolOpr      : ID | NUM ;
6
7  TYPE         : 'int' | 'block' | 'bool' | 'go' | 'void' ;
8  ID           : [a-zA-Z] ([a-zA-Z0-9] | '_' )* ;
9  NUM          : '-'? [0-9]+;
10
11 ANDOR        : '&' | '|' ;
12 COMPARATOR   : '<' | '>' | '<=' | '>=' | '==' | '!=' ;
13
14 MATERIAL     : "ID";
```

Consider the following input string: *bool* $a = 7 > 5$.

From the grammar it is possible to examine how the parser would traverse the grammar in order to determine if the input string is accepted by the grammar. This is shown in figure 7.1

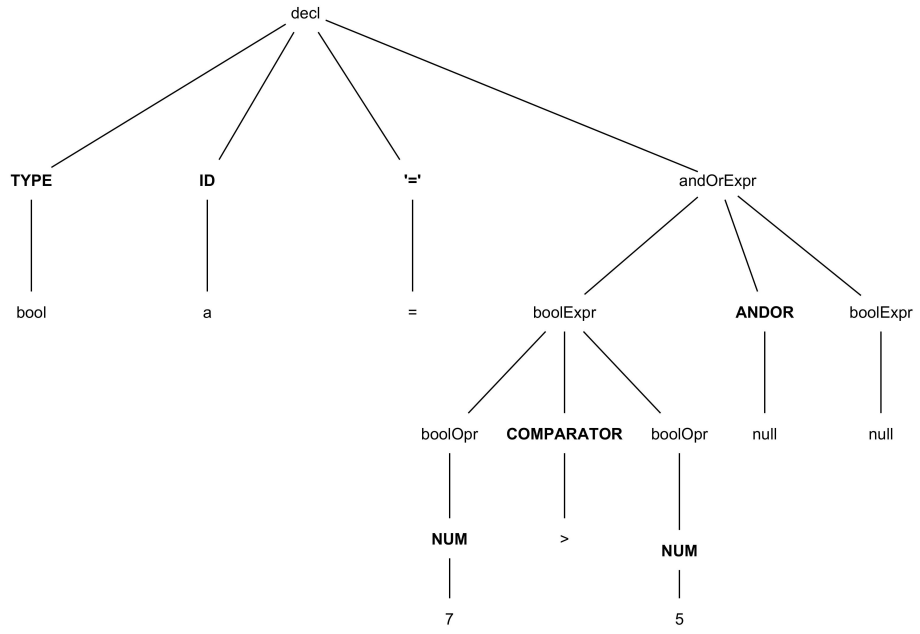

Figure 7.1: Parse tree for bool expression.

Figure 7.1 shows a parse tree of the grammar with the string, which gives an idea of how the LL parser recursively checks if it is an accepted string.

As shown in figure 7.1, the first three tokens: bool, a and '=' are simple to parse, because the first three terminal productions/tokens are determine that if these exist, they are in this order. The andOrExpr is more complex to parse. The parser has to check if 7 is a valid token. First it has to go through boolExpr, then boolOpr and then NUM, becuase 7 is defined as a number. Then it is the $>$ operator. Now we are in NUM, but there is not anything left to do, so it goes back through boolOpr to boolExpr where it checks if the next production i.e. OPERATOR fits. If it did not, the parser would go back to andOrExpr to check if it did there and if not go back and see if MATERIAL did. If not, then the string would not be accepted. But since the OPERATOR accepts $>$, the last production in boolExpr is boolOpr and under NUM it finds 5.

7.2 Bottom-Up parsers

An alternative to the Recursive Descend parser is the Bottom-Up parsers. These are commonly called LR(k) parsers, or just LR for short, where k describes the lookahead, similarly as was the case with the LL(k) parser. This type of parser can handle the largest class of grammars. A LR(k) parser starts from the leaves and works its way up toward the root. Instead of expanding a productions LHS (Left Hand Side) with its RHS (Right Hand Side) as the Top-Down parsers do, it does the opposite and instead reduces the RHS with its LHS symbol, until it reaches the start symbol.

LR-Parser

The LR parser starts with an input of all the terminal symbols and constructs a parse tree using a second stack and two operations called Shift and Reduce. To be able to determine what operation to use the LR parser needs a table of all possible states and all the symbols in the grammar. Within the table the engine can lookup what action it can take for a specific state and symbol. Because the table has every state and symbol in the language it grows exponentially as the complexity of the language increases. The structure of the LR parsers generally make them slower, less obvious, and harder to hand-modify than recursive descend parsers.[31] Some techniques have been implemented to try and limit the growth of the table e.g. SLR and LALR parsing methods.

When the parser and corresponding tables have been created the parser is ready to accept an input string. The parser engine starts by pushing symbols onto the stack and going through states using the Shift operation. When enough symbols have been shifted to make a valid reduction the Reduce operation is done. The LHS symbol, corresponding to the valid grammar rule for the reduction, is placed back on the stack with the reduced symbols prepended. Now the engine starts shifting symbols again until another reduction is possible.

The following example is made using the part of our grammar for a statement production into a step command with a possible increment either as an ID or a NUM. Before constructing the parse table the code is rewritten in CFG form without the extended BNF notation:

As the example grammar is very short the following parse table is made by hand. Looking at all possible derivations, states for the grammar are constructed, and for every symbol an action is filled into the now created parse table. Any field corresponding to a state and a symbol without an action filled in will return a parse error. The finished parse table is shown in table 7.1:

To test the validity of the parse table we try to parse the string: " step 42 forward $ ":

EBNF form:	CFG:
start -> stmt $ stmt : 'step' (ID | NUM)? direction direction : 'forward' | 'back' | ...	start -> stmt $ stmt -> 'step' stmtprod | ID direction | NUM direction | direction direction -> 'right' | 'left'

Figure 7.2: A small piece of the grammar.

	'step'	ID	NUM	'forward'	'back'	$	stmt	stmtprod	direction
0	Shift 2								
1						Reduce start ->stmt			
2		Shift 4	Shift5	Shift 7	Shift 8			Goto 3	Goto 6
3						reduce stmt ->'step' stmtprod			
4				Shift 7	Shift 8				Goto 6
5				Shift 7	Shift 8				Goto 6
6						reduce stmtprod ->direction reduce stmtprod ->ID direction reduce stmtprod ->NUM direction			
7						reduce direction ->'forward'			
8						reduce direction ->'back'			

Table 7.1: Shows the parse table generated from the piece of grammar shown in figure 7.2.

state transition	**input**	**stack**
state 0 ->2: push step onto stack	42 forward $	step
state 2 ->5: push NUM onto stack	forward $	step, NUM
state 5 ->7: push forward onto stack	$	step, NUM, forward
state 7 ->5: reduce direction 'forward'	$	step, NUM, direction
state 5 ->6: Goto 6	$	step, NUM, direction
state 6 ->3: reduce stmt ->NUM direction	$	step, stmtprod
state 3 ->1: reduce stmt ->'step' stmtprod	$	stmt
state 1 ->$: reduce start ->stmt	$	start

Table 7.2: The actions taken by the parse engine on a code string in our language.

7.3 Choice in Parser

We have examined the two common solutions to constructing the parsers for a compiler. The recursive descent parser with its simplicity and "natural" implementation versus the more complex Bottom-Up parsers that trade speed and ease of construction and modification for being able to encompass a broader collection of languages. Speed of interpretation of the language is a priority for this project, as well as an ease of modification of the parser. At the same time the language is limited in its complexity as per the goal of the language meaning constructing a LL parser for the language is possible. Coupled with the speed and ease it will use the LL method in the construction of the parser engine. To keep the complexity low and the speed of compilation high a parser with a look-a-head of one is needed. When constructing a parser that needs more than one look-a-head one may end up with a parser that needs to go back when the chosen production did not fit the input. This makes the parsing much more complex and slows down the process. In the next section, a tool for checking the look-a-head in all productions will be used.

7.4 The Implemented Grammar

Based on the topics discussed in the previous sections, a grammar for the language of this project was constructed.

In order to check if this grammar is LL(1), first and follow sets need to be calculated. A tool was used for this [32], which required the grammar to be rewritten from EBNF (appendix 15.4) to BNF in order for the tool to calculate the first and follow set. The entire rewritten grammar can be found in appendix 15.5 listing 15.2.

For the grammar to be LL(1) the parser has to be able to predict every production with only one look-a-head. Which means that any RHS production with more than one LHS has to have every LHS start with a unique token as the LHS has to be predicted from only looking at the first token. The first set looks at all the possible starting tokens from every RHS production. Any production that has an ϵ as an LHS production has to have its follow set calculated as well. If the production is empty, the look-a-head token for the production will be the first token for any production coming after this. In this case to predict what production is the right one the parser has to have both a unique first set of tokens and a separate unique follow set of tokens, the symbols of which are not also in the first set. If the grammar is LL(1), the following has to be true:

- *No duplicate tokens in the first set*
- *first set* $\cap$ *follow set* $= \emptyset$

In table 7.3 the first and follow set for a few non-terminal productions can be seen. All of the productions in the grammar comply with the rules stated above. For example, none of the first set for the productions in table 7.3 contains duplicates, also the intersection of the first and follow set is the empty set. The entire table with first and follow sets can be found in appendix 15.6 table 15.2.

boolExprMore	COMPARATOR, ϵ	ANDOR, RPAR, 'then', EOL
aexpMore	OPRADDSUB, ϵ	RPAR, COMPARATOR, ANDOR, 'then', EOL
mexpMore	OPRMULTDIV, ϵ	OPRADDSUB, RPAR, COMPARATOR, ANDOR, 'then', EOL

Table 7.3: A part of the first and follow set for rewritten grammar

Part III

Compiler

Chapter 8

Syntax Analysis

In this chapter the benefits and drawbacks of hand-coding the lexer and parser versus using a generator will be discussed and an appropriate solution for this project will be determined. Lastly, the Abstract Syntax Tree (AST) will be examined and discussed.

8.1 Lexer- & Parser Generator

When constructing a compiler for a language some of its parts may be generated using a tool(generator), often referred to as a compiler-compiler. Normally the lexical scanner(lexer) and/or parsing(parser) phases may be manually written, or crafted by a generator. If a generator is the selected method, which generator of the many available should be used? This section describes the decisions made, and their reasons, for the compiler of this project.

By manually writing the lexer and parser phases, a compiler designer has complete control over every detail of how the parsing works, how the results are structured, and the nature of the output of the lexer and parser. If the language is sufficiently simple, these phases can be manually written relatively quickly and efficiently. Conversely, for larger languages, or if the project has limited time resources, a generator may be a better choice.

While the language of this project, see section 7.3, is smaller and simpler than languages such as C or Java, this project is under a certain time constraint. Therefore the use of a generator seems the optimal choice. In addition to time constraints and complexity, a generator is less prone to coding errors, if the generator was made correctly, and can easily be used to reconstruct these phases, should the language be updated at a later time.

There is one significant drawback to using a generator however: Since generators are general in nature, the code they generate is not always as efficient as the language in question could permit. If compiler speed and efficiency is an important factor, this aspect of generators should be considered. Nonetheless, given the educational nature of the language designed in this project, a generator is the chosen method. [30]

8.1.1 Choosing a Generator

There are several lexer and parser generators available, though the choice between them is dependent on several factors. Different generators, generate compiler code in different languages. Therefore a decision has to be made as to what language the compiler should be written in, be it Java, C or some other language, this is of course a matter of choice. It is obvious that a compiler whose language is the same across all stages is most likely to be simpler to construct and maintain, than one consisting of several languages. The most significant restriction on the choice of the generator, however, is the type of grammars it accepts. Since the language of this project is LL(1), the choice of generator is limited to those that can decipher and accept such grammars.

During the AAU lectures on compiler construction and languages, an introduction to several parser generators took place. Among these, ANTLR, COCO/R and Javacc fulfil the requirements listed above. Javacc has not been updated since 2002 and although ANTLR and COCO/R should have most of the same features, ANTLR is accompanied by its own IDE (Integrated Developing Environment). After examining both ANTLR and COCO/R it was determined that given its ease of use and versatility, ANTLR was an appropriate choice for this particular project.

8.2 Abstract Syntax Tree

The product from the parsing process is a parse tree, which contains all the syntactic information from the input. An Abstract Syntax Tree (AST) is a simplified version of this parse tree in appearance. In order to show the importance of an AST, an example will be made of a small program, which can be seen in listing /reffig:actualParsetree.

Listing 8.1: AST code for VisitFunc

```
1 function void example(int b)
2
3 int a = (4 + 5) * b.
4
5 endfunction
```

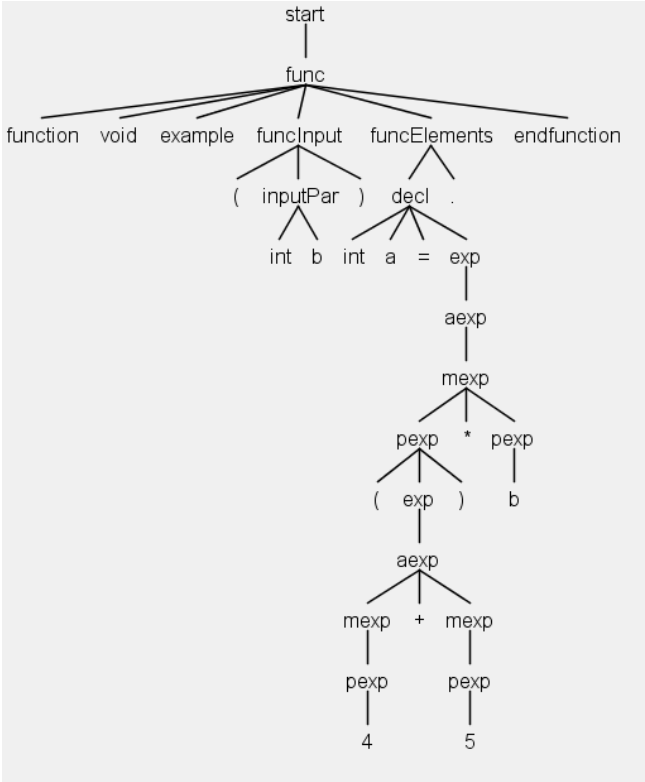

Figure 8.1: Parse tree for code example in listing 8.1

As it can be seen on figure 8.1, a parse tree made from a small line of code can quickly become large and at the same time contain more information than needed to perform the semantic analysis. Creating an AST will remove excess information by creating a new simplified structure that is easier to write visitors for, and perform type checking on. On figure 8.1, it can be seen that the tree includes all non-terminals which become lines of empty nodes with no information other than eventually leading to a usable terminal. The matching AST for this parse tree can be seen on figure 8.2.

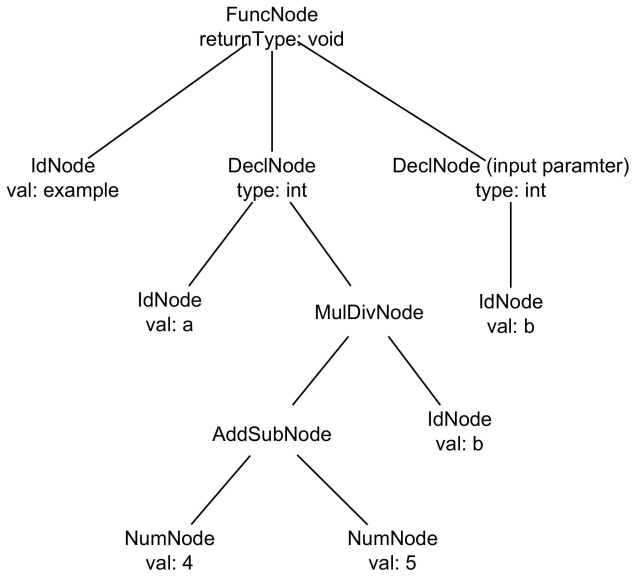

Figure 8.2: Abstract syntax tree for code example in listing 8.1.

As seen on figure 8.2, excess information, such as empty nodes and parentheses have been removed and the information has been compacted to fit into a single node. The assignment is part of the *DeclNode*, through the *MultDivNode* that is attached as its child. Had this node not existed, the declaration would simply have been *int a*. This simplification makes the semantic analysis easier to perform, as the nodes are structured in such a way that a visitor can perform an in-order tree walk, making type checking in the semantic analysis easier as well as the code generation.

8.2.1 The Implementation of the AST

The tree walk of the parse tree is done with a visitor that was auto generated along with the lexer and parser. The visitor comes with auto generated visitor methods for all the productions in the grammar.

Listing 8.2: AST code for VisitFunc

```
1  @Override public T visitFunc(@NotNull MinecraftParser.FuncContext ctx) {
2
3      T child = visitChildren(ctx);
4
5      FuncNode funcNode = new FuncNode();
6      IdNode id = new IdNode();
7      id.setVal(ctx.ID().toString());
8      funcNode.setID(id);
9      if(ctx.TYPE() != null){
10         funcNode.setReturnType(ctx.TYPE().toString());
11     } else {
12         funcNode.setReturnType("void");
13     }
14     if(ctx.funcInput().inputPar() != null){
15         funcNode.addInputParam(ctx.funcInput().inputPar().ASTnode);
16         ctx.funcInput().inputPar().inputParRec().stream().forEach((recCtx) -> {
17             funcNode.addInputParam(recCtx.ASTnode);
18         });
19     }
20
21     ctx.funcElements().stream().forEach((elementCtx) -> {
22         funcNode.addFuncElement(elementCtx.ASTnode);
23     });
24     ctx.ASTnode = funcNode;
25
26     return child;
27 }
```

Listing 8.2 shows an example of the implemented *function* declaration called VisitFunc in the class MineCraftBaseVisitor. The approach is similar when visiting other nodes in the parse tree.

The VisitFunc method is one of the first to be visited by the visitor and one of the only node types that are always visited. This is due to a rule in the grammar that requires all statements to be encapsulated in a function.

The method in listing 8.2 takes in a ctx object at line 1, containing all information for this node, of the type MinecraftParser.FuncContext.

The grammar dictates that a function must have an ID, i.e. a name, and therefore this information is saved in the funcNode at line 8. A return type, however, is not necessarily required and the default is set to void, if no other input is found. This can be seen in the if and else statements, lines 9 through 12.

Line 14 checks for input parameters, if any are found the first one is added to the funcNode and the rest, if any, are added through a foreach loop at lines 16 and 17. The difference between the first input parameter and the rest is that the first is an *inputPar* and the rest are *inputParRecs*.

The foreach loop at line 21 is a similar structure just for the *funcElements*, i.e. the body of the function.

Lastly, the built AST subtree is saved in the ctx.ASTnode at line 24, too make it accessible from the parent parse tree node for further AST construction.

Chapter 9

Semantics

This chapter covers the design and implementation of the semantics governing the language ML8. Using the AST(Abstract Syntax Tree) constructed in the parser, a visitor visits each node and performs semantic analysis upon it and its relatives. The first part of this chapter describes visitors and the use of a symbol table for type checking and scoping, while the remaining section formally covers the semantics of ML8, as Operational Semantics.

9.1 Visitors

By constructing a visitor it is possible to move the different code phases away from the nodes themselves and gather it in visitors representing the different phases of a compiler:

- printVisitor, for printing the AST
- semanticsVisitor, for doing all the semantics and decorating the AST.
- codeVisitor, for generating and emitting code in the target language based on the node visited.

This makes the compiler split the compilation into phases where the semanticsVisitor is called and does all the typechecking and decorating needed before the codeVisitor is called and uses the AST with the newly added information to emit finished target language code.

The visitor works through double dispatch by sending itself to the node and the node feeds the visitor with its own instance through the *this* keyword. Because the node itself always knows its type, the right visit(node) function in the visitor will be executed. A visit(node) function is therefore made in the visitor, for each type of node. The function in each node that accepts a visitor and calls the visit(node) is the only code left in each node when using the visitor pattern with double dispatch as shown in code example 9.1.

Listing 9.1: Accept function

```
1 @Override
2 public void Accept(ASTvisitor v) {
3
4     v.visit(this);
5
6 }
```

A print visitor is made that executes the nodes print() function when it visits the node. This is done to get a print of the AST for debugging while constructing the compiler. After the AST is constructed the compiler executes the semanticsVisitor and the semanticsVisitor instantiates a symbol table to help with the typechecking.

9.2 Symbol Table

After completing the AST the symbol table is constructed as a tool to check if any part of the code is used illegally; e.g. a variable is not declared in the same scope as it is used, or if the same variable name is declared several times within the same scope.

The symbol table is used to process symbol declarations in the AST and to connect each symbol reference with its declaration and the corresponding type.

There are different methods of implementing symbol tables. Three examples are: An ordered list, a binary search tree and a hashtable (or hashmap).

An ordered list of IDs, possibly ordered alphabetically, is a possible way of storing the symbol table. Such a list is simple to implement and uses little space, but it is inefficient. Searching the list could take linear time, depending on how it is represented. Another issue is that symbols in the list would possibly need to be moved when any new symbol is entered.

A binary search tree is another structure that could be used which is space-efficient and will, on average, be faster to search than an ordered list. The issues of an ordered list, when entering a new symbol, is avoided because of the way a binary tree is structured. However, retrieving a symbol could still potentially be in linear time, especially if the tree becomes unbalanced. This could be offset by using balanced trees, such as red-black trees, but it would take extra time to implement.[33]

Hashtables are fast and efficient for structuring symbols based on their value. Hashtables may use slightly more space than binary trees or lists, but if the IDs are inserted uniformly in the table, the search time is constant. Hash tables are also widely used in compilers and an efficient implementation is built into the java core library.[33]

The fact being that hashtables are faster and already implemented in java, as well as the inefficiency of the other data structures leads to the decision that a hashtable will be the implementation of choice for the symbol table of this project.

9.2.1 Structure of the Table

The overall design of the symbol table implementation as well as typechecking and scoping is based on the techniques described in chapter 8 of Crafting a Compiler by Charles Fischer, see [30].

The symbol table is a collection of symbol table entries, symTabEntry, each containing the name of a different ID in the AST. As well as the identifying name, the symTabEntry also contains information concerning the type of the ID, the scope the id exists in *level* and *depth*) and a reference to another ID with the same name, in an outer scope, if such an ID exists. The latter information is useful when a scope is abandoned, making the referenced ID the active declaration in the outer scope scope.

One way of doing scoping is to have a symbol table for every scope, however, considering that the built-in hashmap class of Java is equipped with adequate collision strategies, for input having the same hashvalue; the fact that the scope of an ID is saved in its symTabEntry as well as the non-existence of nested scopes in the language; the complexity of multiple symbol tables outweighs their usefulness in this project. As such, the symbol table primarily exists as a hashtable.

Listing 9.2: The function "EnterSymbol" for the SymbolTable

```
1  public void EnterSymbol(String name, String type) {
2      SymTabEntry oldSym = symTab.get(name);
3      if(oldSym != null && oldSym.getDepth() == this.depth) {
4          System.out.println("ERROR: Duplicate symbol: " + name);
5      }
6      SymTabEntry newSym = new SymTabEntry(name, type);
7      newSym.setLevel(scopeDisplay.get(depth));
8      newSym.setDepth(depth);
9      scopeDisplay.get(depth).add(newSym);
10     if(oldSym == null) {
11         Add(newSym);
12     } else {
13         Delete(oldSym);
14         Add(newSym);
15     }
16     newSym.setVar(oldSym);
```

```
17  }
```

At some point during an ID's declaration, the SemanticsVisitor will call the function in listing 9.2, which enters a given ID's name into the symbol table. Lines 2-4 determine two things, first whether or not the given name already exists in the symbol table, if that is the case, a check is also made to determine if the name is declared in the current scope; given by the *depth* field. Should this be the case, the same ID name has been declared twice in the same scope which is semantically illegal and an error is given. Otherwise lines 6-9 enters the new name into the hashtable as well as a scopeDisplay (see below). Lines 10 and forward handle the case of that an identical ID name was found in lines 2-4, but in the same scope. If so, the now "old" symbol is removed from the symbol table. In either case the new symbol is entered and is given a reference to the old symbol; if one should exist the value is automatically null.

The *scopeDisplay* is a list of lists. Specifically, the ith entry of the scopeDisplay is the list of symTabEntries with their ID names declared in scope i; i.e. these are the IDs currently active in scope i. These symTabEntries are linked by their *level* fields. The scopeDisplay is mainly used when closing the scope, where all IDs at the given scope are removed from the symbol table.

9.2.2 Type Checking & Scoping

The core purposes of maintaining a symbol table is for type checking and scoping. Type checking is of concern when declaring variables and functions (IDs), assigning to variables and calling functions using parameters.

Type Checking:
A simple section of code can illustrate the principles of type checking:

Listing 9.3: Example of type checking

```
1     @Override
2     public void visit(AddSubNode n) {
3
4         ArrayList list = new ArrayList();
5         list.add(n.GetRightChild());
6         list.add(n.GetLeftChild());
7         visitChildren(list, n);
8
9         if(n.GetLeftChild().getType().equals(n.GetRightChild().getType())) {
10            n.setType(n.GetLeftChild().getType());
11        } else {
12            n.setType("error");
13            System.err.println("Type incompatibility between: " +
14            n.GetRightChild().print() + " and " + n.GetLeftChild().print());
15        }
16    }
```

The above example is the code run when the SemanticsVisitor encounters an AddSubNode, which is either an addition operator or a subtraction operator. Lines 4-7 visit the left and right children of the operator; these can be entire expression trees themselves. Addition and subtraction are only meaningful if their operands are of the same type, in the case of this language that type is integer, or *int*. When the children were visited, the outermost terminal nodes had a given type, if not their type was determined by their specific visitor. The types of the left and right children are compared in line 9, and if they are identical the type of the operator node, in this case an AddSubNode is set to the type of the left child, which is also the type of the right child. This is useful, in the case that AddSubNode in itself is a child of some other operator where type checking continues.

If the types are not identical the source code is erroneous and lines 12-14 flag the node as being in error and prints an error message.

Type checking for declarations, assignments, function input and return values operate similarly. For example, an assignment has an expected type. This is indicated by the type of the variable on the left-hand-side (LHS) of the equal sign. The source of the new value to be assigned to the variable is on the right-hand-side(RHS). If the type of the value on the RHS is the same as the type of the LHS, the assignment is legal. Otherwise, the assignment is flagged as an error and an appropriate message is printed.

Lastly, ML8 is statically typed, meaning that the types of all IDs are known at compile time, as opposed to dynamic typing. The reason for this is that it allows the compiler to catch a multitude of errors before any target code is generated, which could result in a faulty program. Since the language is a transitional, educational one, this is the optimal approach.

Since the complex languages, to which ML8 should function as a transition, primarily use static scope rules, ML8 should do so as well. Static scope rules allow the programmer to understand the logical structure of the code in isolation, without having to worry about some variable getting a different value, as dynamic scoping can allow a variable with the same name as a variable that is not in its scope, to change its value.

Scoping

In the language of this project, functions, do-loops and if-else structures denote separate scopes. For example, this means that when a function is entered the scope depth is incremented by 1 and the internal elements of the function comprise a new scope. When the function is left, the scope depth is decremented by 1.

Listing 9.4: Example of scoping, Visit(FuncNode)

```
1  @Override
2     public void visit(FuncNode n) {
3          String rtnodetype = null;
4
5          if(symTab.RetrieveSymbol(n.getId().getVal()) == null){
6              symTab.EnterSymbol(n.getId().getVal(), n.getType());
7          } else {
8              n.setType("error");
9              System.err.println("The id \"" + n.getId().getVal() +
10             "\" has already been declared.");
11         }
12         symTab.OpenScope();
13         ArrayList list = new ArrayList();
14         boolean voidFlag = n.getType().equals("void")
15         || n.getType()== null;
16         if(n.getInputParams() != null){
17             list.addAll(n.getInputParams());
18             symTab.RetrieveSymbol(n.getId().getVal()).setFuncinputref(n.
19             getInputParams());
20         }
21         if(n.getFuncElements() != null){
22             list.addAll(n.getFuncElements());
23         }
24         visitChildren(list, n);
25
26         //A lot of typechecking removed from example...
27
28         symTab.CloseScope();
29     }
```

In listing 9.4 the visitor for functions can be seen, i.e. funcnode. The function itself, is declared as part of a higher scope, in which the name of the function exists as an id. Lines 5-10 determine whether a function with the same name has already been declared in the current scope, which would be an error. If this is not true, the function is entered into the symbol table.

Line 12 and forward, from OpenScope() to CloseScope(), comprises a new scope: The parameters and internal statements of the function. Lines 14-15 determine if the function is void. Lines 16-24 Establishes whether or not any input parameters and function body(FuncElements) exist, if so these are visited at line 24. From line 25 and onwards, within the new scope, a varying amount of typechecking may occur. Specifically for the function itself, typechecking is made between any return nodes in its body against the returntype of the function. This typechecking involves a lot of lines, not in the above listing.

Lines 25 and 27 call functions that detects several semantic issues relating to return statements in the functions body and nested if-else chains respectively. The following lines all print various error messages depending on special conditions concerning type mismatch as well as missing or too many return statements and unreachable code segments.

The following section will detail the formal specifications of the semantics of ML8 using operation semantics.

9.3 Operational Semantics

In this section the semantic meaning behind the language is explained. This will be done through natural semantics, constructed with the *environment-store* model. It should be noted that the semantics described here, fit the grammar as it is and not the functionality that was originally intended for the language. These differences will be further explained in a later chapter.

Two tuples are defined, these are *block* and *var*. *block* contains the x, y, z coordinates along with the material of a given block. If the material is null, the block has no material i.e. it does not exist. A member of a block will be denoted as $block_{member}$ e.g. $block_{material}$ to get the material of the block.

$block = (x, y, z, material)$

The *var* tuple, contains the type and value of a given variable. The valid values for any given type can be found through the T function, which takes a type and then gives a set that belongs to that type, e.g. $T(int)$ gives the set of all integers $\mathbb{Z}$.

$var = (Type, Value)$

$type = int, bool, void, material, go$

$T(int) = \mathbb{Z}$

$T(bool) = \{true, false\}$

$T(block) = \{material\}$

$T(go) = \{forward, left, right, back, up, down\}$

$T(void) = \emptyset$

9.3.1 Environment-Store model

The natural semantics will be described via the *Environment-Store* model. An *environment* is a partial function that maps a variable of some kind, to a location in the computers memory, called *Loc*. This can also be thought of as a symbol table. The *Store* is a partial function that maps*Loc* to specific values. Several environments are used to describe the meaning of the language, thereby requiring these to be described in greater detail.

Variable environment

The variable environment, denoted *EnvV*, holds all the variables that are not directly associated with the player or the world, these have separate environments. The *next* keyword points to the next free memory location (*Loc*), and is often used in context with the *new* keyword, which when used, represents the next location, whether it's free or not, i.e $next\, l = l + 2$, due to a variable taking up two locations (type, variable).

$EnvV = Vars \cup \{next\} \rightharpoonup Loc$

where env_V is an arbitrary member of $EnvV$.

Drone environment

The drone environment, denoted $EnvD$, is responsible for the variables that are directly associated with the player such as the players current location and keywords that point to various blocks around the player. In the code execution the player is replaced by a drone, hence the name env_D. The environment holds the drone's current x, y and z coordinates for its location, as well as a pair of coordinates for where the drone started when the procedure was entered, this allows the use of the keyword *origin* in the code. The keywords in this environment act as pointers to a specific block in the world environment, e.g $env_D(forward)$ points to the block in front of the drone's location. Updating this environment makes use of keywords that point to the blocks around the drone, these are used to update where the keywords in the env_D are pointing to in the env_W e.g. $env_D[forward \mapsto newForward$ makes the forward keyword point to the new block in front of the players, which is used in cases where the drone moves. The keywords used in those situations and in this environment are: $\{newForward, newBack, newRight, newLeft, newUp, newDown\}$.

$EnvD = x, y, z, xs, ys, zs \cup \{forward, back, right, left, up, down\} \rightharpoonup Loc$

where env_D is an arbitrary member of $EnvD$

Procedure environment

The procedure environment, denoted *EnvP*, contains information about bindings of procedure names. The definition is made for the procedures to have static scope rules. For variables this means that they only exist in the scope where they were declared. *Pnames* are the procedure names in the env_P. *Stmt* are the statements within the procedure. $\vec{x}$ is the formal input parameters of the procedure which are given by the abstract grammar. These can vary in size from none to an unlimited amount of input parameters. The number of parameters is the length of the vector. EnvV is the variables to manipulate in env_V. Env_P are the procedures called within the given EnvP. EnvD are the variables to manipulate in env_D and EnvW are the procedures to manipulate in env_W. ReturnType is the type of the procedure's return value with the only exception being void that cannot be returned.

$$EnvP = Pnames \rightharpoonup Stmt \times \vec{x} \times EnvV \times EnvP \times EnvD \times EnvW \times ReturnType$$

where env_P is an arbitrary member of $EnvP$

World environment

The world environment, denoted *EnvW*, maps all the blocks in the game world to locations. it's

$$EnvW = Blocks \rightharpoonup Loc$$

where env_W is an arbitrary member of $EnvW$

Store

Sto is short for *store*, and is a partial function which maps the values of the locations, in the environments, to values.

$$Sto = Loc \rightharpoonup Values$$

where *sto* is an arbitrary element of *Sto*.

Figure 9.1 shows how the *Environment-Store* works when getting the value from a member of *block*. The example shows the world environment, which holds all the blocks in the game world, and how each block maps to four different locations in *Loc*, each denoted by l followed by the name of the block member. The *store* then maps each member to a value. The same approach is used when getting a *var*, only it contains just two values instead of four.

Listing 9.5 shows the syntactic categories alongside the name of the production on the left side of the $\in$ and the set it belongs to on the right side. A description on the far right side explains the meaning of the set. For example: The name of the production is called n which belongs to the set Num, which is the set of all numerals. The sets written in capital letters signify that the production can be found as a transition in listing 15.4 through 15.8 and further described as Abstract Grammar in listings 9.6 through 9.10. For example: aexp can be

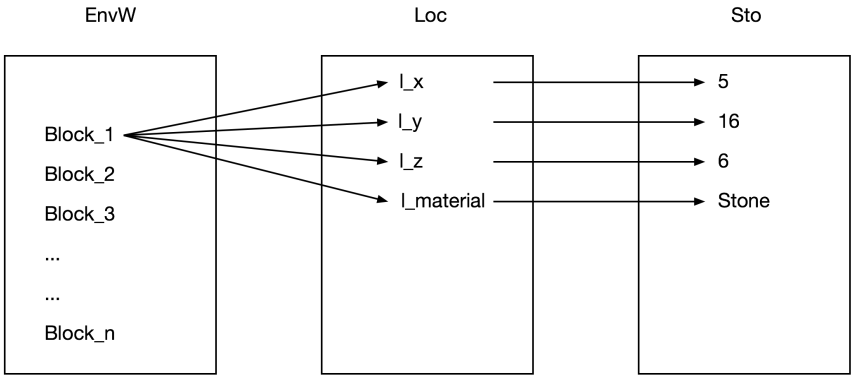

Figure 9.1: Caption

found as an abstract Grammar in listing 9.8 and as a transition rule in listing 15.6 in appendix 15.7.

Listing 9.5: Syntactic Categories

$n \in Num$ - Numerals

$x \in Var$ - Variables (ID's)

$m \in Material$ - Name of any materiial in Minecraft

$d \in Direction$ - Any direction e.g. forward, left, back

$t \in Type$ - int, block, bool, go, void

$aexp \in ARITHMETIC$ - Arithmetic Expressions

$bexp \in BOOL$ - Boolean Expressions

$stmt \in STATEMENT$ - Statements

$decl \in DECLARATION$ - Declarations

$func \in FUNCTION$ - Functions

$funcElements \in FUNCTION$ - Function Elements

Listing 9.6: FUNCTION

```
func    : function t x (inputPar) funcElements endFunction
        | function x (inputPar) funcElements endFunction
        | function t x funcElements endFunction
        | function x funcElements endFunction
```

```
inputPar: t x | inputPar ; inputPar

funcElements: decl | stmt | decl funcElements
            |stmt funcElements | EPSILON
```

Listing 9.7: DECLARATION

```
decl    : t x | t x = aexp | t x = bexp | t x = d | t x = m
        | x = aexp | x = bexp | x = d | x = m |
```

Listing 9.8: ARITHMETIC

```
aexp    :  x | n | (aexp) | aexp1 + aexp2 | aexp1 − aexp2
        | aexp1 ∗ aexp2 | aexp1/aexp2
```

Listing 9.9: BOOL

```
bexp    : x | n | m | b | (bexp)| bexp1&bexp2
        | bexp1|bexp2 | bexp1 < bexp2
        | bexp1 > bexp2 | bexp1 <= bexp2
        | bexp1 >= bexp2 | bexp1 == bexp2
        | bexp1! = bexp2
```

Listing 9.10: STATEMENT

```
stmt    : call funcCall
        | place x-Or-n m d | place x-Or-n m | place m
        | place m d
        | step x-Or-n d |step d
        | do x-Or-n times using x funcElements enddo
        | do x-Or-n times funcElements enddo
        | do x-Or-n times enddo
        | if bexp then funcElements ifElse else
          funcElements endif
        | if bexp then funcElements ifElse endif
        | if bexp then funcElements else
          funcElements endif
        | if bexp then funcElements endif
        | return x | return n | return m
        | return d |return b
        | origin

funcCall: x |x d| x (callPar) d | x (callPar)

callPar : x | n | d |b | m | callPar ; callPar

x-Or-n    : x | n

ifElse  : elseif bexp then funcElements ifElse | EPSILON
```

9.3.2 Example of semantics applied to code

In order to demonstrate how the semantics are applied to the developed language, a small program has been constructed that covers a variety of the semantic transitions. This program can be seen in listing 9.11. Only the relevant transition rules will be explained and the full set of transition rules can be found in appendix 15.7.

Listing 9.11: Example code

```
1  function example(int a)
2      if a < 7 then
3          a = a + 3.
4      else
5          a = a - 3.
6      endif
7      place a stone.
8  endfunction
9
10 function main()
11     int a = 5.
12     call example(a).
13 endfunction
```

$$\frac{\begin{array}{l}[FUNC - DECL_{BSS}] \\ env_V, env_D, env_W \vdash \\ \langle Dp, env_P[P \mapsto (stmt, \vec{x}, env_V, env_D, env_W, env_P, returnType)]\rangle \rightarrow env'_P\end{array}}{env_V, env_D, env_W \vdash \langle functionpreturnType(\vec{x}, env_P)\rangle \rightarrow env'_P} \tag{9.1}$$

$$\frac{\begin{array}{l}[DECL - ASSIGN_{BSS}] \\ env_V, sto \vdash type \rightarrow v_1 \\ env_V, sto \vdash Value \rightarrow v_2 \\ \langle Dv, env''_v, sto[l \mapsto Var(v_1, v_2)]\rangle \rightarrow_{Dv} (env'_V, sto')\end{array}}{\langle type\, x = value, Dv, env_V, sto\rangle \rightarrow_{Dv} (env'_V, sto')} \tag{9.2}$$

The program starts by having the *main* and *example* functions being declared according to the $FUNC-DECL_{BSS}$ rules, shown in equation 9.1. Followed by a call to main, which is where the program starts. All of the environments are affected due to the use of static scope rules in the language. Next is a declaration which assigns an integer value a to 5. It can be seen in the transition rules for the $DECL-ASSIGN_{BSS}$ equation 9.2 where both the value for the type and the number is stored in v_1 and v_2 respectfully. This is due to the language having several variable types. The *variable environment* and *store* are both updated to contain the new variable and its values.

$$\begin{array}{l}[FUNC-CALL_{BSS}] \\ env_V, sto \vdash x_i \rightarrow v_i \\ \dfrac{env_W, env_D, env_V, env_P \vdash \langle K, sto\rangle \rightarrow sto'}{env_W, envD, env_V, env_P \vdash \langle p(\vec{x}), sto\rangle \rightarrow sto'}\end{array} \tag{9.3}$$

The function *example* is now called. According the the rules of $FUNC-CALL_{BSS}$, which can be seen in equation 9.3, the value for every parameter in the $[\vec{x}]$ parameters are being stored, followed by a call to the function, which may change the *store*. It is not possible to give a function as a parameter when calling a function, which is why there are no side effects when determining the value of the input parameters.

$$\begin{array}{l}[SMALTHAN-1_{BSS}] \\ env_V, sto \vdash aexp_1 \rightarrow_a v_1 \\ \dfrac{env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 < aexp_2 \rightarrow_b tt} \quad if\ v_1 < v_2\end{array} \tag{9.4}$$

$$\begin{array}{l}[IF-1_{BSS}] \\ \dfrac{env_V, env_P \vdash \langle K_1, sto\rangle \rightarrow sto'}{env_V, env_P \vdash \langle if\ bexp\ then\ K_1, sto\rangle \rightarrow sto'} \quad if\ env_V, sto \vdash bexp \rightarrow_b tt\end{array} \tag{9.5}$$

Inside the *example* function, an *if* statement decides the value of variable *a*, which is transferred to the *example* function through pass-by-value, due to pointers not existing in the language. The boolean expression inside the *if* is determined through $SMALTHAN-1_{BSS}$ which evaluates the expression to true, according to the rules shown in equation 9.4. The $IF-1_{BSS}$ rules, shown in equation 9.5 determine that the *store* is changed in accordance with the execution of the body *K*, which may change the *store*. The rule for *else* is never in effect as the code is not reached.

$$[PLUS_{BSS}]$$
$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 + aexp_2 \rightarrow_a v} \quad where\, v = v_1 + v_2 \tag{9.6}$$

$$[ASSIGN_{BSS}]$$
$$\frac{env_V, sto \vdash x_{type} \rightarrow v_1 \quad env_V, sto \vdash value \rightarrow v_2 \quad \langle Dv, env_V, sto[l \mapsto Var(v, null)]\rangle \rightarrow_{Dv} (env_V, sto')}{\langle x = value, Dv, env_V, sto\rangle \rightarrow_{Dv} (env_V, sto')} \tag{9.7}$$
$$if\, v_2 \in T(v_1)$$
$$where\, l = env_V\, next$$
$$and\, sto' = sto[l \mapsto value]$$

The *K* body of *if* assigns *a* to itself plus 3. The arithmetic expression is being evaluated through the $PLUS_{BSS}$ according to the transition rules in equation 9.6. If the determined value exists in $T(a_{type})$, as described by $ASSIGN_{BSS}$, shown in equation 9.7, then the arithmetic expression is assigned to *a*.

$$\frac{\begin{array}{c}[PLACE - X = 1 - M_{BSS}] \\ env_V, sto \vdash x_{value} \rightarrow_v v_1 \\ env_V, sto \vdash m \rightarrow_v v_2 \\ env_V \vdash d \rightarrow_v v_3 \end{array}}{env_W \vdash \langle place\ x\ m\ d, sto \rangle \rightarrow sto'} \quad (9.8)$$

$$\begin{array}{l} if\ v_1 = 1 \\ if\ x = null\ then\ v_{1value} = 1 \\ if\ d = null\ then\ v_3 = forward \\ Where\ env_W(env_D(v_3)) = l \\ \quad sto' = sto[l_{material} \mapsto v_2] \end{array}$$

$$\frac{\begin{array}{c}[PLACE - X > 1 - M - D_{BSS}] \\ env_V, sto \vdash x_{value} \rightarrow_v n \\ env_V, sto \vdash n \rightarrow_v v_1 \\ env_V, sto \vdash m \rightarrow_v v_2 \\ env_V, env_D \vdash d \rightarrow_v v_3 \\ env_W, env_D \vdash \langle place\ x\ m\ d, sto'' \rangle \rightarrow sto' \end{array}}{env_W, sto \vdash \langle place\ x\ m\ d, sto \rangle \rightarrow sto'} \quad (9.9)$$

$$\begin{array}{l} if\ x_{type} = int\ and\ n > 1 \\ env_W(env_D(v_3)) = l \\ sto'' = sto[env_W(env_D(v_3))_{material} \mapsto v2][env_D(x) \mapsto l_x] \\ [env_D(y) \mapsto l_y][env_D(z) \mapsto l_z][n \mapsto n - 1] \\ if\ d = null\ then\ v_3 = forward \\ env''_D = env_D[forward \mapsto newForward][back \mapsto newBack] \\ [right \mapsto newRight][left \mapsto newLeft] \\ [up \mapsto newUp][down \mapsto newDown] \end{array}$$

Finally the *place* statement is called with the a variable and the *stone* material. The $PLACE_{BSS}$ is split into a recursive case, shown in equation 9.9, that is being chosen if the variable value is above 1, and a base case, shown in equation 9.8, if the value is exactly 1. The recursive case places a block in front of the drone, because no direction was given. It then moves the drone forward, updates what the keywords in the env_D maps to, and decrements a separately stored value of a_{value}, in order to not alter the variable itself. This is done until said value is 1, which triggers the base case. This case only places a block with the given material in front of the player, thereby updating the world environment env_W.

Chapter 10

Code Generation

On the basis of the semantics of ML8, the code generation phase is constructed. Code generation is the translation of the language to the target language that can be executed in another context, in this case Minecraft.

There were a few options for choosing a target language. The first option was to write a custom mod for Minecraft, which would make the target language a series of function calls to some custom made code, that manipulates the world in the desired fashion. The second option was using an existing mod that allowed programming in Minecraft, and then have the target language be the code that is accepted by said mod. In order to preserve time it was decided to generate code that could run through an existing mod, more specifically the mod *ComputerCraft*. The concept in this mod is that the player can program a special block known as a *turtle*, to move about, destroy and place blocks. The blocks that are destroyed by the *turtle* are placed in its inventory, which consists of 16 slots, each slot can hold up to 64 blocks, depending on the maximum stock size for that particular block type. The programmable block takes in Lua code which will be the target language. Lua is a dynamically typed scripting language with some similarities to C, due to the fact that functions have to be written above the point of their usage, i.e. declared before use.

10.1 Code generation implementation

The language, ML8, gets translated into Lua which, through the code generation, provides a file to add to the Minecraft folder that connects to the *turtles*. Each part of the code that the user types in the language ML8, gets translated to Lua, first after extensive syntax and semantics checking and thereafter through the methods of the class *CodeGenVisitor* located in the folder *ASTVisitor*.

PlaceNode

If, for example, the user types in

place 4 "stone" left.

in a function it goes through the method of *visit(PlaceNode n)* which calls both *visit(NumNode n)*, *visit(DirectionNode n)*, and *visit(MaterialNode n)* that prints the number, direction and material, respectfully, for the given part of code where it is required.

Listing 10.1: A piece of the visit(PlaceNode n) method in codeGeneration

```
@Override
public void visit(PlaceNode n) {
    errorMessage(n, "PlaceNode");
    ArrayList list = new ArrayList();
    DirectionNode direct = (DirectionNode)n.getDirection();
    MaterialNode mat = (MaterialNode)n.getMaterialNode();

    writer.print("if string.find(\" ");

    if(n.getNum() != null){
        list.add(n.getNum());
    }
    else if(n.getId() != null){
        list.add(n.getId());
    }
    visitChildren(list);

    //Looks for all "-" symbols in the value
    writer.print("\", \"%-\") == nil then \n");
    list.clear();

    //If the direction is positive
    if(mat.getVal().contains("door")){
        ... // code if the material is door
    }else if ... //Method calls for up and down
    }else{
        forwardLeftRightBackDirection(n, direct, list);
        list.clear();
    }

    //if the direction is negative
    writer.println("else ");
    if(direct.getVal().contains("up") ||
       direct.getVal().contains("down")){
        ... // method calls for up and down
    } else{
        .. // method calls for forward, back, left and right
    }

    writer.println("end -- ends: if string.find");
}
```

Listing 10.1 shows some of the code from the method *visit(PlaceNode n)*. The dots show that there is more to this method than represented in the listing, which is however not relevant to this example. The first step for every visit-method, in the CodeGenVisitor class, is calling the *errorMessage* check that checks if the type is "error", if so it prevents further execution of the code generation.

To determine the number of times the *turtle* has to place left, it is needed to receive a value, which in this case is the number 4. This is found on line 10 through 16, where the Lua-code on line 8 and 19 encapsulates it in a string.find(), which searches the value for the "-" sign i.e. if it is a negative number. The negative number is a command that allows the *turtle* to remove blocks instead of placing them, but otherwise functions the same as with a positive number.

If it is negative, it will go to lines 31 through 38 in listing 10.1. But since the number is 4, it will go to the method *forwardLeftRightBackDirection* on line 27 which can be seen on listing 10.2.

Listing 10.2: A piece of the FrontLeftRightBackDirection() method in codeGeneration

```
1  protected void forwardLeftRightBackDirection(PlaceNode n,
2                      DirectionNode direct, ArrayList list){
3      if(n.getNum() != null){
4      list.add(n.getNum());
5      }
6      else if(n.getId() != null){
7          list.add(n.getId());
8      }
9      if(direct.getVal().contains("right")){
10         ... // Code for right
11     }else if(direct.getVal().contains("left")){
12         writer.println(" turtle.turnLeft()");
13         writer.print("originX = originX - tonumber(");
14         visitChildren(list);
15         writer.println(") ");
16     }else if
17         ... // code for forward and back
18     }
19     list.clear();
20     forPart(n, list);
21
22     writer.println(" \nturtle.placeDown() \nturtle.forward() \n"
23                    + "end -- ends for");
24
25     //Returns the turtle to the original direction.
26     if(direct.getVal().contains("right"))
27         ...
28     else if(direct.getVal().contains("left"))
29         writer.println(" turtle.turnRight()");
30     else if ...
31
32 }
```

Since it was decided that the *turtle* should always face the same direction, and this is not how the *turtle* normally operates, the implementation of the direction has been implemented as shown in listing 10.2.

First the direction is found, in this case left. Since the *turtle* cannot "walk" sideways or backwards, the *turtle* has to face left in order to go left. This is done with the command *turtle.turnLeft()* on line 12. Hereafter it iterates through the loop the number of times the value dictates e.g. 4, one iteration makes the *turtle* step one forward in the left direction. This is represented in the *forPart()* method. The for-loop ends on line 22 in listing 10.2 where it places one down before taking a step forward. If the number was negative it would have moved before removing one from underneath it. It then turns right, to return to the right direction. The same applies when using the "step" command, except that it does not place any blocks.

Origin Origin is a premade function in the Lua code and is the only generated function not defined by the programmer. This serves to prevent using any keywords that could conflict with any keyword in user generated code. The origin statement will return the *turtle* to the position it started at, when a given function is called, and can be used at any time in a function. If the example from listing 10.2 is used, the originX (the x-axis) is decreased by 4, because the left direction is towards the negative side of the x-axis, shown on line 13 to keep track of how far away the *turtle* is from the "origin". This point was given the coordinates (0, 0, 0), when it started.The values changed according to the direction the *turtle* has moved. Note that the coordinates are a local coordinate system for the *turtle* and essentially have nothing to do with the actual world in Minecraft.

The method *visit(OriginNode n)* calls a function *origin()* that is a function that only contains Lua code. The code in itself is large and the functionality of the function will therefore be explained through figure 10.1

On figure 10.1, on the first picture, a path that the *turtle* has travelled is shown. The arrows indicate in what order it got there and in the middle is the origin where the *turtle* was executed at coordinate (0,0,0). For each function called the *turtle* is assigned a new origin. Every time the *turtle* takes a step or places a block the variables originX, originY, originZ count up or down depending on the direction. The function for this example could be as shown in listing 10.3.

Listing 10.3: Code in the language ML8

```
1 function test()
2     step 2 back.
3     step 3 left.
4     step 5 forward.
5     step 3 up.
6     origin.
7 endfunction
```

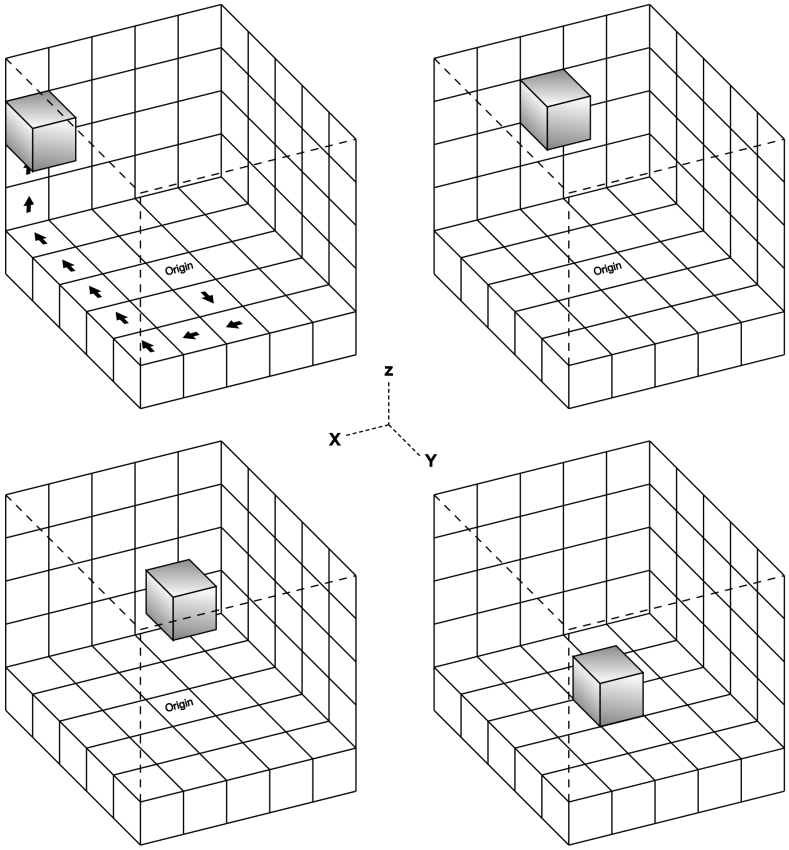

Figure 10.1: An illustration of the origin functionality

Going through each command from line 2 to 5, adds or substracts from the three coordinates. In the first case $originY = -2$, the next $originX = -3$ while originY and originZ remains the same, then $originY = -2 + 5 = 3$ and $originZ = 3$. In the end the *turtle* is positioned at coordinates (-3, 3, 3).

When origin is called it starts with the x-coordinate where it runs from 1 to 3 while stepping right towards 0 as seen in the second picture on figure 10.1 while this happens it checks for every step, if something is standing in its path. If so, it removes it and when it walks past it, it turns around and places the block again. When originX = 0, it moves on to the coordinate y, with the same approach. And lastly coordinate z, where it returns to its starting point. If the origin (0,0,0) in the meantime, has been filled with another block, the *turtle* will destroy it and take its place.

Part IV

Evaluation

Chapter 11

Language Limitations

This chapter will touch on the limitations of the language that was discovered through tests, along with their impact on its functionality.

11.1 Test

During the implementation of the compiler, it was necessary to test the functionality of the different elements such as the AST, semantics and code generation. When the compiler was assumed to be complete, further testing was also performed to ensure that all elements worked together properly. Two different methods of testing were performed on each element.

The first method was a form of concrete testing, using *System.out.Print()* to ensure that different aspects, e.g. functions had the correct values or a variable got incremented correctly and worked as intended. It could also be used to pinpoint errors occurring when running the program. On a bigger scale, this method was used to check if the AST visited all the nodes and in the correct order for any code example. This was accomplished with several *System.out.Print()* that were incorporated into a method called MiniVisitor(). By typing in different kinds of input that used all the nodes in different contexts, it was possible to compare the results with the expected result done by hand. If it did not match, a search for the underlying problem was performed with more prints. Even though the prints for the nodes were only used as testing of the AST element, the prints proved useful later in the process to check how code was represented in the AST.

The second method was a form of general testing, mostly used when an element was deemed complete. This method was based on watching the output from the finished parts of the compiler as a whole and reacting to it if needed. Regarding the semantics, it was used to type in different kinds of input, correctly and and incorrectly, in order to see if the expected error messages were printed. This was done with each of the commands and with different variations. In code generation the method was used to check if the translation from ML8 to Lua was made correctly by typing in an input and checking if the translation to Lua was made correctly. This was mostly used for smaller details e.g. if a specific ID was placed or incremented correctly. If not, a combination of the two methods could be used to check where the incrementation went wrong and correct it.

When the compiler was thought to be finished. The second method was used to ensure that the compiler functioned properly and the *turtle* was given the right commands. This was done by giving it various lengths of code to translate into Lua. If the code was not written correctly or any variables weren't declared, the semantics would provide error messages that explained the issue, and prevent the code from being translated using code generation as explained earlier in this chapter. When the code was written correctly and the generated file was placed where the *turtle* could run it, the functions could be written in the Computercraft console, thereby executing them. If the translation from ML8 to Lua was incorrect, Computercraft would provide an error message to react to. When everything went as planned the *turtle* would move and reveal if the translation was done incorrectly, making the *turtle* move in a wrong pattern. An example of one of the code pieces that was tested this way, can be found in appendix 15.8.

11.2 Errors and Shortcomings

Varies errors in the ML8 language were discovered through test methods that are described in section 11.1. Errors were found in both the language's grammar and implementation of the compiler. Some of the errors in the syntax could be handled in the semantic analysis, only those that could not, are described in this section.

The biggest limitation of the language is that it was not designed for the functionality that a *turtle* can provide. The *turtle* can't go through blocks, which contrasts the original plan for the functionality of the language. To work around this problem it was decided to have the *turtle* move up and then place the blocks below it, like a 3D printer. Although this is adequate in many scenarios it can cause some confusion, as the *turtle*'s actual movements do not correspond directly to the way it is coded in ML8. The difference between when the turtle places one block up and one block down is that when going up, the *turtle* moves up one block and places it beneath it.

When placing it down, the *turtle* moves one down and then places the block above it. The command essentially yields the same result, only the *turtles* position after the execution of the statement differs. Having the *turtle* move in some direction before placing the block, has made it impossible to remove a block that is directly in front of it, as it would try to move one up, then one ahead and finally try and remove the block beneath it. All of these small quirks don't limit the possibilities of what the *turtle* can make, it merely has to be done differently and might require more work than anticipated.

Syntactically it is possible to use the return statement in a function, although this does not have any effective use, due to the syntax not allowing the use of function calls in declarations, arithmetic expressions or as input parameters in other function calls, making the return statement useless. When going back through the thought process of designing the language, no clear reason exists as to why the return statement should be in the language. Adolescents have a hard time understanding functions, which is why many of the functionalities of languages such as Java and C# were omitted, including recursion and the need to explicitly write a void return type. In retrospect, a return statement should not have been in the language.

It was not possible to implement a sufficient way to choose which material that should be used, within the time constraint of the project. The development process for this functionality proved to be longer than expected, therefore other functionalities were prioritised above it. Alternative functionalities have been implemented in order to create some control of used materials. The *turtle* uses a specific slot from its inventory when placing blocks. Whatever material is in that slot, is the material that is placed. As of now, there are comments in the code that give instructions of where to place materials. Along with this, the *turtle* will check for more material of the same kind in its other inventory slots, when there is only one block left in the current slot. If it finds more material in other inventory slots, it will be moved to the current slot. Although this is an imperfect implementation of the "choose material" functionality that was originally envisioned, it does help the user control what materials will be used.

Due to a syntactical shortcoming, it is not possible to use the direction in boolean expressions directly. e.g. $a == forward$ is not possible. Although this is possible if the direction is represented as a variable id instead. This is an error in the language, which would have to be corrected in future work.

Since the *turtle* does behave exactly as the semantics say that it should, and there is no way to command it to instantly go to a specific location, it has not been possible to implement the origin statement correctly. The functionality of it is implemented by an algorithm that tries to navigate the *turtle* one axis at a time, removing and replacing any blocks that might be in its way, one axis at a time.

The algorithm guarantees that the *turtle* returns to its point of origin, although there is no guarantee, that when the turtle replaces the block it removes in order to proceed, will be of the same type, as there is no way of telling where in the turtles inventory the block went, when it was picked up. Another shortcoming is that whenever a function is called, the point of origin is overwritten. This means that when the program returns to a previous function after executing a function call, the point of origin for the previous function has been overwritten by the function call. This could however be fixed if there was more time for the project.

It is not possible to write arithmetic expressions in place and step statements, this is an error in the language and can be worked around by assigning a variable to the arithmetic expression first, and then passing the variable to the step or place statement.

When a function is called and it takes a parameter of the type *go* which is a direction, the placeholder id can not be used in a step or place statement. The syntax dictates that it must be a direction i.e. forward, back, right, left, up or down, which is an error in the language.

Chapter 12

Discussion

Many decisions were made during the course of this project and objectives were scheduled based on these decisions. In this chapter, a reexamination or reaffirmation of some of these choices is appropriate, as the knowledge and experience gained provides new viewpoints and issues to consider.

Grammar
When the grammar for ML8 was first written, it was tested for correctness by using First & Follow sets as well as some rudimentary testing done with a built in feature of the ANTLR utility, based on the same principles. These tests showed no grammatical errors and the project carried on. Subsequently, when implementing the compiler, both when writing the code for the AST as well as when writing the code for the semantic checks, numerous errors or missing elements were discovered in the grammar, some of them detrimental to the functionality of the features of the language. Such errors were found on multiple occasions and required considerable time and effort to correct, given that entire sections of code had to be rechecked, retested and sometimes even rewritten to accommodate changes to the parser.

An important lesson from this, for any future projects involving grammar and parsing, is therefore to meticulously check and test the grammar after it is initially deemed complete, as well as theorise possible scenarios for missing features, in order to avoid most of such mistakes.

ANTLR
ANTLR served as a useful lexer and parser generating utility for this project and given the many grammar rewrites mentioned previously, reprogramming these compiler phases, continually, would have taken a lot of time without it. However, the newest version of ANTLR(4.5), which was used for this project, has one detriment in that its parser generates a full parse tree and not an abstract syntax tree. A large amount of the information retained by a complete parse tree is non-essential, or even unnecessary, for writing the later compiler phases. It was therefore decided that for this project, the parse tree needed to be converted into an abstract syntax tree, the coding of which took several work-hours to complete. For future projects, careful consideration should be taken upon deciding whether or not the features of ANTLR is worth the time it takes to convert the parse tree, when an abstract syntax tree is desired.

Testing
The first method for testing, as mentioned in section 11.1 proved very useful when trying to locate exactly where an error occurred. This was because the ASTNodes were printed in a specific order and as a result, if any one of them was faulty, there would be a noticeable error in the printed list. Examples could be: Missing nodes, nodes of an unexpected type and nodes in a wrong position. A node printer, such as the MiniVisitor(), should not be discounted for testing of future projects.

Computercraft
An alternative to *Computercraft* could be a mod called *Scriptcraft*, which allows the user to create in-game programs using javascript. The mod is harder to install than *Computercraft* but also has much more functionality, as the mod isn't limited to navigating around a block. The simpler API, visual representation and easier installation of Computercraft, is the reason it was the chosen mod. The simple API also allows the user to easier manipulate the raw Lua code that is generated, should the user want to do so.

Chapter 13

Conclusion

This chapter serves to reiterate the process of the project as well as to determine whether or not the end result of the project meets the requirements specified in the problem statement, in section 5.1, and to what extend those requirements have been met.

13.1 Flow of Information

This section will reiterate the process of the project and the flow of information from each part to the next.

Starting with the analysis, various ways of teaching programming were examined through a number of interviews. These interviews where helpful by providing avenues of focus for the later development of ML8, such as how advanced and abstract the features of the language should be. The interviews also helped solidify the idea that familiarity with platforms and creative aspects would be optimum ways of engaging adolescents in programming.

In the subsequent chapters of the analysis, existing educational languages were reviewed for form, structure and use which provided further ideas for constructing ML8, such as it having a simple syntax, similar to natural languages. The concept of a transitional language was defined, establishing a reasonable middle-ground between basic and advanced languages which is what ML8 should be. Finally, the problem statement was made, channelling the information attained by the previous chapters into a goal and a set of requirements for the transitional language. These requirements are the basis of section 13.2.

Following the problem statement the design choices and subsequent implementations of the various features of the language, including structure, keywords, data types and other language constructs were discussed and detailed based on the knowledge gained from the analysis. After the language had been designed, different mechanisms and tools for constructing a compiler for it were discussed, with primary focus on choosing a suitable parsing technique.

Lastly, after the details of the language and the methods for its parsing were determined, the exact mechanism for parsing ML8 and structuring source code in the compiler was decided, based on the features of the language, its complexity and more importantly its area of use. The exact implementation could then be described along with specific methods for type checking, scoping and other semantic checks. Thoroughly detailed operational semantics were thereafter specified for all productions of the language, before finally establishing the methods for actual code generation, transforming the abstract syntax tree of the source code into the target language.

13.2 Evaluating ML8

This section will evaluate ML8, based on the requirements of the problem statement.
The constructed language is completely text-based and can be written in a simple text-editor. Most of its keywords are plain English, conforming to the idea of ease of understanding and readability. Like most western natural languages, statements end with a full stop, '.'. There are only 30 keywords whose denotations closely match the intended functionality in the language, such as "place" signifying a function that places an object.

Parentheses can be used for signifying elements such as function input parameters and if-conditions, but they are not forced when they are unnecessary, e.g. when a function has no input parameters. The code can be structured in clearly defined blocks, replacing curly-brackets, which are widely used, with fitting bookending keywords, such as *endfunction*. There are a number of mathematical and logical operators, the mathematical ones are identical to those adolescents should recognise from mathematics classes in school. The language is as such, text-based and focused on readability.

The language should be able to serve as a transitional language:
The language has a few, simple data types such as int and bool. The highest level of abstraction in the language is in the form of functions, since there are no classes or other custom data types. There are also no reference types, such as arrays or lists and the language is both statically typed and statically scoped.

The simplicity of its structure and low level of abstraction connects the language to the likes of Scratch and dolittle, however, the ability to declare and call custom functions; the ability to construct complicated statements, including if-else chains; the forced declaration of all variables and therefore awareness of type compatibility as well as the blocked structure format, all serve to bridge the gap between the aforementioned basic languages and the more advanced languages. In short, the language should be useful as a transitional language.

The language should provide immediate visual output and function within the game:
The implementation of ML8 does eventually provide visual output through the fact that its output can be used as input to the *ComputerCraft* mod, which constructs the coded objects in the Minecraft game-world. Furthermore, what can be done directly in *ComputerCraft* using Lua, can be done with far fewer lines of ML8 code. Unfortunately, the fact that ML8 functions as an extension, of sorts, for *ComputerCraft* clearly means that the visual output is not immediate nor does the language function directly within the confines of the game world, as was originally desired. Therefore it must be concluded that in contrast to the previous two, these requirements have only been partially met.

In summation, the requirements set forth in the problem statement have largely been met, the aforementioned shortcomings notwithstanding.

Chapter 14

Future Work

This chapter will briefly discuss a few venues that would be desirable to explore, given that more time and resources were allocated to this project: The principal missing feature of the language, as mentioned in the conclusion, is that it provides no immediate visual output, nor is it "embedded" and functioning within Minecraft's game world.

The process of authoring, compiling and executing code could happen inside the target platform (Minecraft), making the integration of ML8 more seemlessly. One possible entry point, already existing in Minecraft, is an in-game item called "book and quill" (see figure 14.1), which could be modified in such a way that it would allow the user to write code therein. This would help fulfil the requirement of the language functioning within the target platform, though it does have its limitations, such as reduced space in which code can be written. It would likely require considerable time and effort, first to research how Minecraft is modified and what tools to use for this purpose, as well as actually developing and constructing a mod that would allow this functionality.

Figure 14.1: Visual example of the Book and Quill interface in Minecraft where it is possible to write text. The book and quill could be used for writing, reading and sharing code. This design would allow integration with a - for the language's target group - known environment .

If the book-and-quill idea was wholly discarded, however, another possibility would be either constructing an IDE, specifically for ML8, or modifying an existing one. Having some form of code completion, pre-compilation error messages and suggestions, underlining or other helpful features would facilitate ML8's use as a transitional language. At present time, there are many and varied useful error messages, and warnings, the user is presented with, if his or her code contains errors, however, even though these messages contain some information about where the errors are located, they are not as clear as visual underlining would be. The code must also be compiled before any error messages are displayed.

It could be interesting to invite a few adolescents, with different levels of programming experience, to try to use the language and record their suggestions and comments regarding its usefulness. This would serve a as way to further improve the language and could possibly provide new pathways towards expanding the language. Programming teachers might also be an appropriate test group especially for determining whether or not the language is useful as an educational tool. It would be beneficial if at least some of these teachers were the same as those interviewed for the analysis part of the report.

Bibliography

[1] U.S. Department of Labor Bureau of Labor Statistics. Occupational outlook handbook, 2014-15 edition, software developers. [Online]. Available on: http://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm, 2014. URL, accessed 2015-20-02 - Job outlook.

[2] Sarah Noehr. Kodning for børn er det nye hit. 2014.

[3] Mikkel Wenzel Andreasen. Nu skal programmering ind i folkeskolen. [Online]. Available on: http://www.computerworld.dk/art/231742/nu-skal-programmering-ind-i-folkeskolen, aug 2014. URL, accessed 2015-20-02.

[4] Brigid O'Rourke. Coding and creativity. [Online]. Available on: http://news.harvard.edu/gazette/story/2014/11/coding-and-creativity/, nov 2014. URL, accessed 2015-20-02.

[5] Andrew Manches. Keep it creative to get kids into coding. [Online]. Available on: http://theconversation.com/keep-it-creative-to-get-kids-into-coding-15968, jul 2013. URL, accessed 2015-20-02.

[6] Lars Jacobsen. Kode-amok: 100 danske børn programmerede i 24 timer med coding pirates. [Online]. Available on: http://www.computerworld.dk/art/232591/kode-amok-100-danske-boern-programmerede-i-24-timer-med-coding-pirates, dec 2014. URL, accessed 2015-20-02.

[7] Jeppe Bundsgaard et. al. Digitale kompetencer - it i danske skoler i et internationalt perspektiv. [Online]. Available on: http://www.uvm.dk/~/media/UVM/Filer/Udd/Folke/PDF14/Nov/141121_Digitale%20kompetencer.pdf, 2014. URL, accessed 2015-1-03.

[8] Institute for the future. Future work skills 2020. [Online]. Available on: https://docs.google.com/viewer?a=v&pid=sites&srcid=

ZGVmYXVsdGRvbWFpbnxlZ2VkYWxjb2RlcmRvam98Z3g6NDgxY2Y0Mjc0Y2U5OGI2Yw, - 2011. URL, accessed 2015-1-03.

[9] Stanford Report. Playing to learn: Panelists at stanford discussion say using games as an educational tool provides opportunities for deeper learning. [Online]. Available on: http://news.stanford.edu/news/2013/march/games-education-tool-030113.html, mar 2013. URL, accessed 2015-8-03.

[10] Constance Steinkuehler. The mismeasure of boys: Reading and online videogames. [Online]. Available on: http://www.wcer.wisc.edu/publications/workingpapers/working_paper_no_2011_03.pdf, jul 2011. URL, accessed 2015-8-03.

[11] Mitchel Resnick et. al. Scratch: Programming for all. [Online]. Available on: http://cacm.acm.org/magazines/2009/11/48421-scratch-programming-for-all/fulltext#F1, nov 2009. URL, accessed 2015-20-02.

[12] MIT Media Lab. Scratch. [Online]. Available on: https://scratch.mit.edu/about/, feb 2015. URL, accessed 2015-20-02.

[13] Rie Mitarai Shingo Fukui Yasushi Kuno Susumu Kanemune, Takako Nakatani. Dolittle — Experiences in Teaching Programming at K12 Schools. IEEE, first edition, 2004. Pages: 177-184, ISBN: 0-7695-2166-5.

[14] Robert W. Sebesta. Concepts of Programming Languages. Pearson, 10th edition, 2012. ISBN 0273769103, 9780273769101.

[15] Electronic Arts Inc. Electronic arts terms of service. [Online] http://www.ea.com/terms-of-service, sep 2012. count 13.

[16] Riot Sargonas. Riot's stance on 3rd party mods (and curse voice). [Online] http://forums.na.leagueoflegends.com/board/showthread.php?t=4491087&page=1#post46870550, may 2014.

[17] Pegi-ratings.

[18] Esrb-ratings.

[19] Inc. Blizzard Entertainment. Battle.net terms of use. [Online] http://us.blizzard.com/en-us/company/about/termsofuse.html, mar 2014.

[20] Wiki community for DOTA 2. Modding. [Online] http://dota2.gamepedia.com/Modding, may 2014.

[21] STEAM by Valve Corporation. Steam-abonnentaftale. [Online] http://store.steampowered.com/subscriber_agreement/?l=danish, mar 2015. count 2.G.

[22] SMITE. Smite end user license agreement. [Online] http://www.hirezstudios.com/docs/default-source/default-document-library/smite-end-user-license-agreement.txt?sfvrsn=2, jan 2013.

[23] Trion Worlds. Terms of use. [Online] http://www.trionworlds.com/en/legal/terms-of-use/, aug 2012.

[24] PCGamer Owen Hill. Notch reveals minecraft mod plans. changes them 30 minutes later due to 'overwhelming feedback'. [Online] http://www.trionworlds.com/en/legal/terms-of-use/, apr 2011.

[25] Mike Minotti. World of warcraft closes in on league of legends' most-played pc game spot on raptr. [Online]. Available on: http://venturebeat.com/2014/12/16/world-of-warcraft-closes-in-on-league-of-legends-most-played-pc-game-spot-on-raptr/, December 2014. accessed 2015-29-4.

[26] pgeuder. [Online]. Available on: https://twitter.com/pgeuder/status/481812186390880257, 25 Jun 2014. URL, accessed 2015-1-03.

[27] peniel789. Tutorial on a good looking house. [Online]. Available on: http://www.minecraftforum.net/forums/show-your-creation/screenshots/1589280-tutorial-on-a-good-looking-house, June 2011. accessed 2015-29-4.

[28] Mojang. Minecraft. [Online] https://minecraft.net/.

[29] Undervisningsministeriet. Fælles mål 2009 - matematik. [Online] http://uvm.dk/Service/Publikationer/Publikationer/Folkeskolen/2009/Faelles-Maal-2009-Matematik.

[30] Charles N. Fischer et. al. Crafting a Compiler. Pearson, 2009. ISBN13: 978-0-13-606705-4.

[31] Lr parser. [Online]. Available on: http://en.wikipedia.org/wiki/LR_parser, year = 2015, month = February, note = accessed 2015-20-3.

[32] HackingOff.com. Hackingoff. [Online] http://hackingoff.com/compilers/predict-first-follow-set.

[33] Thomas H.; et. al. Cormen. Introduction to Algorithms. Pearson, 3rd edition, 2009. ISBN 978-0262033848.

Chapter 15

Appendix

15.1 Pupils experience with programming

Interview conducted by group sw411f15

Number of pupils interviewed: 11.

Their motivation: Make games: 4 Get visual output: 2 Create home-pages: 2 Learn about programming and computers: 2 No motivation: 1

Why they chose this class: Knowledge about computers: 4 To make games: 3 New and exciting: 2 Their friend(s) chose it: 1 Animation: 1

What (in the correlation of programming) is hard: Textual code: 6 Connect elements: 3 No problems encountered: 2

5 pupils were shown Python code and 6 pupils were shown C# code. first row tells how many were able to tell what the code did, second row tells how many were able to tell what the code did but after a small example, last row tells how many were not able to understand the code given.

	C#			**Python**		
Construct	Easy	Medium	Hard	Easy	Medium	Hard
Assignment	3/6	2/6	1/6	1/5	4/5	0/5
Print	2/6	3/6	1/6	1/5	1/5	3/5
If	3/6	2/6	1/6	1/5	4/5	0/5
While	2/6	3/6	1/6	2/5	3/5	0/5
For	0/6	1/6	5/6	0/5	0/5	5/5
List	0/6	2/6	4/6	3/5	2/5	0/5
Switch	0/6	2/6	4/6	2/5	1/5	2/5

Table 15.1: Result with code shown to pupils. 6 pupils were shown C# while 5 were shown Python.

15.2 Interview: Hanne Kåg

Interview conducted by group sw411f15 Hanne Kåg (hereafter refered to as HK) Primary schhol teacher for 3 years Self-taught in programming Target-group: Elementary school, elective course, 7th-9th grade. About 2/3 self-motivated, and 1/3 not motivated.

Programming as elective course (7th, 8th and 9th grade).

They've tried hour of code; unity; App-inventor; gameSalad; code-adventures; Lego-mindstorm (will participate in Lego First League).

Had a visitor related to GameCamp.

Some pupils chose programming to make apps or computer-games or the like (chosen by interest). She thinks it might be for the graphical output. Maths is not main focus. Some pupils (about 1/3) chose programming because they had to choose an elective course to attend to.

Focus was on learning by doing, this made an easier approach for her as a teacher so she did not have to spend a lot of time to learn everything up front.

HK states: It i important to make it easy for the teacher to get to use the programming language.

[08:00] The purpose of programming is to show the pupils a world underneath the everyday use - to show what a computer can do and how this would be applicable professionally, not only in gaming but also ie. automatic doors.

[13:00] HK states that when the pupil have to sit in front of the computer for a long time, it helps with a task to be concrete (ie. a mobile-app with visuals or sound) - she elaborates - because they started their day in fron of the computer.

[14:00] Most pupils have a really hard time to understand programming. Because they changed so often between the different programming languages and platforms (mobile and computer) the pupils never really got to understand any of the concepts fully. Due to the many functions ie. Lego Mindstorm has, the pupils find it confusing. It can be hard for the pupil to figure out how to accomplish what they want with the constructs available to them. A reason for this, HK states, might be because they do not have enough time and because they lose concentration when nothing happens (ie. they are not able to solve the problem).

[15:20] Sometimes the ambitions of the pupils are in the way for them to reach their goal. They might have to great ambitions but lack the knowledge and/or the skills. Especially when debugging, they seem to find it hard.

[16:30] HK tells about a simple program the pupils were to make in 2,5 hours, where only 1 or 2 pupils actually managed. She compares the desire from the

pupils, wanting to make this in the short amount of time given, to the desire of thinking one should be able to read the first day in school.

[17:30] Sometimes the limitations lie within the pupils themself, because they are not able to set up the permises of the program they want to create. The best result was when the task given would be relatively easy.

[19:00] Programming-wise there is a big difference on how much the pupils know as well as how well motivated they were towards programming. Some pupils only really learned how to put basic blocks together.

[20:00] The pupils have very great ambitions and want results imidiately, but do not really have the needed skills and knowledge.

HK states that the use of english (a non-native language) in programming might not be ideal, ie. if you were to create a line and you first had to know what what a line is called, you would be less prone to succeed.

[23:00] HK's response to the usage of '=' (for assignment) and '==' (for comparison) is that it is good to keep equally looking symbols apart, so the pupils do not mix their meanings.

[23:45] When asked directly, HK states that a reason for the lack of knowledge in english related to programming, might be the pupil's lack of interest in the field. She states that they have been taught english since 3rd grade but language might still be a barrier, ie. she mentions a dyslexic pupil from her class. She also states that a problem with programming is that you can not just write "I want a 23cm long line", you have to type it in a specific way.

[24:50] A thing HK has learned and want to change if she had to teach such a class again is to make it really simple and make a list of ie. 5 things, where the pupil then have to select one thing he/she wants to work with. this important for the teacher because elsewhere there is just too much functionallity available.

[27:00] A problem in the process of the pupils doing ie. Hour of Code is that if the pupil does something and it works, they do not stop to think why it worked, HK states. This happens when the task given is too easy, which leads to frustration when there is a harder task the pupil can not just solve in the same manner.

[28:30] In generel the pupils were very independent and not so prone to help each other. "this is my program and my computer"

[32:30] Visual output is very important. If you have to use maths to teach programming, it will be very dificult, HK states, to teach a language (maths) through another language (programming) is very difficult if the pupil does not know either, and if you then use english on top of this it only adds to the complexity.

15.3 Interview: Magnus Toftdal Lund

All interviews have been condensed and edited. Interview was performed as a semi-structured interview by group SW411F15, saved on disc. Interviewee: **Magnus Toftdal Lund (herafter refered to as MTL), head of Coding Pirates Aalborg (hereafter refered to as CP).**

Work: Senai Invest

Education: Revision and organisation/management. (in danish: Revisor ass. samt hd-o organisation og ledelse)

Motivation: ***I could have used such an offer when I was growing up.***

[04:00] MTL: I Belive that programming for the sake of programming will not help towards teaching programming. You learn to program because you can see some use of it - aha, Minecraft, I can tinker with that - and the children are off to program The children need a goal and have to see that programming helps them solve something they find useful.

[06:00] Our aproach in Coding Pirates is that when the teacher think something is fun, it will affect the children in the same direction.

[08:45] MTL states that it is important to have basic knowledge about what computers are good at and what they are not good at, in order to make good decisions when using them.

[11:15] The interest in learning programming lay with both the parents and children, except for the elder childern, they generally chose programming themselves.

[13:25] At CP textual programming is actually taught. In the start the children generally program in Scratch (visual programming), but soon figure that ie. Python meets their needs better while at the same time providing output to the screen in a short time of programming. Some children also leap from this stage towards ie. Java or C#. Already at the age of about 13they perform the transition into textual programming.

The interview-group tells they are to create a programming language which should work as a transition from ie. Scratch to textual programming where MTL answers: then you need something like a scripting language, like Javascript, PHP or HTML.

[15:35] The children generally learn by trial and error, and by splitting working code in parts and try to assemble something again.

[17:00] When MTL is asked about visual output as a motivation factor for the children to learn programming his answer is: I think you have to approach this in another way; ie. we use rewarding by giving a badge like "You managed to declare a variable, assign a value and print it" so the children can show their

friends what they have accomplished. It should not be a competition but the fact that there is some sort of prestige in it, this really seems to work.

[19:00] The concept of variables is hard for the children to be taught. Same goes for lists and loops - it is very abstract to them, and they have to use them in order to see why it is smart.

[20:00] In a way the children need a complete program and then disassemble this and assemble it again.

[20:45] Python is easy for the children because there is not so many strange things that is needed to be done and not strange curly paranthesis etc. Generally the children tend to easily see how one construct in one language correlates to another similar construct in another language (ie. float in Java-script and double in Java).

[22:00] The IDE is generally simple in construction, since we need to keep it simple. If the children can find a tutorial on youtube on a program they tend to want to use this and because most tutorials on ie. Visual Studio is more business oriented, they tend to skip these and as such do not want to use this.

[24:00] The children are really keen on english and feel a strong relation towards this language - this should not be underestimated, states MTL when questioned if keywords in a potential language should be in english.

[26:00] MTL: It is not important if it is visual or textual output, the importance is fast feedback.

[27:00] It is not important to replace symbols with words, ie. "==" and "is". they have to learn both in a way. [32:00] They know the symbols from maths so to construct it by words instead does not help the children. MTL states that using keywords would only lead to confusion of concepts of words used, ie. the word "set" which is also used in badminton.

[27:45] When adressing 13-16 it is a visual programming is a good starting point and then fast move on to textual programming. We even have 8 y/o that have no problem with progamming related to command-prompts.

[28:30] It is not important that we can program, however we need to be able to point them in the right direction.

[30:00] when asked about syntax, MTL answer: One of the things Hour of Code and Code Academy does right is: One concept at a time, instant visual feedback. That the children can see what happens and that they realise they are in control. This is by far the most important, then syntax is not so important. However he continue: the semi-colon in C# is a big "show-stopper" since they forget it every time because they come from a language where this is not important. A better compiler is in his opinion a more forgiving compiler, ie. in Java-script where the number 1 is the same as the string "1". The children generally find the concept of types, when to use which and why, along with them not being comparable, hard. MTL states that implicit conversion of types is a good idea.

[34:00] MTL: A CP quarter fee function as a motivation for the children to actually show up.

Interview was performed as a semi-structured interview by group sw411f14, saved on disc .

Half the children find it interesting when something is physical (makes a reference to the Banana space bar in the MaKey MaKey promotion video) and the other half when it is visual. Surprisingly many children does not get caught till they get their hands on it.

15.4 EBNF

Listing 15.1: full grammar

```
NewLine      : ( '\r'? '\n'+ | '\r' ) { skip();} ;

Space        : ( ' '+ | '\t' ) {skip();} ;

LINE_COMMENT : '//' .*? '\r'? '\n' -> skip ;

COMMENT      : '/*' .*? '*/' -> skip ;

start        : func+;

func         : 'function' TYPE? ID funcInput funcElements* 'endfunction' ;

funcInput    : LPAR inputPar? RPAR
             | ;

inputPar     : TYPE ID inputParRec*;

inputParRec  : ';' TYPE ID ;

funcElements: decl EOL
             | stmt;

decl         : TYPE? ID ('=' (exp | andOrExpr | DIRECTION | material))?;

stmt         : 'call' funcCall EOL
             | 'place' ( ID | NUM )? material DIRECTION? EOL
             | 'step' ( ID | NUM )? DIRECTION EOL
             | 'do' ( ID | NUM ) 'times' ( 'using' ID )? funcElements* 'enddo'
             | 'if' andOrExpr 'then' funcElements* elseif* elseEnd? 'endif'
             | 'return' ( ID | NUM | material | DIRECTION | BOOLVALUE ) EOL
             | 'origin' EOL;

elseif       : 'elseif' andOrExpr 'then' funcElements*;

elseEnd      : 'else' funcElements*;

andOrExpr    : boolExpr (ANDOR boolExpr)*;
boolExpr     : boolOpr (COMPARATOR boolOpr)*;
boolOpr      : ID | NUM | material | BOOLVALUE | LPAR andOrExpr RPAR
               | exp;

funcCall     : ID callInput? DIRECTION?;
```

```
callInput    : LPAR callInputPar? RPAR;

callInputPar: ( ID | NUM | DIRECTION | BOOLVALUE | material ) callInputParRe

callInputParRec: ';' ( ID | NUM | DIRECTION | BOOLVALUE | material );

exp          : aexp ;
aexp         : mexp ( OPRADDSUB mexp )* ;
mexp         : pexp ( OPRMULTDIV pexp )* ;
pexp         : ID | NUM | LPAR exp RPAR ;

material     : QUOT ID QUOT;

ANDOR        : '&'
             | '|';

DIRECTION    : 'forward'
             | 'back'
             | 'right'
             | 'left'
             | 'up'
             | 'down';

TYPE         : 'int'
             | 'block'
             | 'bool'
             | 'go'
             | 'void';

COMPARATOR   : '<'|'>'|'<='|'>='|'=='|'!=' ;

BOOLVALUE    : 'true'
             | 'false';

OPRADDSUB    : '+'
             | '-';

OPRMULTDIV   : '*'
             | '/';

NUM          : '-'? [0-9]+;

ID           : [a-zA-Z] ([a-zA-Z0-9] | '_' )* ;

LPAR         : '(' ;
```

```
RPAR        : ')' ;

EOL         : '.' ;

QUOT        : '"' ;
```

15.5 BNF

Listing 15.2: full rewritten grammar in BNF format

```
start -> func

func -> 'function' typeMore ID funcInput funcElements 'endfunction' funcMore
funcMore -> 'function' typeMore ID funcInput funcElements 'endfunction' funcMore
          |EPSILON

funcInput -> LPAR inputPar RPAR | EPSILON

inputPar -> TYPE ID inputParRec inputPar | EPSILON

inputParRec  -> ';' TYPE ID  inputParRec | EPSILON

funcElements -> decl EOL funcElements
                | stmt funcElements
                | EPSILON

decl -> typeMore ID assignment

assignment -> '=' assignmentMore | EPSILON
assignmentMore -> exp | andOrExpr | DIRECTION | material

stmt -> 'call' funcCall EOL
        | 'place' idOrNum  material DIRECTION? EOL
        | 'step' idOrNum  DIRECTION EOL
        | 'do' idNum 'times' usingID funcElements 'enddo'
        | 'if' andOrExpr 'then' funcElements elseif elseEnd 'endif'
        | 'return' ( ID | NUM | material | DIRECTION | BOOLVALUE ) EOL
        | 'origin' EOL

elseif -> 'elseif' andOrExpr 'then' funcElements elseif | EPSILON

elseEnd -> 'else' funcElements | EPSILON

funcCall -> ID callInput directionMore
directionMore -> DIRECTION | EPSILON
```

```
callInput -> LPAR callInputPar RPAR | EPSILON
callInputPar -> inputOptions callInputParRec | EPSILON
callInputParRec -> ';' inputOptions callInputParRec | EPSILON
inputOptions -> ID | NUM | DIRECTION | BOOLVALUE | material

andOrExpr -> boolExpr andOrMore
andOrMore -> ANDOR boolExpr andOrMore | EPSILON
boolExpr -> boolOpr boolExprMore
boolExprMore -> COMPARATOR boolOpr boolExprMore  | EPSILON
boolOpr -> ID | NUM | material | BOOLVALUE | LPAR andOrExpr RPAR | exp

exp          -> aexp
aexp         -> mexp aexpMore
aexpMore     -> OPRADDSUB mexp aexpMore | EPSILON
mexp         -> pexp mexpMore
mexpMore     -> OPRMULTDIV pexp mexpMore | EPSILON
pexp         -> ID | NUM | LPAR exp RPAR

idNumEpsilon -> ID | NUM | EPSILON
idOrNum -> ID | NUM
usingID -> 'using' ID | EPSILON
typeMore -> TYPE | EPSILON
directionMore -> DIRECTION | EPSILON
material -> QUOT ID QUOT
```

15.6 First and followset

Productions	**First set**	**Follow set**
start	'function'	
func	'function'	
funcMore	'function'	$
funcInput	LPAR, ϵ	ID, TYPE
inputPar	TYPE, ϵ	RPAR
inputParRec	';', ϵ	TYPE, RPAR
funcElements	ID, TYPE, ϵ	'elseif', 'endfunction'
decl	ID, TYPE, ϵ	
assignment	'=', ϵ	EOL
assignmentMore	DIRECTION, ID, NUM, LPAR, QUOT, BOOLVALUE	
stmt	'call'	
elseif	'elseif', ϵ	
elseEnd	'else', ϵ	
andOrExpr	ID, NUM, BOOLVALUE, LPAR	
andOrMore	ANDOR, ϵ	RPAR, 'then', EOL
boolExpr	ID, NUM, BOOLVALUE, LPAR	
boolExprMore	COMPARATOR, ϵ	ANDOR, RPAR, 'then', EOL
boolOpr	ID, NUM, BOOLVALUE, LPAR, QUOT	
exp	ID, NUM, LPAR	
aexp	ID, NUM, LPAR	
aexpMore	OPRADDSUB, ϵ	RPAR, COMPARATOR, ANDOR, 'then', EOL
mexp	ID, NUM, LPAR	
mexpMore	OPRMULTDIV, ϵ	OPRADDSUB, RPAR, COMPARATOR, ANDOR, 'then', EOL
pexp	ID, NUM, LPAR	
funcCall	ID	
callInput	LPAR, ϵ	DIRECTION, EOL
callInputPar	ϵ, ID, NUM, DIRECTION, BOOLVALUE, QUOT	RPAR
callInputParRec	';', ϵ	RPAR
inputOptions	ID, NUM, DIRECTION, BOOLVALUE, QUOT	
material	QUOT	
typeMore	TYPE, ϵ	ID
directionMore	DIRECTION, ϵ	EOL
usingID	'using', ϵ	
idNumEpsilon	ID, NUM, ϵ	
idNum	ID, NUM	

Table 15.2: First and follow set for rewritten grammar

15.7 Transitional rules for operational semantics

Listing 15.3: VALUE transition rules. Denoted $\rightarrow_v$

$[VALUE - N_{BSS}]$
$env_V, sto \vdash n \rightarrow_v v$
$if\ T(n) = v$

$[VALUE - M_{BSS}]$
$env_V, sto \vdash m \rightarrow_v v$
$if\ T(m) = v$

$[VALUE - D_{BSS}]$
$env_V, sto \vdash d \rightarrow_v v$
$if\ T(d) = v$

$[VALUE - B_{BSS}]$
$env_V, sto \vdash b \rightarrow_v v$
$if\ T(b) = v$

$[VALUE - X_{BSS}]$
$env_V, sto \vdash x \rightarrow_v v$
$if\ T(x) = v$
$where\ v = Var$

$[VALUE - X - TO - NUM_{BSS}]$
$env_V, sto \vdash x_{value} \rightarrow_v n$
$if\ x_{type} = int$

Listing 15.4: FUNCTION transition rules. Denoted $\rightarrow_f$

$[FUNC - DECL_{BSS}]$
$env_V, env_D, env_W \vdash$
$\langle Dp, env_P[P \mapsto (stmt, \vec{x}, env_V, env_D, env_W, env_P, returnType)]\rangle \rightarrow env'_P$
$\overline{env_V, env_D, env_W \vdash \langle functionpreturnType(\vec{x}, env_P)\rangle \rightarrow env'_P}$

$[FUNC - CALL_{BSS}]$
$\dfrac{env_V, sto \vdash x_i \rightarrow v_i \quad env_W, env_D, env_V, env_P \vdash \langle K, sto\rangle \rightarrow sto'}{env_W, envD, env_V, env_P \vdash \langle p(\vec{x}), sto\rangle \rightarrow sto'}$

$where 1 \leq i \leq |\vec{x}|$
$and \vec{x} = (x_1 ... x_j)$
$|\vec{x}| = j$
$K = functionbody$

Listing 15.5: DECLARATION transition rules. Denoted $\rightarrow_{dv}$

$[DECL - ASSIGN_{BSS}]$
$env_V, sto \vdash type \rightarrow v_1$

$$\frac{env_V, sto \vdash Value \rightarrow v_2 \quad \langle Dv, env''_v, sto[l \mapsto Var(v_1, v_2)]\rangle \rightarrow_{Dv} (env'_V, sto')}{\langle type\ x = value, Dv, env_V, sto\rangle \rightarrow_{Dv} (env'_V, sto')}$$

$$if\ v_2 \in T(v_1)\ where\ l = env_V\ next\ and\ env''_V = env_V[x \mapsto l][next \mapsto new\ l]$$

[$DECL_{BSS}$]

$$\frac{env_V, sto \vdash type \rightarrow v \quad \langle Dv, env''_V, sto[l \mapsto Var(v, null)]\rangle \rightarrow_{Dv} (env'_V, sto')}{\langle type\ x, Dv, env_V, sto\rangle \rightarrow_{Dv} (env'_V, sto')}$$

$$if v_2 \in T(v_1)\ where\ l = env_V\ next\ and\ env''_V = env_V[x \mapsto l][next \mapsto newl]$$

[$ASSIGN_{BSS}$]

$$\frac{env_V, sto \vdash x_{type} \rightarrow v_1 \quad env_V, sto \vdash value \rightarrow v_2 \quad \langle Dv, env_V, sto[l \mapsto Var(v, null)]\rangle \rightarrow_{Dv} (env_V, sto')}{\langle x = value, Dv, env_V, sto\rangle \rightarrow_{Dv} (env_V, sto')}$$

$$if v_2 \in T(v_1)\ where\ l = env_V\ next\ and\ sto' = sto[l \mapsto value]$$

Listing 15.6: ARITHMETIC transition rules. Denoted $\rightarrow_a$

[$PLUS_{BSS}$]

$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 + aexp_2 \rightarrow_a v} \quad where\ v = v_1 + v_2$$

[$MINUS_{BSS}$]

$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 - aexp_2 \rightarrow_a v} \quad where\ v = v_1 - v_2$$

[$MULT_{BSS}$]

$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 * aexp_2 \rightarrow_a v} \quad where\ v = v_1 * v_2$$

[DIV_{BSS}]

$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1/aexp_2 \rightarrow_a v} \quad where\ v = v_1/v_2$$

[$PARENT_{BSS}$]

$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1}{env_V, sto \vdash (aexp_1) \rightarrow_a v_1}$$

[NUM_{BSS}]

$$env_V, sto \vdash n \rightarrow_a v \quad if\ T[n] = v$$

$$[VAR_{BSS}]$$
$$env_V, sto \vdash x \rightarrow_v v \quad if\ env_V\ x = l \quad and\ sto\ l_{type} = int \quad and\ sto\ l_{value} = v$$

Listing 15.7: BOOL transition rules. Denoted $\rightarrow_b$

$$[EQUAL - 1_{BSS}]$$
$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 == aexp_2 \rightarrow_b tt} \quad if\ v_1 = v_2$$

$$[EQUAL - 2_{BSS}]$$
$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 == aexp_2 \rightarrow_b ff} \quad if\ v_1 \neq v_2$$

$$[NOTEQUAL - 1_{BSS}]$$
$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 != aexp_2 \rightarrow_b tt} \quad if\ v_1 \neq v_2$$

$$[NOTEQUAL - 2_{BSS}]$$
$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 != aexp_2 \rightarrow_b ff} \quad if\ v_1 = v_2$$

$$[SMALTHAN - 1_{BSS}]$$
$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 < aexp_2 \rightarrow_b tt} \quad if\ v_1 < v_2$$

$$[SMALTHAN - 2_{BSS}]$$
$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 < aexp_2 \rightarrow_b ff} \quad if\ v_1 \not< env_V, stov_2$$

$$[EQUALSMALTHAN - 1_{BSS}]$$
$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 \leq aexp_2 \rightarrow_b tt} \quad if\ v_1 \leq v_2$$

$$[EQUALSMALTHAN - 2_{BSS}]$$
$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 \leq aexp_2 \rightarrow_b ff} \quad if\ v_1 \not\leq v_2$$

$$[GREATTHAN - 1_{BSS}]$$
$$\frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 > aexp_2 \rightarrow_b tt} \quad if\ v_1 > v_2$$

$$[GREATTHAN - 2_{BSS}]$$
$$env_V, sto \vdash aexp_1 \rightarrow_a v_1$$

$$\frac{env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 > aexp_2 \rightarrow_b ff} \quad if\ v_1 \not> v_2$$

$$[EQUALGREATTHAN-1_{BSS}]\ \frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 \geq aexp_2 \rightarrow_b tt} \quad if\ v_1 \geq v_2$$

$$[EQUALGREATTHAN-2_{BSS}]\ \frac{env_V, sto \vdash aexp_1 \rightarrow_a v_1 \quad env_V, sto \vdash aexp_2 \rightarrow_a v_2}{env_V, sto \vdash aexp_1 \geq aexp_2 \rightarrow_b ff} \quad if\ v_1 \not\geq v_2$$

$$[AND-1_{BSS}]\ \frac{env_V, sto \vdash bexp_1 \rightarrow_b tt \quad env_V, sto \vdash bexp_2 \rightarrow_b tt}{env_V, sto \vdash bexp_1 \& bexp_2 \rightarrow_b tt}$$

$$[AND-2_{BSS}]\ \frac{env_V, sto \vdash bexp_i \rightarrow_b ff}{env_V, sto \vdash bexp_1 \& bexp_2 \rightarrow_b ff} \quad i \in \{1, 2\}$$

$$[OR-1_{BSS}]\ \frac{env_V, sto \vdash bexp_i \rightarrow_b tt}{env_V, sto \vdash bexp_1 | bexp_2 \rightarrow_b tt}$$

$$[OR-2_{BSS}]\ \frac{env_V, sto \vdash bexp_1 \rightarrow_b ff \quad env_V, sto \vdash bexp_2 \rightarrow_b ff}{env_V, sto \vdash bexp_1 | bexp_2 \rightarrow_b ff}$$

$$[PARENT_{BSS}]\ \frac{env_V, sto \vdash bexp_1 \rightarrow_b v}{env_V, sto \vdash (bexp_1) \rightarrow_b v}$$

Listing 15.8: STATEMENT transition rules. Denoted $\rightarrow_s$

$$[PLACE-N=1-M_{BSS}]\ \frac{env_V, sto \vdash n \rightarrow_v v_1 \quad env_V, sto \vdash m \rightarrow_v v_2 \quad env_V, sto \vdash d \rightarrow_v v_3}{env_W \vdash \langle place\ n\ m\ d, sto \rangle \rightarrow sto'}$$

$if\ v_1 = 1$
$if\ n = null\ then\ v_1 = 1$
$if\ d = null\ then\ v_3 = forward$
$Where\ env_W(env_D(v_3) = L)$
$sto' = sto[L_{material} \mapsto v_2]$

$$[PLACE-N>1-M-D_{BSS}]\ \frac{\begin{array}{l} env_V, sto \vdash n \rightarrow_v v_1 \\ env_V, sto \vdash m \rightarrow_v v_2 \\ env_V, env_D \vdash d \rightarrow_v v_3 \\ env_W, env_D \vdash \langle place\ n\ m\ d, sto'' \rangle \rightarrow sto' \end{array}}{env_W, sto \vdash \langle place\ n\ m\ d, sto \rangle \rightarrow sto'}$$

$if\ n > 1$
$env_W(env_D(v_3)) = l$
$sto'' = sto[env_W(env_D(v_3))_{material} \mapsto v2][env_D(x) \mapsto l_x]$
$[env_D(y) \mapsto l_y][env_D(z) \mapsto l_z][n \mapsto n-1]$
$if\ d = null\ then\ v_3 = forward$
$env''_D = env_D[forward \mapsto newForward][back \mapsto newBack]$
$[right \mapsto newRight][left \mapsto newLeft][up \mapsto newUp][down \mapsto newDown]$

$[PLACE - X = 1 - M_{BSS}]$

$$\frac{\begin{array}{c} env_V, sto \vdash x_{value} \rightarrow_v v_1 \\ env_V, sto \vdash m \rightarrow_v v_2 \\ env_V \vdash d \rightarrow_v v_3 \end{array}}{env_W \vdash \langle place\ x\ m\ d, sto\rangle \rightarrow sto'}$$

$if\ v_1 = 1$
$if\ x = null\ then\ v_{1value} = 1$
$if\ d = null\ then\ v_3 = forward$
$Where\ env_W(env_D(v_3)) = l$
$sto' = sto[l_{material} \mapsto v_2]$

$[PLACE - X > 1 - M - D_{BSS}]$

$$\frac{\begin{array}{c} env_V, sto \vdash x_{value} \rightarrow_v n \\ env_V, sto \vdash n \rightarrow_v v_1 \\ env_V, sto \vdash m \rightarrow_v v_2 \\ env_V, env_D \vdash d \rightarrow_v v_3 \\ env_W, env_D \vdash \langle place\ x\ m\ d, sto''\rangle \rightarrow sto' \end{array}}{env_W, sto \vdash \langle place\ x\ m\ d, sto\rangle \rightarrow sto'}$$

$if\ x_{type} = int\ and\ n > 1$
$env_W(env_D(v_3)) = l$
$sto'' = sto[env_W(env_D(v_3))_{material} \mapsto v2][env_D(x) \mapsto l_x]$
$[env_D(y) \mapsto l_y][env_D(z) \mapsto l_z][n \mapsto n-1]$
$if\ d = null\ then\ v_3 = forward$
$env''_D = env_D[forward \mapsto newForward][back \mapsto newBack][right \mapsto newRight]$
$[left \mapsto newLeft][up \mapsto newUp][down \mapsto newDown]$

$[STEP - N = 1 - D_{BSS}]$

$$\frac{\begin{array}{c} env_V, sto \vdash n \rightarrow_v v_1 \\ env_V, sto \vdash d \rightarrow_v v_2 \end{array}}{env_W, env_D \vdash \langle step\ n\ d, sto\rangle \rightarrow sto'}$$

$if\ v_1 = 1$
$env_W(env_D(v_2)) = l$
$sto'' = sto[env_D(x) \mapsto l_x][env_D(y) \mapsto l_y][env_D(z) \mapsto l_z]$

$[STEP - N > 1 - D_{BSS}]$

$$\frac{\begin{array}{c} env_V, sto \vdash n \rightarrow_v v_1 \\ env_V, sto \vdash d \rightarrow_v v_2 \\ env_W, env''_D, sto \vdash \langle step\ n\ d, sto''\rangle \rightarrow sto' \end{array}}{env_W, env_D \vdash \langle step\ n\ d, sto\rangle \rightarrow sto'}$$

$if\ ston > 1$
$env_W(env_D(v_2)) = l$
$sto'' = sto[env_D(x) \mapsto l_x][env_D(y) \mapsto l_y][env_D(z) \mapsto l_z][n \mapsto n-1]$
$env''_D = env_D[forward \mapsto newForward][back \mapsto newBack][right \mapsto newRight]$
$[left \mapsto newLeft][up \mapsto newUp][down \mapsto newDown]$

$$[STEP - X = 1 - D_{BSS}]$$
$$\frac{\begin{array}{l} env_V, sto \vdash x_{value} \rightarrow_v v_1 \\ env_V, sto \vdash d \rightarrow_v v_2 \end{array}}{env_W, env_D, sto \vdash \langle step\ x\ d, sto \rangle \rightarrow sto'}$$

$$if\ v_1 = 1$$
$$env_W(env_D(v_2)) = l$$
$$sto' = sto[env_D(x) \mapsto l_x][env_D(y) \mapsto l_y][env_D(z) \mapsto l_z]$$

$$[STEP - X > 1 - D_{BSS}]$$
$$\frac{\begin{array}{l} env_V, sto \vdash x_{value} \rightarrow_v n \\ env_V, sto \vdash n \rightarrow_v v_1 \\ env_V, sto \vdash d \rightarrow_v v_2 \\ env_W, env_D, sto \vdash \langle step\ n\ d, sto'' \rangle \rightarrow sto' \end{array}}{env_W, env''_D, sto \vdash \langle step\ x\ d, sto \rangle \rightarrow sto'}$$

$$if\ sto\ n > 1\ and\ x_{type} = int$$
$$env_W(env_D(v_2)) = l$$
$$sto'' = sto[env_D(x) \mapsto l_x][env_D(y) \mapsto l_y][env_D(z) \mapsto l_z][n \mapsto n - 1]$$
$$env''_D = env_D[forward \mapsto newForward][back \mapsto newBack]$$
$$[right \mapsto newRight][left \mapsto newLeft][up \mapsto newUp][down \mapsto newDown]$$

$$[DO - VAR_{BSS}]$$
$$\frac{\begin{array}{l} env_V, sto \vdash x_{value} \rightarrow_v n \\ env_V, sto \vdash n \rightarrow_v v \\ env_V, env_D, env_P, env_W \vdash \langle do\ x\ times\ K\ enddo, sto'' \rangle \rightarrow sto' \end{array}}{env_V, env_D, env_P, env_W \vdash \langle do\ x\ times\ K\ enddo, sto \rangle \rightarrow sto'}$$

$$if\ x_{type} = int\ and\ sto\ n > 0$$
$$sto'' = \langle K, sto''' \rangle$$
$$sto''' = sto[n \mapsto n - 1]$$

$$[DO - NUM_{BSS}]$$
$$\frac{\begin{array}{l} env_V, sto \vdash n \rightarrow_v v \\ env_V, env_D, env_P, env_W \vdash \langle do\ n\ times\ K\ enddo, sto'' \rangle \rightarrow sto' \end{array}}{env_V, env_D, env_P, env_W \vdash \langle do\ n\ times\ K\ enddo, sto \rangle \rightarrow sto'}$$

$$sto\ n > 0$$
$$sto'' = \langle K, sto''' \rangle$$
$$sto''' = sto[n \mapsto n - 1]$$

$$[DO - USING - VAR_{BSS}]$$
$$\frac{\begin{array}{l} env_V, sto \vdash x_{1value} \rightarrow_v n \\ env_V, sto \vdash n \rightarrow_v v_1 \\ env_V, sto \vdash x_2 \rightarrow_v v_2 \\ env_V, env_D, env_P, env_W \vdash \langle do\ x_1\ times\ using\ x_2\ K\ enddo, sto'' \rangle \rightarrow sto' \end{array}}{env_V, env_D, env_P, env_W \vdash \langle do\ x_1\ times\ using\ x_2\ K\ enddo, sto \rangle \rightarrow sto'}$$

$$if\ x_{1type} = int\ and\ x_{2type} = int\ and\ sto\ n > 0$$
$$sto'' = \langle K, sto''' \rangle$$
$$sto''' = sto[n \mapsto n - 1][x_{2value} \mapsto x_{2value} + 1]$$

$$[DO - USING - NUM_{BSS}]$$
$$\frac{\begin{array}{l} env_V, sto \vdash n \rightarrow_v v_1 \\ env_V, sto \vdash x \rightarrow_v v_2 \\ env_V, env_D, env_P, env_W \vdash \langle do\ n\ times\ using\ x\ K\ enddo, sto'' \rangle \rightarrow sto' \end{array}}{env_V, env_D, env_P, env_W \vdash \langle do\ n\ times\ using\ x\ K\ enddo, sto \rangle \rightarrow sto'}$$

$if\ x_{type} = int\ and\ sto\ n > 0$
$sto'' = \langle K, sto''' \rangle$
$sto''' = sto[n \mapsto n-1][x_{value} \mapsto x_{value} + 1]$

$[IF - 1_{BSS}]$
$$\frac{env_V, env_P \vdash \langle K_1, sto \rangle \to sto'}{env_V, env_P \vdash \langle if\ bexp\ then\ K_1, sto \rangle \to sto'} \quad if\ env_V, sto \vdash bexp \to_b tt$$

$[IF - 2_{BSS}]$
$$\frac{env_V, env_P \vdash \langle K_1, sto \rangle \to sto}{env_V, env_P \vdash \langle if\ bexp\ then\ K_1, sto \rangle \to sto} \quad if\ env_V, sto \vdash bexp \to_b ff$$

$[ELSEIF - 1_{BSS}]$
$$\frac{env_V, env_P \vdash \langle K_1, sto \rangle \to sto'}{env_V, env_P \vdash \langle elseif\ bexp\ then\ K_1, sto \rangle \to sto'} \quad if\ env_V, sto \vdash bexp \to_b tt$$

$[ELSEIF - 2_{BSS}]$
$$\frac{env_V, env_P \vdash \langle K_1, sto \rangle \to sto}{env_V, env_P \vdash \langle elseif\ bexp\ then\ K_1, sto \rangle \to sto} \quad if\ env_V, sto \vdash bexp \to_b ff$$

$[ELSE_{BSS}]$
$$\frac{env_V, env_P \vdash \langle K_1, sto \rangle \to sto'}{env_V, env_P \vdash \langle else\ bexp\ then\ K_1, sto \rangle \to sto'}$$

$[RETURN - X_{BSS}]$
$$\frac{env_V, sto \vdash x \to v_1}{env_P, sto \vdash \langle return\ x, sto \rangle \to sto}$$

$if\ env_P, sto \vdash ReturnType = env_V, sto \vdash v_{1type}$

$[RETURN - N_{BSS}]$
$$\frac{env_V, sto \vdash n \to v_1}{env_P, sto \vdash \langle return\ n, sto \rangle \to sto}$$

$if\ sto(env_P(ReturnType) = int)$

$[RETURN - M_{BSS}]$
$$\frac{env_V, sto \vdash m \to v_1}{env_P, sto \vdash \langle return\ m, sto \rangle \to sto}$$

$if\ sto(env_P(ReturnType) = material)$

$[RETURN - D_{BSS}]$
$$\frac{env_V, sto \vdash d \to v_1}{env_P, sto \vdash \langle return\ d, sto \rangle \to sto}$$

$if\ sto(env_P(ReturnType) = go)$

$[RETURN - B_{BSS}]$
$$\frac{env_V, sto \vdash b \to v_1}{env_P, sto \vdash \langle return\ b, sto \rangle \to sto}$$

$if\ sto(env_P(ReturnType) = bool)$

$[ORIGIN_{BSS}]$
$env_D, sto \vdash sto(env_D(x)) \to v_1$

$$\frac{\begin{array}{l} env_D, sto \vdash sto(env_D(y)) \to v_2 \\ env_D, sto \vdash sto(env_D(z)) \to v_3 \end{array}}{env_D, sto \vdash \langle origin, sto \rangle \to sto'}$$

$where\ env_D(xs) = l_x\ and\ env_D(ys) = l_y\ and\ env_D(zs) = l_z$
$sto' = sto[l_x \mapsto v_1][l_y \mapsto v_2][l_z \mapsto v_3]$

15.8 Use of the language exammple

Listing 15.9: An example of the code in the ML8 language. The functions together builds a house of any given length, width and height, when calling house

```
function base(int length; int width)
    do width times
        place length "wood" forward.
        step length back.
        step 1 right.
    enddo
    origin.
endfunction

function wall(int length; int height; int width)
    int newLength.
    int newWidth.
    length = length - 1.
    if length - (2 * (length/2)) == 0 then
        length = length / 2.
        newLength = length - 1.
    else
        length = length / 2.
        newLength = length.
    endif

    width = width - 1.
    if width - (2 * (width/2)) == 0 then
        width = width / 2.
        int newWidth = width - 1.
    else
        width = width / 2.
        int newWidth = width.
    endif

    int i.

    do height times using i
        place length "stone" forward.
        place 1 "window" forward.
        place newLength "stone" forward.
        place width "stone" right.
        place 1 "window" right.
        place newWidth "stone" right.
        place length "stone" back.
        place 1 "window" back.
        place newLength "stone" back.
        place width "stone" left.
        if i <= 2 then
            step 1 left.
        else
            place 1 "stone" left.
        endif
        place newWidth "stone" left.
        step 1 up.
    enddo
```

```
52  endfunction
53
54  function roof(int width; int length)
55      int newWidth = width / 2.
56      length = length - 1.
57      width = width - 1.
58
59      do newWidth times
60          place length "wool" forward.
61          place width "wool" right.
62          place length "wool" back.
63          place width "wool" left.
64          length = length - 2.
65          width = width - 2.
66          step 1 up.
67          step 1 right.
68          step 1 forward.
69      enddo
70      origin.
71  endfunction
72
73  function house(int length; int width; int height)
74      int newWidth = width / 2.
75      call base(length; width).
76      step up.
77      call wall(length; height; width).
78      call roof(width; length).
79      step back.
80      step height down.
81      step 1 down.
82      step newWidth right.
83      place 1 "door" forward.
84  endfunction
```